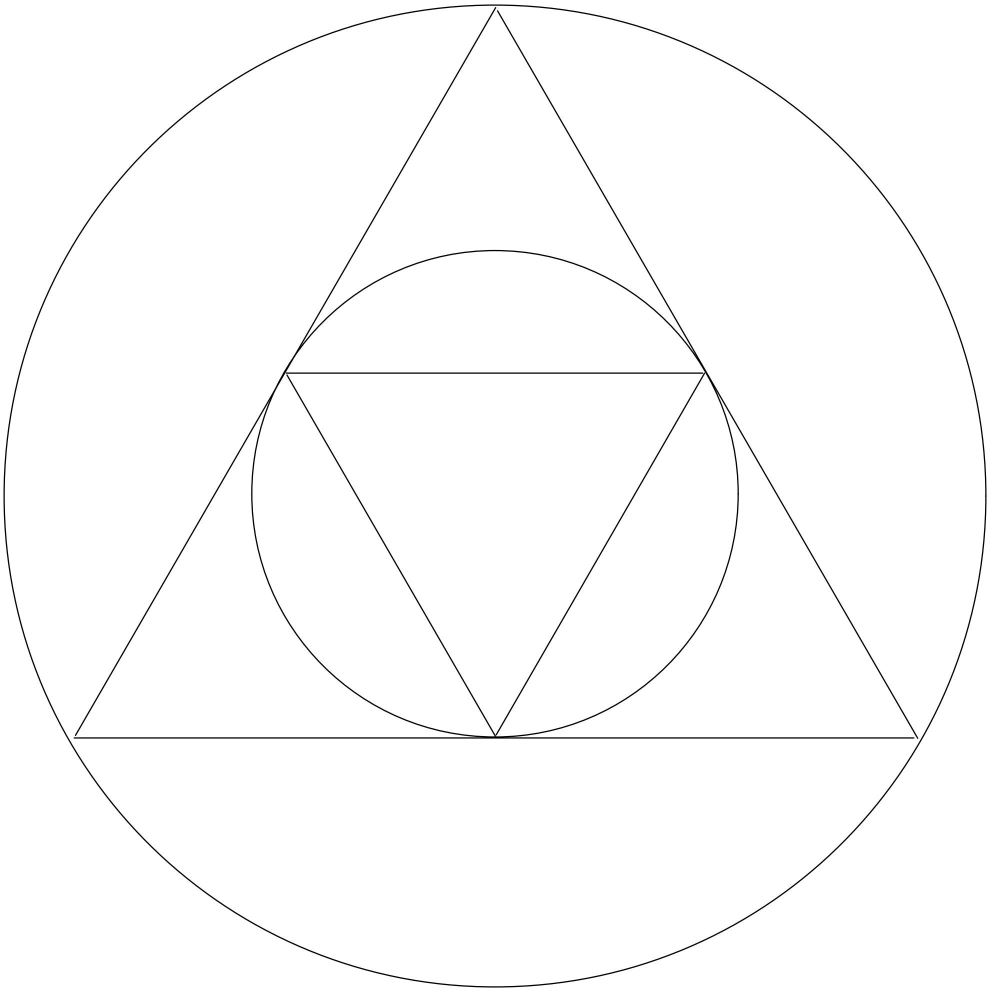


Syntax Definition for Language Prototyping

Syntax Definition for Language Prototyping

Syntax Definition for Language Prototyping

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr J. J. M. Franse
ten overstaan van
een door het college van decanen ingestelde commissie
in het openbaar te verdedigen
in de Aula der Universiteit
op dinsdag 23 september 1997
te 13.30 uur

door

Eelco Visser

geboren te Rijswijk (ZH)

Promotor: prof. dr P. Klint

Faculteit Wiskunde, Informatica, Natuurkunde en Sterenkunde
Universiteit van Amsterdam
Kruislaan 403
1098 SJ Amsterdam

NWO logo

This research was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO). Project 612-317-420: Incremental parser generation and context-sensitive disambiguation: a multi-disciplinary perspective.

ISBN 90-74795-75-7

Printed by Ponsen & Looijen, Wageningen

Author's address:
Nieuwe Leliestraat 140/3
1015 SX Amsterdam
The Netherlands
visser@acm.org

To my parents

Preface

This thesis is one of the results of the project *Incremental parser generation and context-sensitive disambiguation: a multi-disciplinary perspective*, supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO). The other main result is the thesis of Annius Groenink (1997) who was the other OIO on the project.

My side of the project can be divided in three main lines of research. In the first place I have been concerned with techniques for parsing and disambiguation of context-free languages targeted at the design of programming languages. This led to a number of results on disambiguation, including a general framework for studying disambiguation methods and a number of new parsing techniques. Furthermore, the syntax definition formalism SDF was redesigned and improved as the formalism SDF2.

The second line of research was concerned with the extension of first-order many-sorted algebraic specifications to include polymorphism and higher-order functions. This research followed-up on research done for my Master's thesis (Visser, 1993) that was partly dedicated to the study of higher-order functions in a first-order framework. This resulted in a general interest in the area of type systems and a particular interest in multi-level specifications. The result was the design of a multi-level algebraic specification formalism.

The third line combined the previous two in the study of two-level grammars for the purpose of polymorphic syntax definition.

As a sideline I have always been interested in typesetting and literate programming. I continued to maintain the literate programming tool ToLaTeX that I took over from Paul Klint during the work on my Master's thesis. The wish to improve the typesetting of terms in equations of specifications led to a cooperation with Mark van den Brand who was working on the generation of formatters. The cooperation was very fruitful and produced several results. A box calculus was developed that was used to translate box terms for pretty-printing to TeX code (Van den Brand and Visser, 1994). A paper on the formatter generator was published in the ACM Transactions on Software Engineering and Methodology (Van den Brand and Visser, 1996). This work, although related to syntax definition, is not represented in this thesis, except indirectly in the literate specifications in Part II and Part III.

Acknowledgments

Many people contributed to the development of this thesis. Here I would like to thank them.

Paul Klint, my promotor, has created an active and stimulating research environment at the University of Amsterdam and CWI. His lecture on the generation of programming environments introduced program generation, the mysteries of dynamic syntax and ASF+SDF. I was hooked at once. Many of the ideas in this thesis have their origin in our discussions. We wrote a joint paper on disambiguation filters and had many discussions on the intricacies of ATerms and AsFix. His suggestion to forget about scanners when I had just specified a scanner generator, referring to the work of Salomon and Cormack (1989), turned out to generate a lot of interesting research. Most importantly he gave me the freedom to find my own way in research and he didn't complain too much when I spent time on multi-level specifications and polymorphic syntax definition instead of parsing. In the end, despite my resistance, he expertly tricked me into finishing this thesis.

I thank the members of the reading committee Krzysztof Apt, Jan Bergstra, Jan van Eijck, Karl Meinke, and Anton Nijholt for refereeing this thesis.

Jan van Eijck was one of the proposers of the NWO/SION project. His last minute questions and remarks about reject productions considerably improved Chapter 3.

Mark van den Brand was my supervisor in the first years. I learned a lot about parsing from him. We cooperated on formatting and typesetting of program documentation tools and wrote several joint papers on the subject. He expertly managed the infrastructure of the programming research group.

Annius Groenink was my partner in the NWO project. In those early discussions ideas such as disambiguation filters and polymorphic syntax originate, although they were not recognizable as such at the time. Each of us worked out the ideas in different ways. At the last moment came his idea of using reject to define intersection of context-free languages.

Klaas Sikkel's parsing schemata framework turned the twilight zone of parsing algorithms into a bright and clear landscape.

With Arie van Deursen I had many discussions on (the future of) algebraic specification. He was always good for a broad gesture and idem smile mocking computer science ideologies. He was a critical reader of several of the chapters in this thesis and some other stuff that vanished. He invited me to write a chapter on specification of functional languages for his Language Prototyping book. This turned out, quite differently and quite a bit longer than intended, as the design of a multi-level specification formalism.

Dinesh could utterly confuse me by his stubborn questions and statements that he refused to explain. In the end there was often a good idea or insight underlying his oracles. His ideas about error messages for typechecking have been applied in the specification of multi-level specifications.

Jan Heering was always willing to answer a question, which would invariably turn into afternoon long discussion of a wide range of subjects, from the his-

torical design choices in SDF to publishing strategies and trends in scientific funding.

Jan Bergstra's views on software engineering inspired me to get my master's degree at the Programming Research Group. His views on language design have influenced many design decisions in the work described in this thesis.

Wilco Koorn initiated me to the subtleties and undocumented features of ASF+SDF.

Susan Üsküdarlı was my first roommate. Despite this, we became good friends. We would even sometimes cut back on the hours spent on our scientific quest to throw some time at a quest for a restaurant with just the right food for the night, which should not be regarded as much easier.

Pieter Olivier was my second roommate. We initiated PETR and the use of CVS and he supported my idea to include layout in parse trees. He almost always knew the answer to my programming queries.

The daily coffee breaks with Bas Luttik, justified by RSI paranoia, formed a welcome diversion from writing. They even led to a scientific cooperation which resulted in a joint paper on rewriting strategies (Luttik and Visser, 1997). He also relieved me of some of the organizational burdens of a graduation by acting as paranimf.

Tobias Kuipers used the SGLR parser in his structure editor, which turned out to be a good test. His scrutiny will undoubtedly turn up an error as soon as he opens this thesis. Maybe I shouldn't give him a copy.

Merijn de Jonge enthusiastically tested the ideas about renamings by building a renaming tool for ASF+SDF specifications. This resulted in an improvement of the definition of renaming in SDF2, although not all anomalies have been solved.

Several people, other than the members of the program committee, have read parts or all of this thesis and commented on it. They include at least: Mark van den Brand, Arie van Deursen, Dinesh, Annius Groenink, Pieter Hartel, Merijn de Jonge, Jasper Kamperman, Tobias Kuipers, Teo Rus, Alex Sellink and several anonymous referees of the various papers.

Many other people in the surroundings of the Programming Research groups at UvA and CWI contributed in some way or another to my development as a researcher, in general and to this thesis, in particular, I mention: Huub Bakker, Doeko Bosscher, Jacob Brunekreef, Casper Dik, David Griffioen, Calin Groza, Pieter Hartel, Joris Hillebrand, Jasper Kamperman, Ralf Lämmel, Job Ganzevoort, Hugh McEvoy, Sjouke Mauw, Emma van der Meulen, Jon Mountjoy, Dimitri Naidich, Martijn Res, Piet Rodenburg, Judi Romijn, Frank Tip, Leon Moonen, Alban Ponse, Alex Sellink, Chris Verhoef, Bas van Vlijmen, Pum Walters and Jos van Wamel.

At the computer science department of the University of Amsterdam, the computers are always up or if they do go down they are up in no time again. They just work. This seems obvious, but one hears these horror stories. Thanks for the systems group for that. In particular, to Jan Wortelboer for all his help and to Gert Poletiek for keeping the LaTeX installation up to date. The people at the secretariat were always willing to make another series of reservations for the PEMs. Hugo Suidgeest and Monique Kleinendorst smoothly arranged the trips to workshops and conferences.

Fien McColl encouraged me in pursuing a research career and in the first not so easy years of doing research. Angelo Welling, Willemiek Kluifhout and Selma Visser were always there. Ingmar Visser was one of the few outside the university who could at least vaguely understand what I was doing. As a paranimf he helped me with the organization of the graduation. Willemijn van der Laan stood by me in the last stage of writing this thesis. She kept me sane. Being together with her I could sometimes even forget this book. Wim and Connie Visser have always supported me in many ways. To them I dedicate this thesis.

Finally, one of the great contributors to this thesis is the deadline. Soft deadlines, firm deadlines, extended deadlines, nearing deadlines, changed deadlines, passed deadlines, the empowering deadlines that make page after page appear as if by magic, the deadlines that spook at night, deadlines that you can really feel, deadlines that are suddenly there. Now, writing the last words of this book, all pages printed except this one, I look back nostalgically. I know there will always be new deadlines, but none of them will be like this one. Anyway, it was fun, now I really have to stop. Enjoy reading.

Eelco Visser

Amsterdam
August 1997

Contents

1 Introduction

Language prototyping is the activity of designing and testing definitions of new or existing computer languages. An important aspect of a language definition is the definition of its syntax. The subject of this thesis are new formalisms and techniques that support the development and prototyping of syntax definitions. There are four main subjects: (1) Techniques for parsing and disambiguation of context-free languages. (2) Design and implementation of a new syntax definition formalism. (3) Design of a multi-level algebraic specification formalism. (4) Study of polymorphic syntax definition. This chapter sketches the background and motivation of this work and gives an overview of the thesis.

1.1 General

1.1.1 Language Prototyping

Computer languages are used to instruct computers or to encode data processed by computers. According to their application area languages are classified as programming language, domain specific language, specification language or data format.

Language design is a recurrent activity in computer science and software engineering. Kinnersley (1995) lists about 2350 languages that have been designed since computers were first developed in the 1940s. Bearing in mind that this list contains only a fracion of all languages it is probably an underestimation to say that a new language appears every week. New general purpose languages are designed as new technology becomes available that poses new requirements or provides new opportunities. A recent example is the Java programming language that addresses problems posed by exchanging programs over networks. New domain specific languages are developed to encapsulate domain knowledge previously expressed in a general purpose language.

The design of a new computer language requires a considerable investment. By developing a prototype of the new language that contains its essential features at an early stage, its design can be tested and adjusted if necessary. Design tools can considerably speed up the design process by generating components of a prototype.

1.1.2 Language Definitions

The core of the design of a language is a language definition consisting of the description of its syntax and semantics. The syntax describes the form and structure of its sentences. The semantics describes the meaning of the syntactic constructs, which can vary from the interpretation of expressions as computer programs to the translation of expressions to another language. A *formal* language definition is a definition of syntax and semantics in a formal language specification formalism, consisting of a syntax definition formalism and a logical or computational formalism for the expressing its semantics.

Language prototyping involves testing a language definition by executing it as a computer program. Execution of a language definition comprises the syntactic analysis of expressions in the language according to the syntax definition of the language and the computation of the semantics of syntactically correct expressions. A specification formalism is called executable if language definitions can be tested directly on a computer or if tools exist that construct executable computer programs from language definitions. A set of tools that supports the development of programs or specifications in some language could be called a programming environment. A programming environment for developing languages and their programming environments is called a meta-environment. Other requirements for language definition formalisms besides executability are that they support the description of existing languages, that language definitions are extensible and can be combined with other language definitions, and that the formalism is not overly restrictive.

1.1.3 Syntax Definition Formalisms

A syntax definition formalism is a formal language for the specification of the syntactic rules of a language. The syntax describes the sentences of the languages and assigns a structure to these sentences. An example of a syntax definition formalism is the context-free grammar formalism introduced by Chomsky (1956). A context-free production $A_1 \ldots A_n \to A_0$ determines that a sentence of type A_0 can be composed by concatenating sentences of type A_1 to A_n in that order. It can also be considered as a composition rule for trees: a tree of type A_0 can be composed by creating a new tree node labeled A_0 with trees of type A_1 to A_n as direct descendants.

In general, a syntax definition formalism can be characterized as follows. A syntax definition (also called grammar) is a set of rules that describe how to generate a set of trees. The concatenation of all leafs nodes of a tree (the *yield*) gives a sentence. The language defined by a grammar is the set of yields of the trees it generates. A parser is a function that assigns to a sentence a tree that represents its structure. If more than one tree has a sentence as its yield, the sentence is ambiguous. To solve ambiguities the syntax definition formalism may provide a disambiguation method that allows the formulation of disambiguation rules for selecting the intended tree for a sentence.

This abstract approach to syntax is also applicable to the type systems of programming languages. A signature describes the valid typed expressions (the trees), untyped expressions (the sentences) can be derived from the typed expressions, a type checker analyses untyped expresssions and assigns them a type. In particular, it is applicable to algebraic signatures.

1.1.4 Algebraic Specification of Languages

An algebra is a set of data with operations on those data. In case of many-sorted algebras the data can be divided over a collection of sets. An algebraic specification is the description of an algebra. It consists of a signature declaring the types of the algebraic operations and a logical formula describing properties of these operations. Several algebraic specification formalisms have been developed that use conditional equational logic as defining logic. Equations can be executed by interpreting them as rewrite rules.

Rus (1972) and later also Goguen *et al.* (1977) showed that context-free grammars correspond to algebraic signatures (see also Hatcher and Rus (1976)). The composition of a tree, i.e., the construction of a new tree from subtrees, corresponds to the application of a function. In this view, languages correspond to algebras. A number of algebraic specification formalisms (for instance, OBJ, Pluss, ASF+SDF, Elan) exploit this property by using signatures with mixfix operators or even arbitrary context-free grammars instead of a prefix signature. A definition can be viewed as a context-free grammar or as an algebraic signature. The grammar view is used to generate parsers from a definition. The signature view describes the abstract syntax trees that are used by semantic tools. A mapping from parse trees to abstract syntax trees is used as interface between parser and semantic tool. In Heering *et al.* (1989) these views are made explicit by translating an SDF definition to a contex-free grammar (BNF) and to a first-order algebraic signature and by providing a translation from parse trees to abstract syntax trees.

1.1.5 ASF+SDF

ASF+SDF is an algebraic specification formalism designed for the specification and prototyping of programming language tools. It uses the syntax definition formalism SDF for the definition of the syntax of a language. This enables an expressive notation in specifications, since functions can be prefix, postfix, infix and mix-fix. Furthermore, the syntax of the programming language under consideration is also expressed in these signatures. The semantics of a language can be defined by means of operations on the language specified by means of conditional equations.

Language prototyping is supported by the ASF+SDF Meta-Environment (Klint, 1993, Van Deursen *et al.*, 1996). It is an interactive development environment for modular ASF+SDF specifications. Given a specification of a language, a programming environment for that language is generated automatically. The use of incremental generation techniques makes that changes to the specification are immediately applied to the generated environment. This makes it possible to experiment with alternative designs.

Although the formalism in combination with the Meta-Environment provides a powerful system for language prototyping, there are several shortcomings. For instance, it is not possible to generate stand-alone environments and the evaluation of equations by means of term rewriting is interpreted instead of compiled. Currently much research is invested in overcoming these shortcomings.

1.1.6 Syntax Definition for Language Prototyping

This thesis is concerned with the design and implementation of methods to enhance the expressive power and usability of the syntactic aspects of language definition formalisms. The main theme is the development of techniques for providing an *expressive* syntax definition formalism. The point of departure is the syntax definition formalism SDF of Heering *et al.* (1989) that is used in combination with the algebraic specification formalism ASF of Bergstra *et al.* (1989b). This setting provides the direct background and motivation for this work, but the techniques developed are applicable in other syntax definition settings as well. There are four main results:

— Scannerless generalized-LR parsing is a new approach to parsing without scanners that solves a number of problems of conventional parsing techniques, by combining the following techniques: parsing without scanner, generalized-LR parsing, static disambiguation with priority and associativity declarations, lexical disambiguation with follow restrictions and reject productions.

— SDF2 is an expressive syntax definition formalism for context-free syntax defintion. It is a redesign of SDF as a family of orthogonally defined features for syntax definition.

— The multi-level algebraic specification formalism MLS extends first-order many-sorted algebraic specification by making the sorts used in a signature a user-definable algebraic data type. This provides a simple and uniform framework for the specification of advanced type constructs including polymorphism and higher-order functions.

— Polymorphic syntax definition is the combination of the flexible notation facilities of SDF with the flexible typing facilities of MLS.

Each of these subjects brings its own technical problems that are addressed in this thesis. In the rest of this chapter we give an overview of this development and indicate the connections.

1.2 Part I: Context-free Parsing Techniques

Part I describes techniques for parsing and disambiguation of context-free languages.

1.2.1 Scannerless Generalized-LR Parsing

Current deterministic parsing techniques have a number of problems. These include the limitations of parser generators for deterministic languages and the complex interface between scanner and parser. Scannerless parsing is a parsing technique in which lexical and context-free syntax are integrated into one grammar and are all handled by a single context-free analysis phase. This approach has a number of advantages including discarding of the scanner and lexical disambiguation by means of the context in which a lexical token occurs. Scannerless parsing generates a number of interesting problems as well. Integrated grammars do not fit the requirements of the conventional deterministic parsing techniques. A plain context-free grammar formalism leads to unwieldy grammars, if all lexical information is included. Lexical disambiguation needs to be reformulated for use in context-free parsing.

The scannerless generalized-LR parsing approach presented in Chapter 3 solves these problems. Grammar normalization is used to support an expressive grammar formalism without complicating the underlying machinery. Follow restrictions are used to express longest match lexical disambiguation. Reject productions are used to express the prefer keywords rule for lexical disambiguation. An adaptation of the SLR(1) parser generation algorithm is used to implement disambiguation by general priority and associativity declarations and interprets follow restrictions. Generalized-LR parsing is used to provide dynamic lookahead and to support parsing of arbitrary context-free grammars including ambiguous ones. An adaptation of the GLR algorithm supports the interpretation of grammars with reject productions.

1.2.2 Disambiguation Filters

An ambiguous context-free grammar defines a language in which some sentences have multiple interpretations. For conciseness, ambiguous context-free grammars are frequently used to define even completely unambiguous languages and numerous disambiguation methods exist for specifying which interpretation is the intended one for each sentence. The existing methods can be divided in 'parser specific' methods that describe how some parsing technique deals with ambiguous sentences and 'logical' methods that describe the intended interpretation without reference to a specific parsing technique.

Chapter 4 proposes a framework of *filters* to describe and compare a wide range of disambiguation problems in a parser-independent way. A filter is a function that selects from a set of parse trees (the canonical representation of the interpretations of a sentence) the intended trees. A number of general properties of disambiguation filters is defined and several case studies are discussed including disambiguation by means of priorities.

1.2.3 Optimizing Parsing Schemata by Disambiguation Filters

Although disambiguation filters give an abstract account of disambiguation, implementation of disambiguation by means of a filter applied to the parse forest after parsing can be too inefficient for a number of disambiguation methods. Therefore, it would be attractive if a declaratively defined disambiguation filter could be efficiently implemented by applying it during parsing or even during parser generation.

In Chapter 5 a study into the optimization of the composition of parsing algorithms and disambiguation filters is started, by considering two filters based on priorities. The first filters a set of parse trees and selects trees without priority conflict. The second selects the trees which are lowest in the multi-set ordering on parse trees induced by the priority relation on productions.

The theory of parsing schemata of Sikkel (1993) gives an abstract account of parsing algorithms. In Chapter 5 the parsing schema for Earley's parsing algorithm is optimized by applying the two priority filters. For the priority conflict filter this results in an optimized LR(0) parser generator that yields parsers that do not produce parse trees with a priority conflict. This provides the formal derivation of the imlementation rules presented in Chapter 3. For a restricted case of the multi-set filter an optimization of Earley's algorithm is derived.

1.3 Part II: A Family of Syntax Definition Formalisms

The formalism SDF is a syntax definition formalism for specification of lexical and context-free syntax of programming languages. The design of the formalism is rather monolithic, which makes it difficult to extend with new features or experiment with the implementation. In Part II SDF is redesigned and specified as a modular and extensible family of syntax definition formalisms. Each feature is specified as an extension of a kernel formalism, orthogonal with respect to other features. The meaning of most features is expressed in terms of the primitives of the kernel formalism by means of normalization functions. One of the members of this family is SDF2, the successor of SDF.

The syntax definition formalism SDF2 is a formalism for the concise definition of context-free syntax. The semantic core of the formalism is formed by context-free grammars extended with character clases, priorities, follow restrictions and reject productions. Grammars in this basic format describe a set of parse trees to which strings are associated that form the language of the grammar. In connection with semantics specification formalisms such as ASF, a grammar is interpreted as a signature and the parse trees it generates as terms in the term algebra generated by the signature. The implementation of SDF2 consists of a grammar normalizer, a parser generator and a generic parser. It supports arbitrary context-free grammars using the GLR parsing algorithm.

One of the main contributions of SDF2 is the complete integration of lexical and context-free syntax. The formalism supports the definition of lexical and context-free syntax providing a separate name space for symbols such that interference is prevented. The grammar normalizer integrates lexical and context-free syntax into a single context-free grammar. The scannerless parser generated for such a grammar reads input characters directly and combines lexical analysis with context-free analysis in a single parsing phase.

Ambiguous grammars can be disambiguated by means of three disambiguation facilities. Priority and associativity declarations can be used to disambiguate mixfix expression grammars in a very general way. Disambiguation by means of priorities is implemented in the parser generator. For the disambiguation of lexical ambiguities two features are introduced. With follow restrictions the follow-set of grammar symbols can be restricted, which enables the expression of the 'prefer longest match' disambiguation rule. With reject productions one can express the 'prefer keywords' rule. Follow restrictions are interpreted during parser generation, reject productions are interpreted during parsing.

Other disambiguation methods can be defined as filters on parse forests (compact representations of sets of parse trees). Due to the open design of the SDF2 implementation such filters can be easily attached to the parser. A number of case studies of disambiguation filters are discussed in Chapter 4.

Other features of SDF2 are literals, an expressive set of regular expressions, and symbol aliases that serve to abbreviate complicated regular expressions. Furthermore, the formalism supports modular syntax definitions and flexible reuse of modules by means of symbol and production renamings.

Many of the features are defined in terms of the core features by means of a normalization function on syntax definitions. The formalism can be coupled to any semantics specification language based on first-order many-sorted signatures providing user-definable syntax. The modular design of the formalism supports experiments with new features.

Chapter 6 gives an introduction to SDF2 and discusses the approach of designing it as a family of syntax definition formalisms. Chapter 7 defines the kernel of the family consisting of context-free grammars with sorts, character classes and literals. The semantics of the formalism is defined by means of a well-formedness predicate on parse trees characterizing the trees generated by a grammar. Chapter 8 defines disambiguation by means of priorities, follow restrictions and reject productions. Regular expressions are defined to abbreviate several common patterns in syntax definitions such as lists, optional constructs and tuples. The integration of lexical and context-free syntax is defined. Chapter 9 introduces a renaming operator on grammars that can be used to rename sorts and productions. Renamings are then used to define symbol aliases. A module mechanism is defined that supports the modularization of syntax definitions. Modules can be parameterized with a list of symbols and renamings can be applied to imports. Chapter 10 discusses the assembly of SDF2 from the features defined in the previous chapters and discusses possible improvements.

1.4 Part III: Multi-Level Algebraic Specification

Polymorphic, higher-order functions in functional programming languages provide a powerful abstraction method to construct reusable software. The first-order signatures provided by conventional many-sorted first-order algebraic specification formalisms (such as ASF+SDF) do not support polymorphic or higher-order functions. In Part II a multi-level algebraic specification formalism is designed and specified in order to study the extension of first-order formalisms with polymorphism and higher-order functions.

The multi-level specification formalism MLS extends first-order many-sorted algebraic specification by making the algebra of types a user-definable data type. The structure of the types used in the signature of a specification is specified by means of an algebraic specification itself. This process is formalized in a multi-level setting. The terms over a signature at level $i + 1$ can be used as type expressions at level i. Variables in type expressions are interpreted as universally quantified type parameters. Function declarations with such a universally quantified type are interpreted as declaration schemata for functions with closed type expressions and thus represent polymorphic functions and constants. Functions can also be overloaded, i.e., have more than one type. The term structure is applicative, enabling higher-order functions. The formalism supports modular specifications.

The formalism MLS is defined by means of a specification in ASF+SDF. This specification also forms the basis for a prototype environment for MLS. The environment consists of a typechecker that is defined in terms of a well-formedness checker and a type assignment procedure. Type assignment is an extension of the Hindley/Milner algorithm to many-sorted types, multi-levels and overloading of functions. Furthermore, the environment contains a term rewrite interpreter for equations in specifications.

Applications of multi-level specifications include all functional programs expressible in a Hindley/Milner system. Due to the many-sortedness of the signatures of types and kinds (as opposed to the single-sorted types of functional languages) more distinction can be made in type assignment. This enables the definition of data types such as stratified stacks and tuples. By means of equations over types still more advanced typing constructs can be modeled. An example is the type of the `zip` function that maps a list of tuples to a tuple of lists. Other applications of type equations are type abbreviations, recursive types, record types, the polytypic functions of Jansson and Jeuring (1997), the type classes of Haskell and the constructor classes of Jones (1993). The specification of type assignment presented here only deals with syntactic equality and not with equality modulo type equations.

Chapter 11 gives an introduction to the formalism and discusses related work. Chapter 12 handles the case of specifications consisting of a single level. This corresponds to first-order algebraic specification. First an untyped equational specification formalism with equational logic and term rewriting is defined. This is extended with a first-order monomorphic applicative type system. In Chapter 13 the possibilities of multi-level specifications are explained by means of a number of example specifications. The formalism is defined in Chapter 14, building on the language of Chapter 12. Appendix B defines several auxiliary tools such as substitution, matching and unification.

1.5 Part IV: Polymorphic Syntax Definition

The signatures of multi-level specifications only support prefix and infix functions. Chapter 15 develops theory to combine the type flexibility of multi-level specifications with the notational flexibility of context-free grammars.

The combination of the idea of grammars as signatures with multi-level algebraic specification leads to a multi-level grammar formalism. In a multi-level grammar the set of non-terminals becomes a user-definable data type in the same way as the types in multi-level specifications. Moreover, types and object level data are specified by means of a context-free grammar instead of with a signature, leading to flexible notation.

The combination provides a formalism for polymorphic syntax definition, in which common language constructs can be described generically and reused in many specifications. It turns out that while both formalisms have a decidable type-assignment/parsing problem, the combination in its full generality has an undecidable parsing problem. However, a subset of such multi-level grammars can be characterized that have a decidable parsing problem, while not being too restrictive for use in abstract data type specification. For this class of grammars that satisfy the finite-chain property, a parsing algorithm is presented.

When restricted to two levels we have a formalism that is similar to Van Wijngaarden grammars. The difference is that VWGs use derived strings with variables (sentential forms) as types at level 0, while our two-level grammars use parse trees with variables. This restriction ensures that syntactic unification is decidable, which it is not in VWGs. The further restriction of two-level grammars to grammars that satisfy the finite-chain property results in grammars with a decidable parsing problem. Van Wijngaarden grammars were not succesful in executable definition of programming languages. The discovery that ϵ-productions could be used to encode the static and even dynamic semantics of a language led to a formalism with a very difficult parsing problem. This sparked developments in the usage of VWGs as a programming language instead of a grammar formalism. In Chapter 15 it is shown that this development has hidden the very useful application of two-level grammars to polymorphic syntax definition, opening the flexibility of polymorphism to grammar development.

1.6 Short Trips

The main theme of this thesis is that of improving formalisms and techniques for syntax definition in support of language prototyping. The organization of this thesis reflects the development from context-free syntax definition, through multi-level specifications to polymorphic syntax definition. For the reader interested in subsets of these subjects we suggest some alternative itineraries through this thesis.

Three Formalisms For a quick overview of the formalisms developed in this thesis look at the following sections with examples: §6.2 discusses the main features of SDF2 illustrated by means of an example. The main features of MLS are discussed in §11.2. A large number of examples of multi-level specifications are presented in §13.1. §15.4 presents several examples of two-level grammars used for data type specification.

Priorities The definition and implementation of priorities is presented in a couple of sections. §3.4 gives a short overview of disambiguation with priorities. A more precise definition of the interpretation of priorities as disambiguation filters is presented in §4.5. The derivation of the efficient implementation of priorities during parser generation is presented in Chapter 5. Rules for the implementation of a parser generator are presented in §3.5.

Parsing Parsing of context-free grammars extended with character-classes, priorities, follow restrictions and reject productions is discussed in Chapter 3. Parser generation in §3.5 and the adaptation of the GLR algorithm in §3.8. A parsing algorithm for two-level grammars is presented in §15.6.

Disambiguation For disambiguation by means of priorities see above. Other disambiguation methods for context-free grammars are discussed in Chapter 4. Parse forests are the subject of §3.8 and §7.5.5. Disambiguation is also an issue in type assignment for signatures with overloaded functions. In §14.4.2 the type assignment for multi-level specifications with overloading is presented.

ATerms The annotated term format (ATerms) of Van den Brand *et al.* (1997a) is a generic, untyped format for the representation of structured data. ATerms are used in Chapter 7 to represent parse trees and parse forests. The definition of the subset of ATerms that represent well-formed parse forests is given in §7.5.5. ATerms and their role in data representation are also discussed in §15.2.3.

Terms & Term Rewriting In Part II terms also play an important role. In Chapter B a number of standard operations such as substitution, matching and unification on terms are defined. In §12.1.4 these tools are used to specify a term rewrite interpreter of equations.

Type Systems The chapters in Part III present the specification of an advanced type system. Grammars can also be considered as signatures. The view of grammars as signatures is discussed in Chapter 7 and Chapter 15.

Coupling of Syntax to Semantics In §9.3 an example is given of the coupling of user-definable syntax to conditional equations. In Chapter 12 and Chapter 14 the typing of equations given a signature is discussed.

Language Design In Part II the specification of SDF2 is presented. Several considerations in the design of the formalism are discussed in §6.3. In Chapter 12 the specification of a simple applicative equational specification formalism is presented. The specification includes a discussion of equational logic, term rewriting, well-formedness under a signature and type assignment.

Literate Specification Part II presents the specification of the syntax definition formalism SDF2. Part III presents the specification of a multi-level algebraic specification formalism. These specifications were written as literate specifications. The typeset specifications with comments were prepared using the ASF+SDF Meta-Environment.

1.7 Origins of the Chapters

Most of the chapters in this thesis were published before as separate papers. We list their origin.

— Chapter 3 on the *scannerless generalized-LR parsing* approach is a new paper that gives an overview of the design and implementation of the SDF2 normalizer, parser generator and parser. It appeared as a technical report (Visser, 1997f).

— Chapter 4 on *disambiguation filters* is joint work with Paul Klint. It presented under the title *using filters for the disambiguation of context-free grammars* at the *ASMICS Workshop on Parsing Theory* in Milan and appeared in the proceedings (Klint and Visser, 1994).

— Chapter 5 was presented at the *Accolade{95* conference on logic in Amsterdam and appeared in the proceedings (Visser, 1995a). It has been accepted for presentation at the *International Workshop on Parsing Technology (IWPT'97)* in Boston and for publication in the proceedings (Visser, 1997a).

— Part II on the specification of SDF2 as a *family of syntax definition formalisms* is an update and extension of a paper that was presented at the *ASF+SDF'95* workshop on *Generating Tools from Algebraic Specifications* and appeared in the proceedings (Visser, 1995b). In its current form it appeared as a technical report (Visser, 1997d).

— Part III appeared as a single chapter in the book *Language Prototyping. An Algebraic Specification Approach* (Visser, 1996a). The version in this thesis has been split up in five chapters and several example specifications have been added. Furthermore, the specification has been improved in a few places.

— An extended abstract of Chapter 15 was presented at the AMAST workshop on Algebraic Methods in Language Processing (AMiLP'95) in Enschede and was published in the proceedings (Visser, 1995c). The current version is accepted for publication in a special issue of *Theoretical Computer Science* dedicated to the workshop (Visser, 1997e).

2

Specification in ASF+SDF

ASF+SDF is a first-order many-sorted algebraic specification formalism designed for the specification of computer languages. Development of ASF+SDF specifications is supported by the ASF+SDF Meta-Environment that generates a programming environment from the specification of a language. This chapter gives an introduction to the basic notions of ASF+SDF.

2.1 Introduction

Algebraic specification is concerned with the formal description of abstract data types. An algebraic specification consists of a signature describing the structure of the data type—sorts of data and operations on the data—and expressions in some form of logic that define the meaning of the operations. An algebraic specification formalism is characterized by the form of the signatures, the logic and additional features supporting specification development.

ASF+SDF is a modular algebraic specification formalism with first-order, many-sorted signatures and conditional equations. It is the union of the algebraic specification formalism ASF and the syntax definition formalism SDF. An important feature of the formalism is the use of context-free grammars as signatures (the contribution of SDF), providing an expressive notation for functions. This feature makes the formalism well suited for programming language specification. The signature describes the concrete syntax of a language and functions can be defined directly on the concrete constructs.

Development of specifications is supported by means of the interactive, incremental ASF+SDF Meta-Environmnent. It is called a *meta*-environment because it supports the generation of programming environments from specifications. Tool support for literate specification encourages the documentation of specifications.

In this chapter we give an introduction to ASF+SDF. In §2.2 and §2.3 we discuss algebraic structure, signatures and the use of grammars as signatures. In §2.4 and §2.5 we discuss equational logic and its operationalization by means of term rewriting. In §2.7 we discuss the ASF+SDF Meta-Environment. In §2.8 we discuss the documentation of specifications as literate specifications. In §2.9 we discuss some concepts related to the specification of programming languages. We conclude in §2.10 with references to the literature on ASF+SDF and algebraic specification.

2.2 Many-Sorted Algebra

An algebra is a collection of values or data A with constants $a \in A$ and operations $f : A^n \to A$ working on the values. A many-sorted algebra is a family $(A(s) \mid s \in S)$ of such collections of data indexed by a set of sorts S with a collection of constants $c \in A(s)$ and a collection of functions $f : A(s_1) \times \cdots \times A(s_n) \to A(s_0)$.

The structure of an algebra can be described by means of a signature. A signature consists of the declaration of the sorts (names for the collections) and the declaration of the constants and functions of an algebra. (In other contexts signatures are also called interfaces.) For example, the following signature describes an algebra consisting of two sorts `Bool` and `Nat` and several operations on those sorts.

```
sorts Bool Nat
functions
  true  : Bool
  false : Bool
  not   : Bool        -> Bool
  and   : Bool # Bool -> Bool
  zero  : Nat
  succ  : Nat         -> Nat
  plus  : Nat # Nat   -> Nat
  geq   : Nat # Nat   -> Bool
```

A declaration $c : s$ declares a constant c of sort s. A declaration $f : s_1 \# \ldots \# s_n \to s_0$ declares a function with n arguments of sorts $s_1 \ldots s_n$ and result sort s_0, i.e., a function $f : A(s_1) \times \cdots \times A(s_n) \to A(s_0)$.

There are many algebras corresponding to a signature. Each structure that has the prescribed sets of data and operations on them is considered an algebra in the family of algebras described by a signature. A special algebraic structure in this family is the term algebra $\mathrm{T}(\Sigma)$ corresponding to a signature Σ. The data of this algebra are terms over the signature, which are constructed according to the following rules.

(Constants) If $c : s$ is a constant declaration, i.e., an element of the function declarations $\mathrm{F}(\Sigma)$ of Σ, then c is a term of sort s.

$$\frac{c : s \in \mathrm{F}(\Sigma)}{c \in \mathrm{T}(\Sigma)(s)}$$

(Functions) If $f : s_1 \# \ldots \# s_n \to s_0$ is a function declaration and t_i for $1 \leq i \leq n$ are terms of sort s_i, then $f(t_1, \ldots, t_n)$ is a term of sort s_0.

$$\frac{f : s_1 \# \ldots \# s_n \to s_0,\ t_i \in \mathrm{T}(\Sigma)(s_i)}{f(t_1, \ldots, t_n) \in \mathrm{T}(\Sigma)(s_0)}$$

For example, according to these rules the expression

```
not(geq(zero(), plus(succ(zero()), zero())))
```

is a term of sort Bool in the signature above.

The term algebra over a signature is special because it is initial in the family of algebras for the signature. For each algebra A in $\mathrm{Alg}(\Sigma)$ there is a unique translation from the term algebra $\mathrm{T}(\Sigma)$ to A. This is the interpretation function I_A defined as follows:

$$
\begin{aligned}
I_A(c) &= c_A \\
I_A(f(t_1, \ldots, t_n)) &= f_A(I_A(t_1), \ldots, I_A(t_n))
\end{aligned}
$$

Here c_A is the value of the constant c in A and f_A is the function corresponding to f in A. The fact that the term algebra is initial has another consequence, it means that it has 'no junk', i.e., all values in the algebra correspond to a term.

2.3 Grammars as Signatures

2.3.1 Context-free Syntax

In ASF+SDF signatures are specified by means of a context-free grammar in the form of a syntax definition in SDF. Basically, an SDF definition consists of a declaration of sorts listing all the sorts and a context-free syntax section declaring the functions. The differences with signatures is that the names of functions are not restricted to prefix functions, but patterns that can be written before, in between and after the arguments of the function. A function declaration is of the form $l_0\ s_1\ l_1 \ldots l_n\ s_n\ l_{n+1} \to s_0$, where the l_i are lists of zero or more literals and the s_i are sorts. A literal is a string of characters between double quotes that indicates a part of the function name that is to be written *literally*.

Corresponding to the signature of Boolean and natural number values in the previous example we can define a grammar of Boolean and natural number values:

```
sorts Bool Nat
context-free syntax
  "true"             -> Bool
  "false"            -> Bool
  "not" Bool         -> Bool
   Bool "and" Bool   -> Bool
   Bool "or"  Bool   -> Bool
  "0"                -> Nat
  "s" "(" Nat ")"    -> Nat
  Nat "+"  Nat       -> Nat
  Nat "*"  Nat       -> Nat
  Nat ">=" Nat       -> Bool
```

2.3.2 Parse Trees as Terms

Again, we can formulate rules for the construction of terms over such syntax definitions. Let $\mathcal{G}$ be a context-free grammar, $\mathrm{S}(\mathcal{G})$ the set of sorts of grammar $\mathcal{G}$, and $\mathrm{P}(\mathcal{G})$ the set of productions of $\mathcal{G}$.

(Constants) If $l \rightarrow s$ is a constant context-free syntax rule (without sorts on the left-hand side), then l is a term of sort s.

$$\frac{l \rightarrow s \in \mathrm{P}(\mathcal{G})}{l \in \mathrm{T}(\mathcal{G})(s)}$$

(Functions). If $l_0\ s_1\ l_1 \ldots l_n\ s_n\ l_{n+1} \rightarrow s_0$ is a context-free syntax rule and t_i for $1 \leq i \leq n$ are terms of sort s_i, then $l_0\ t_1\ l_1 \ldots l_n\ t_n\ l_{n+1}$ is a term of sort s_0.

$$\frac{l_0\ s_1\ l_1 \ldots l_n\ s_n\ l_{n+1} \rightarrow s_0 \in \mathrm{P}(\mathcal{G}), t_i \in \mathrm{T}(\mathcal{G})(s_i)}{l_0\ t_1\ l_1 \ldots l_n\ t_n\ l_{n+1} \in \mathrm{T}(\mathcal{G})(s_0)}$$

For example, the expression

```
not 0 >= s(0) + 0
```

is a term of sort Bool in the syntax definition above.

2.3.3 Priorities

One problem with this definition is that terms can be *ambiguous*. For instance, the term `0 + 0 + 0` is a term of sort Nat in two ways, depending on whether we take `+` left or right associative. In SDF this problem is solved by declaring the associativity and priority of functions. The attributes {`left`}, {`right`} and {`assoc`} declare a function to be left-associative, right-associative or simply associative, respectively. By means of priorities one can express that a function has higher priority than another function. For example, the operations for the natural numbers in the grammar above can be disambiguated in the following manner:

```
context-free syntax
  Nat "+" Nat -> Nat {left}
  Nat "*" Nat -> Nat {left}
  "(" Nat ")" -> Nat {bracket}
priorities
  Nat "*" Nat -> Nat >
  Nat "+" Nat -> Nat
```

This gives multiplication higher priority than addition and declares both operators as left associative.

In order to override the declared priorities one can introduce bracket functions, function declarations of the form $l_0\ s\ l_1 \rightarrow s$ {`bracket`}. Such functions are identity functions, i.e., we have the implicit identity $l_0\ t\ l_1 = t$. For instance, in the example above, parentheses are declared as brackets for natural numbers, such that

```
s(s(0)) * (s(s(0)) + s(0))
```

can be written to override the priority of `*` over `+`.

2.3.4 Lists

SDF has special support for functions that have an arbitrary number of arguments of the same sort. For example, the declaration

```
context-free syntax
  program FunDef* -> Program
```

declares that the function `program` constructs a `Program` from zero or more function definitions.

2.3.5 Injections

ASF+SDF permits "syntax-less" chain rules for injecting a sort into another sort. For example, to define that a natural number is an integer we declare

```
context-free syntax
  Nat -> Int
```

2.3.6 Lexical Syntax

Syntax definitions do not only define the syntax of sentences, the context-free syntax, but also define the syntax of words, the lexical syntax. We have already seen literals that describe exactly one word. The literal `"true"` indicates that the word consisting of the letters `t`, `r`, `u` and `e` is a unit of the language. Definitions can also contain a lexical syntax section that describes collections of words. For example, the definition

```
sorts Id
lexical syntax
  [a-z][a-zA-Z0-9]* -> Id
```

defines that identifiers (sort Id) are words starting with a lowercase letter followed by zero or more letters or digits.

2.3.7 Layout

The words in sentences can be separated by whitespace and comments. What exactly constitutes this so called *layout* has to be defined in the lexical syntax section. A conventional definition is the following

```
lexical syntax
  [ \t\n]      -> LAYOUT
  "%%" ~[\n]* -> LAYOUT
```

It defines spaces, tabs and newlines as whitespace and furthermore defines everything on a line after two percent signs as comments.

2.4 Conditional Equations

Now we have seen how algebraic structure of data can be specified by means of signatures and grammars, but we have said nothing about their *meaning.* In ASF+SDF conditional equations are used to describe the meaning of functions. Equations define functions by specifying the equality of terms. An equation of the form $t = s$ declares the terms t and s to be equal. For example, the following equations define addition and ordering of natural numbers.

```
equations
  [0] 0 + n     = n
  [1] s(n) + n' = s(n + n')

  [2] 0 > n        = false
  [3] s(n) > 0     = true
  [4] s(n) > s(n') = n > n'
```

The tags $[x]$ before the equations are for documentation purposes only.

A conditional equation has the form:

$$\frac{t_1 = s_1, \ldots, t_n = s_n}{t = s} \quad \text{or} \quad t = s \textbf{ when } t_1 = s_1, \ldots, t_n = s_n$$

It declares that term t is equal to term s, if t_i equals s_i for $1 \leq i \leq n$.

2.5 Term Rewriting

For sets of equations respecting certain criteria, equality of terms can be decided by interpreting the equations as term rewrite rules to simplify a term to normal form. Term rewriting is the process of repeatedly replacing a subterm that matches the left-hand side of an equation by the correspondingly instantiated right-hand side of the equation until no such subterm can be found. The resulting term is in normal form.

2.6 Modularization

A specification can be divided in a number of modules. A module can be reused by importing it in other modules. The signature of a module is divided in an export and an import part. Items in the exported signature are visible in modules that imported the module. Hidden items are for use in the module itself and are invisible to other modules. Currently, ASF+SDF does not support renamings of sorts or functions or parameterized modules. This entails that instantiations of a data type have to be created by copying a module.

2.7 The ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment is a programming environment for the development of ASF+SDF specifications. The environment provides editors for editing and type checking modules. Specifications can be tested by opening a term editor over some module. Such a term editor provides a basic programming environment for the language being defined. The basic operations on terms in term editors are checking of the syntax and reducing a syntactically correct term to its normal form according to the equations in the specification.

2.8 Literate Specification

Specifications can be used to prototype languages and programming environments, but are meant in the first place as language descriptions for implementors and users of the language. Therefore, it is important that specifications are documented properly. Maintaining the documentation of a specification separately from the specification itself is a guarantee for a divergence between the two. Therefore, documentation of ASF+SDF specifications is supported by means of literate specification. The specification developed and tested in the Meta-Environment is also the basis for the documentation. The documentation is generated automatically from the specification by means of a literate specification tool for ASF+SDF. The tool—appropriately called ToLaTeX—translates an ASF+SDF specification to LaTeX code for typesetting. Comments in the specification are passed directly to LaTeX. The syntax declarations and equations of a specification are typeset. To accommodate the use of mathematical symbols, ToLaTeX is parameterized with a user extendable mapping from ASCII symbols to LaTeX commands. For instance `>=` can be mapped to `\geq`, which comes out as $\geq$. All specifications in this book have been prepared in this manner.

2.9 Specification of Programming Languages

We discuss some terminology and conventions in the usage of ASF+SDF for the specification of programming languages. In ASF+SDF a programming language is considered a data type. The specification of a programming language consists of a specification of the *syntax*, the strings that are sentences of the language, a description of the *semantics*, the meaning of sentences of the language, and a description of the *context-sensitive requirements* on sentences.

2.9.1 Language Processors

We distinguish several basic tools used in manipulating programs in a programming language.

— A parser determines whether a string is a syntactically correct sentence of the language.

— A typechecker checks context-sensitive requirements, mainly with respect to type declarations.

— An interpreter computes the value for a program and its input.

— A compiler translates a program to a program in another language.

— A program transformation tool transforms a program in another program in the same language.

Parsers are generated automatically from the syntax definition of the language. Type checkers, interpreters and compilers are defined by means of functions in the specification.

2.9.2 Programming Environments

A programming environment is a collection of tools supporting the development of programs in a programming language. In addition to parsers, type checkers, interpreters and compilers such environments may contain the following tools:

— A (structure) editor is used to create and maintain the program text and invoke other tools. In the ASF+SDF Meta-Environment structure editors are generated from the syntax definition of a language.

— A debugger helps finding errors in programs.

— A pretty-printer transforms the text of a program such that it becomes more readable.

— A typesetter typesets a program for printing and usage in program documentation.

— Semantics analysis tools help in understanding a program and finding errors in them.

— A version manager supports maintenance of versions and configurations.

2.9.3 Structure of a Specification

ASF+SDF does not enforce a modularization style for specifications of programming languages. To promote reuse and extensibility, we use the following style. The specification of a programming language is defined in several modules that can be classified as:

— Syntax modules describe the syntax of the language. If it concerns a larger language we divide the syntax in several modules containing parts of the language that belong together. Some languages can be divided in a kernel and several extensions to it.

— Data type modules describe auxiliary data structures.

— Interpreter modules describe the evaluation function for language constructs.

For each other kind of processor, a separate module is introduced.

An *extension* of a language consists of a syntax module for the syntax of the new constructs, a data type module describing additional data structures and an interpreter module describing the extension of the evaluation function.

2.10 Literature

The algebraic specification formalism ASF and algebraic specification with ASF and ASF+SDF are introduced in Bergstra *et al.* (1989a). The syntax definition formalism SDF is defined in Heering *et al.* (1989). The principles of the implementation of the ASF+SDF Meta-Environment are described by Klint (1993). Directions for the usages of the ASF+SDF Meta-Environment can be found in Klint (1995). Literate ASF+SDF specification is described in Visser (1994a, 1994b). Case studies and experience with ASF+SDF are described in various publications. In a recent book Van Deursen *et al.* (1996) give an introduction to ASF+SDF as language prototyping formalism using a number of advanced case studies, mainly in type checking, and present several research issues. Recent advances in research in this area are reported on in the ASF+SDF workshops (Van den Brand *et al.*, 1995, Sellink, 1997).

Background on universal algebra can be found in Meinke and Tucker (1992) and Wechler (1992). Literature on algebraic data type specification includes Loeckx *et al.* (1996).

Part I

Context-Free Parsing Techniques

3

Scannerless Generalized-LR Parsing

Current deterministic parsing techniques have a number of problems. These include the limitations of parser generators for deterministic languages and the complex interface between scanner and parser. Scannerless parsing is a parsing technique in which lexical and context-free syntax are integrated into one grammar and are all handled by a single context-free analysis phase. This approach has a number of advantages including discarding of the scanner and lexical disambiguation by means of the context in which a lexical token occurs. Scannerless parsing generates a number of interesting problems as well. Integrated grammars do not fit the requirements of the conventional deterministic parsing techniques. A plain context-free grammar formalism leads to unwieldy grammars, if all lexical information is included. Lexical disambiguation needs to be reformulated for use in context-free parsing.

The scannerless generalized-LR parsing approach presented in this chapter solves these problems. Grammar normalization is used to support an expressive grammar formalism without complicating the underlying machinery. Follow restrictions are used to express longest match lexical disambiguation. Reject productions are used to express the prefer keywords rule for lexical disambiguation. The SLR(1) parser generation algorithm is adapted to implement disambiguation by general priority and associativity declarations and to interpret follow restrictions. Generalized-LR parsing is used to provide dynamic lookahead and to support parsing of arbitrary context-free grammars including ambiguous ones. An adaptation of the GLR algorithm supports the interpretation of grammars with reject productions.

3.1 Introduction

Parsing is one of the areas of computer science where program generation is a routine technique that is successfully applied to generate parsers for programming languages given their formal definition by means of a context-free grammar. At least, in theory. In practice, most parser generators accept only a limited subset of the context-free grammars such as LL(1) or LALR(1) grammars. Since most natural grammars for languages do not respect these limitations, the language designer or compiler writer has to bend over backwards to fit the grammar into the restrictions posed by the grammar formalism by rewriting grammar rules, introducing ad-hoc solutions for parse table conflicts or resorting to side effects in the parser. Even if one succeeds in producing a grammar that respects the restrictions, a small extension or modification of the language might jeopardize the careful balance of tricks, which makes maintenance of tools for the language troublesome.

Another source of problems in generated parsers is the division between the lexical analysis phase and the context-free analysis phase and the corresponding division of the grammar into a regular grammar defining the lexical syntax and a context-free grammar defining the context-free syntax. A scanner divides the character string into tokens according to the lexical syntax. A parser structures the token string into a tree according to the context-free syntax.

At the interface between scanner and parser the lexical tokens are passed from the scanner to the parser. In the most straightforward scenario the scanner produces a stream of tokens without intervention from the parser. This entails that no knowledge of the parsing context is available in the scanner and thus no lexical analysis decisions can rely on such information. It is difficult to unambiguously define the lexical syntax of a language by means of only regular grammars. Therefore, lexical analysis and the interface with context-free analysis are usually extended. First lexical disambiguation heuristics such as 'prefer longest match' and 'prefer keyword' are applied to reduce the number of readings. If there remain ambiguities after application of these rules the scanner might produce multiple streams of tokens representing all possible partitionings of the string into tokens according to the regular grammar. The parser should then be able to cope with this non-linear input. It is also possible to supply feedback from the parser to the scanner to reduce the number of applicable grammar rules. For instance, by specifying the lexical categories that are expected for the next token.

In all such schemes lexical analysis becomes more complicated than the simple finite automaton model that motivated the use of regular grammars. Context-free parsing functionality starts to appear both inside the scanner and at the interface between scanner and parser and often operational elements corrupt the declarativity of the language definition. As a consequence, many grammars are ambiguous if only the pure regular and context-free grammar are considered as such and reasoning about the language being defined becomes difficult.

3.1.1 Scannerless Generalized-LR Parsing

In this chapter we describe an approach to syntax definition and parser generation that overcomes many of these problems. The approach is based on the integration and improvement of scannerless parsing, generalized-LR parsing and grammar normalization. Because of the integration of the former two, the approach is called *scannerless generalized-LR parsing*.

Scannerless Parsing Scannerless parsing is a parsing technique that does not use a scanner to divide a string into lexical tokens. Instead lexical analysis is integrated in the context-free analysis of the entire string. It comes up naturally when considering grammars that completely describe the syntax of a language. The term scannerless parsing was coined by Salomon and Cormack (1989, 1995). They use 'complete character level grammars' describing the entire syntax of a language down to the character level. Since conventional LR parser generation yields tables with too many conflicts, they use an extension of SLR(1) parser generation called non-canonical SLR(1). However, even this extension makes it hard to define a grammar without conflicts.

Generalized-LR Parsing The conventional LR parsing techniques and especially scannerless LR parsing suffer from conflicts in the parse table. There are two causes for conflicts in LR parse tables: ambiguities and lack of lookahead. If a conflict is caused by an ambiguity, any of the possible actions will lead to a successful parse. If it was caused by a lookahead problem, one of the actions will lead to success and the others will fail. Which action will be successful cannot be decided statically. Since ambiguity of a context-free grammar is undecidable (Floyd, 1962), it is also undecidable whether a conflict is due to an ambiguity or to a lack of lookahead. Because complete character level grammars frequently need arbitrary length lookahead, methods to solve conflicts in the table will not always succeed.

Generalized-LR parsing is an extension of LR parsing that interprets the conflicts in the parse table by forking off a parser from the main parser for each possible action in case of a conflict. If such a conflict turns out to lead to an ambiguity the parser constructs a parse forest, a compact representation of all possible parse trees for a sentence. But if the conflict was caused by lack of lookahead, the forked parsers for the wrong track will fail. In this manner lookahead is handled dynamically. Therefore, generalized-LR parsing is an ideal technique to solve the lookahead problems of scannerless parsing. Generalized-LR parsing was introduced by Tomita (1985) building on the theoretical framework of Lang (1974). It was improved by Rekers (1992) to handle all context-free grammars. In this chapter we extend Rekers' version of the algorithm with reject reductions, a facility needed for lexical disambiguation.

Grammar Normalization An aspect of the division between lexical and context-free syntax that affects the specification of syntax is the definition of layout, i.e., the whitespace and comments that can occur at arbitrary places between tokens. In the conventional setting layout is analyzed by the scanner and then thrown away. The parser never sees the layout tokens. Therefore, layout can also be ignored in the specification of context-free syntax. However, in a complete character level grammar all aspects of the syntax are completely defined, including the syntax and positions of layout. This can lead to rather unwieldy grammars that declare the occurrence of layout as separator between all grammar symbols in context-free productions.

Grammar normalization is a technique used to define an expressive grammar formalism in terms of simple context-free grammars. An example of a normalization procedure is the addition of layout symbols between the symbols in context-free productions. Other examples are the definition of regular expressions by means of productions and the flattening of modular grammars. An important aspect of the scannerless generalized-LR approach is the use of grammar normalization to keep grammars small and usable. The syntax definition formalism SDF2 used in the approach is a formalism for concise definition of complete character level grammars. SDF2 is a generalization of the syntax definition formalism SDF of Heering *et al.* (1989). The formalism and normalization procedure is defined in Part II.

3.1.2 Architecture

The typical architecture of an application of SDF2 is depicted in Figure 3.1. A program text[1] processor that transforms text into text is composed of (1) a parser front-end that analyzes the input text and produces a structured representation of the text in the form of a parse tree, (2) the actual processor that performs a transformation from a parse tree to another one and (3) a pretty-printer back-end that produces text corresponding to a transformed parse tree. Processors can be, for instance, interpreters, compilers, data flow analyzers or program transformation tools.

The input language of a processor is specified in the syntax definition formalism SDF2. Given a language definition in SDF2 and a tree to tree processor, the corresponding text to text processor is constructed using a grammar normalizer, a parser generator, a parser and a pretty-printer generator.

Grammar Normalizer A language definition in SDF2 is normalized to a plain context-free grammar extended with character classes, priority rules, follow restrictions and reject productions. Normalization is briefly discussed in §3.3. A full definition of SDF2 normalization can be found in Part II.

Parser Generator From a normalized syntax definition a parse table is generated using an extension of the standard SLR(1) algorithm with character classes, priorities, follow restrictions, and reject productions. The parser generator accepts arbitrary context-free grammars. The techniques used in the parser generator are discussed in §3.5.

Parser A parse table is interpreted by a generic, language independent SGLR parser, which reads a text and produces a parse tree. At the heart of the parser is an extension of the GLR algorithm of Rekers (1992) that reads characters directly, without using a scanner. The extension of the GLR algorithm with reject reductions is discussed in §3.8.

Pretty-Printer A pretty-printer is used to translate the output tree of the processor to text. The pretty printer itself can also be generated from the definition of the output language. This is described in Van den Brand and Visser (1996) and is not further discussed here.

[1]Here text denotes a linear representation of a program in some character code, e.g., ASCII or UniCode.

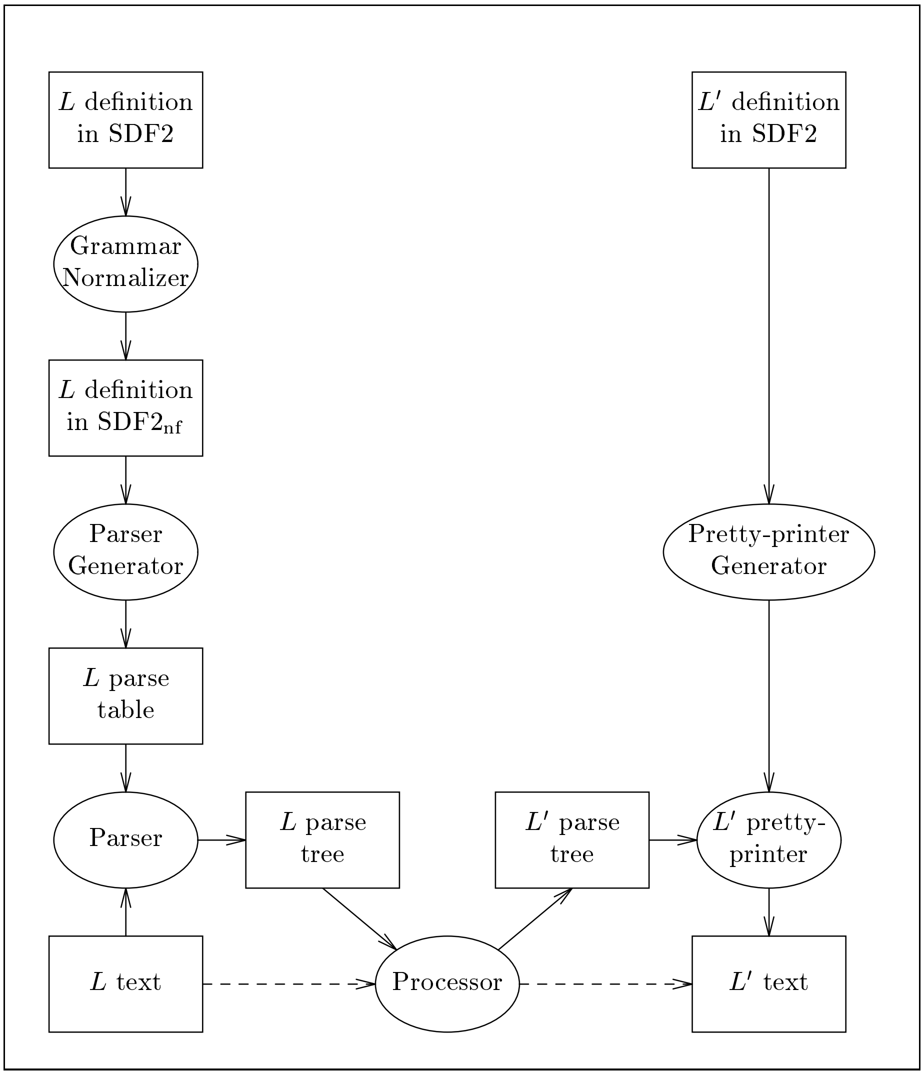


Figure 3.1: Architecture of an SDF2 application.

3.1.3 Contributions

The scannerless generalized-LR parsing approach presented in this chapter is a new powerful parsing method that supports concise specification of languages. The technical contributions (the details of which will be discussed later on) of the approach are:

— The normalization of grammars to eliminate features enhancing the expressivity of the formalism, in particular, the integration of lexical and context-free syntax by means of normalization into a single grammar.

— The use of GLR parsing for scannerless parsing to deal with unbounded lookahead.

— Static disambiguation by means of priorities by interpreting priority declarations in the parser generator. Priorities are completely expressed in the parse table.

— The use of character classes in grammars to compact the parse table.

— The use of follow restrictions to define longest match disambiguation and the interpretation of follow restrictions in the parse table.

— Prefer literals disambiguation by means of reject productions. Several expressivity results about context-free grammars with reject productions. Implementation of parsers for such grammars in an extension of the GLR algorithm.

3.1.4 Overview

In the next section we will argue in more detail that scannerless parsing has a number of definite advantages over parsing with scanners, but that it has not been introduced before because of the limitations of conventional parsing techniques. In the rest of the chapter we present several techniques that overcome these limitations and result in a combined approach encompassing grammar formalism and parsing techniques that does make scannerless parsing feasible.

3.2 Scannerless Parsing

The term *scannerless parsing* was coined by Salomon and Cormack (1989, 1995) to indicate parsing without a separate lexical analysis phase using a scanner based on a deterministic finite automaton. The parser directly reads the characters of a text. This entails the integration of the definition of lexical and context-free syntax in one grammar.

Consider the following SDF2 definition of a simple language of expressions consisting of identifiers, additions and multiplications.

```
sorts Id Exp
lexical syntax
  [a-z]+   -> Id
  [\ \t\n] -> LAYOUT
context-free syntax
  Id          -> Exp
  Exp "*" Exp -> Exp {left}
  Exp "+" Exp -> Exp {left}
context-free priorities
  Exp "*" Exp -> Exp >
  Exp "+" Exp -> Exp
```

The first line declares the sorts (say the non-terminals) of the grammar. The next three lines declare the lexical syntax of the language such that identifiers are lists of one or more lowercase letters and layout consists of spaces, tabs and newlines. The next four lines declare the context-free syntax of the language. An expression is either an identifier or an addition or multiplication of two expressions. Observe that the grammar is ambiguous and that in order to disambiguate it, priority and associativity declarations have been added. The last three lines declare that multiplication has higher priority than addition. The `left` attribute declares addition and multiplication to be left-associative.

The conventional way to interpret such a grammar to parse a string is as follows: (1) Divide the string into tokens according to the lexical syntax in all possible ways. (2) Apply lexical disambiguation rules to select the desired division. For instance, given the string `ab␣`, the rule 'prefer longest match' would prefer the division $\boxed{\texttt{ab}}\boxed{\texttt{␣}}$ over $\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\texttt{␣}}$, i.e., the longest possible identifier is selected. (3) Throw away layout tokens. (4) Parse the resulting token string according to the context-free syntax. The result is a parse tree that contains as leafs the tokens yielded by lexical analysis.

In scannerless parsing we have the following sequence: (1) Combine the definition of lexical and context-free syntax into a single context-free grammar. All tokens on the left-hand side of productions in the context-free syntax are explicitly separated by layout. All grammar symbols are renamed, such that the symbols occurring in the lexical syntax have the form ⟨_-LEX⟩ and those in the context-free syntax have the form ⟨_-CF⟩. This is done to keep the two levels separated. For instance, the addition production is transformed into

```
<Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>
```

The symbol ⟨LAYOUT?-CF⟩ represents the syntax of layout that can appear between tokens. In §3.3 this will be explained in more detail. The complete integrated grammar corresponding to the definition above is presented in Figure 3.3 on page 39. (2) Parse the characters of the string according to the normalized grammar. The result is a parse tree that contains as leafs the characters of the string. The tokens are recognizable as subtrees. For example, consider the parse tree in Figure 3.2. Observe that the symbols ⟨L-LEX⟩ and ⟨L-CF⟩ are abbreviations for ⟨LAYOUT-LEX⟩ and ⟨LAYOUT-CF⟩ and denote layout nodes.

In a sense, nothing is new. In a conventional parser, if we would instruct the scanner to make each character into a corresponding token, the parser that reads these tokens would in effect be 'scannerless'. The reason that we distinguish scannerless parsing from parsing with a real scanner is that the former generates some special problems that are avoided by using a scanner.

3.2.1 Advantages

Now that we have an understanding of what scannerless parsing is, we might ask why it is any good. We will discuss the advantages one by one.

No Scanner The obvious advantage of scannerless parsing is that no implementation of a scanner and scanner generator is needed and that the complicated interface between scanner and parser can be eliminated.

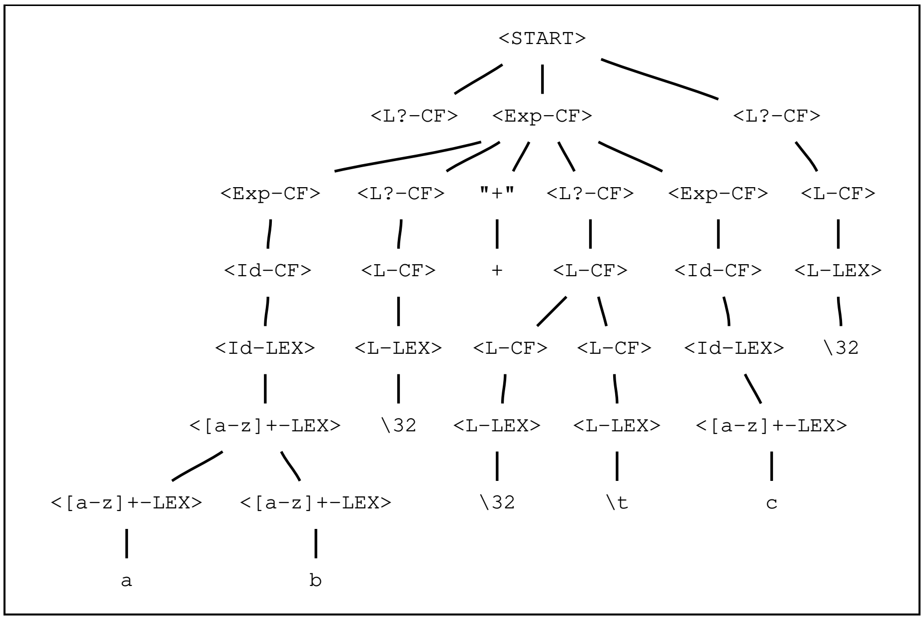


Figure 3.2: Parse tree for the string `ab␣+␣\tc␣`

Integrated Uniform Grammar Formalism A language is completely defined by means of one grammar. All grammar rules are explicit and formally specified. Lexical syntax and context-free syntax are specified with the same formalism. There is no longer a distinction between regular and context-free grammars. This makes the formalism more uniform and orthogonal. All features available for lexical syntax are available for context-free syntax and vice versa. This simplifies use and implementation of the formalism.

Disambiguation by Context Because of the integration of lexical and context-free syntax, lexical analysis is guided by context-free analysis. If a token does not make sense at some position, it will not even be considered. For instance, in the example above, the longest match rule does not have to be used to prefer `ab␣` over `a` `b` `␣` because the latter situation—two adjacent identifiers—is never syntactically correct.

The paradigmatic example of context-dependent lexical disambiguation is the interplay between subrange types and floating point number constants in Pascal. Subrange types have the form `k..l`, where `k` and `l` are constants. If floating point number constants could have the form `i.` and `.j` with `i` and `j` numbers, then `i..j` could be scanned either as `i` `..` `j`, i.e., the range from `i` to `j`, or as `i.` `.j`, i.e., two adjacent floating point numbers. In scannerless parsing, this ambiguity is solved automatically by context. A scanner that has no access to the context and applies the longest match rule, would always choose the second possibility (two adjacent numbers) and fail. Apparently for this reason the syntax of Pascal only allows real numbers of the form `i.j`, where `i` and `j` are non-empty lists of digits (Jensen and Wirth, 1975). Similar examples can be found in many languages.

Another example where the parsing context is relevant for making lexical decisions is the syntax of lists of statements that can be separated by semicolons or newlines. Consider the grammar

```
lexical syntax
  [\ \t\n] -> LAYOUT
context-free syntax
  "begin" {Stat ";"|"\n"}* "end" -> Block
```

The lexical syntax defines newlines (`\n`) to be layout. The context-free syntax defines blocks as lists of zero or more statements starting with the keyword `begin` and ending with the keyword `end`. The list is declared by the construct $\{A\ B\}*$, which declares a list of As separated by Bs, i.e., a list of the form $A\ B\ A\ \ldots\ B\ A$. In this case the separator is either a semicolon or a newline. This means that newlines are both layout and non-layout. If the disambiguation rule 'prefer non-layout' is applied to the tokens of this language, all newlines—even those not used as separator of statements—will be wrongly characterized as non-layout. A scannerless parser will recognize the newlines used as separator simply by considering the parsing context.

Conservation of Lexical Structure Scanners do usually not maintain the phrase structure of the tokens they produce. For example, the grammar

```
lexical syntax
  [a-z]+          -> Id
  "/"? {Id "/"}+ -> Path
```

defines the lexical syntax of path expressions as occur, for instance, in the naming conventions of tree-structured filesystems. This syntax has to be lexical since no layout should occur between the identifiers and separators of a path. A scanner would produce a string containing the characters of a path expression without the structure assigned to it by the grammar, i.e., the distinction between identifiers in the path is lost. This entails that the semantic processor must reparse such tokens to deal with their internal structure.

Conservation of Layout Scanners throw away the layout between tokens of a phrase. In this way the parser can ignore layout, which simplifies the parsing problem. However, there are examples of operations on programs that require the structure of the program, i.e., the parse tree, but also the layout in the source. Examples are source to source translations, transformations on the source text and program documentation tools. Although a conventional parser could be instructed to add the layout to the parse tree via some detour, this would usually require a non-standard extension of the method. If the layout would be explicitly specified in the grammar we would get an approach that is very similar to scannerless parsing.

Expressive Lexical Syntax Context-free grammars provide a more expressive grammar formalism for lexical syntax than regular grammars. This additional expressive power opens the way to concise definitions of nested comments and syntactically correct expressions in comments. For example, consider the following extension of the expression grammar above that defines C-like comments as a list of comment words between `/*` and `*/`.

```
sorts ComWord Comment
lexical syntax
  ~[\ \t\n\|\/]+ -> ComWord
context-free syntax
  "/*" ComWord* "*/" -> Comment
  Comment            -> LAYOUT
```

A comment word is a non-empty list of characters that are not whitespace, `|` or `/`. Since the definition of comments is part of the context-free syntax, comment words can be separated by layout. These comments are made into layout by the last line of the grammar, which is an injection of comment into layout. Because layout can occur between any two adjacent tokens, comment can as well.

According to this definition, comments can be nested, because comment words are separated by layout, which includes comments. For instance, the string

```
  h /* height */
/*
  * w /* width */
  * d /* depth */
*/
```

is a syntactically correct expression over the grammar above in which part of an expression including comments is commented out. This is a tedious job if nested comments are not supported by the language.

Moreover, the following extension of the grammar above defines that a comment word can also be an expression between two `|`s.

```
context-free syntax
  "|" Exp "|" -> ComWord
```

This entails that comments can contain quoted expressions that must be syntactically correct. For instance, the following sentence contains the expression `x + y` as part of a comment.

```
 a + b /* an expression |x + y| denotes
          the addition of |x| and |y| */
 + c
```

This is useful for typesetting comments in literate programs and for generating cross-references.

3.2.2 Problems & Solutions

Now one might ask why scannerless parsing was not introduced earlier, if it has so many advantages. The answer is that there are several problems caused by the integration of lexical and context-free syntax as well. In this chapter we discuss solutions to these problems that make scannerless parsing feasible.

Limitations of Parsing Techniques The main problem with scannerless parsing are the limitations of the conventional deterministic parsing techniques. Most complete character level grammars are *not* LR(1), LL(1), or even LR(k) due to lookahead needed for lexical elements. When parsing with a scanner a lookahead of 1 entails looking one token ahead. In scannerless parsing a lookahead of 1 entails only considering the next character. Furthermore, when layout is skipped by the scanner this need not be considered in the lookahead. The solution used in the SDF2 implementation is to use the generalized-LR parsing algorithm of Tomita (1985) and Rekers (1992) to get dynamic lookahead.

Grammar Size Another problem is the size of grammars. Complete character level grammars are large because all constructs have to be specified down to the character level. Furthermore, the placement of layout between tokens should be explicitly declared in productions. For maintenance and readability of grammars this is problematic. To support the development of complete character level grammars an expressive formalism is needed that hides the details of the interface between lexical and context-free syntax and of the placement of layout. In §3.3 we discuss the approach of grammar normalization in order to provide an expressive formalism with a minimal semantic basis. In §3.4 we discuss the extension of context-free grammars with various disambiguation constructs to keep grammars concise.

Lexical Disambiguation Although many lexical ambiguities are solved automatically through the integration of lexical and context-free syntax, there are still cases where disambiguation of lexical constructs needs to be expressed. Since lexical analysis is now based on context-free parsing, familiar lexical disambiguation rules such as 'prefer longest match' and 'prefer keyword' have to be redefined and their implementation reconsidered. In §3.4 we discuss two disambiguation constructs for lexical disambiguation: follow restriction and reject productions that suffice to express all common lexical disambiguation rules.

Interpretation of Disambiguation Rules There are a number of ways to interpret disambiguation constructs. One possibility is to implement them as a filter on parse forests as proposed in Chapter 4. However, for disambiguation of lexical constructs and context-free expressions with priorities this can lead to an exponential size of the parse forest before filtering, which makes the method too inefficient. In §3.5 we discuss the techniques used in parser generation to encode disambiguation rules in the parse tables such that decisions are taken early. In §3.8 an extension of the GLR parsing algorithm with reject reductions is presented.

Efficiency The first problem that comes to mind when considering scannerless parsing is efficiency. Since scanning with a finite automaton has a lower complexity than parsing with a stack, scannerless parsing, i.e., replacing the finite automaton part by a stack machine, should be less efficient. The following considerations led us to attempt scannerless parsing, nonetheless: (1) LR parsing is linear, in particular for regular grammars. Since lexical syntax is traditionally formulated by means of regular grammars, we should expect linear behaviour for the lexical part of scannerless parsers. (2) The complete complexity of the scanner/parser setup should be considered including lexical disambiguation. If lexical disambiguation rules cannot solve all ambiguities and disambiguation has to be deferred to the parser, a kind of graph structured stack has to be maintained to keep track of the possible segmentation of the string in tokens. (Such a setup is used in the ASF+SDF Meta-Environment (Klint, 1993) that forms the background for the development of SDF2.) It seems even more efficient to maintain a single graph structured stack, instead of two. (3) If more complex grammars for lexical syntax are used, we get into an area where scannerless parsing and parsing with scanners can no longer be properly compared because such syntax is not expressible in the scanner framework. Therefore, the worst case complexity of context-free parsing should not be taken as a reference point for considering the complexity of scannerless parsing.

Of course, these considerations should be verified by means of experiments. However, experiments with scannerless parsing can only be performed after solutions have been found for the other problems discussed above. It seems that these problems are the cause for the late introduction of scannerless parsing rather than bad efficiency of the method. In §3.9 we will discuss a few simple experiments that have been performed with the scannerless parsing method described in this chapter and that seem to confirm our expectations.

3.3 Grammar Normalization

We need an expressive grammar formalism in which lexical syntax and context-free syntax are integrated and that supports concise syntax definitions. SDF2 is such an expressive formalism. It provides regular expressions, lexical and context-free syntax, character classes, literals, priorities, modules, renaming, and aliases. The first version of the formalism was described in Visser (1995b). The full definition is presented in Part II. Because it is expensive to extend tools to such an expressive formalism, all features that are expressible in more primitive features are eliminated by means of a normalization function on grammars.

3.3.1 Normal Form

The expressive power of the syntax definition formalism SDF2 can be characterized by the equation

$$\begin{aligned}\text{SDF2} = {} & \text{context-free grammars} + \text{character-classes} + \text{priorities} \\ & + \text{reject productions} + \text{follow restrictions}\end{aligned}$$

That is, any SDF2 definition is equivalent to a context-free grammar making use of character classes, priorities, reject productions and follow restrictions. All other features are expressed in terms of these features. The equivalence is such that a definition is equivalent to a definition of the form

sorts $s_1 \ldots s_j$
syntax $p_1 \ldots p_k$
priorities $pr_1, \ldots, pr_l$
restrictions $r_1 \ldots r_m$

where the s_i are sort symbols, the p_i are context-free productions of the form $\alpha \to A$, the pr_i priority declarations of the form p_j R $p_{j'}$ with R a priority relation, and the r_i follow restrictions of the form $\alpha \not\rightharpoondown cc$ with α a list of symbols and cc a character class.

A production can have a number of attributes that may include the attribute `reject`, which makes the production a reject production. A priority relation is one of `left`, `right`, `assoc`, `non-assoc` or `>`. A symbol can be a character class or some other symbol. Only character classes are interpreted during parser generation. Other symbols constructed using symbol operators are simply interpreted as a name. For instance, the symbol $A+$ used to indicate the iteration of symbol A has no special meaning after normalization.

Given a grammar $\mathcal{G}$ the following projection functions are defined:

$$
\begin{aligned}
\mathrm{S}(\mathcal{G}) &\mapsto \text{sorts of } \mathcal{G} \\
\mathrm{Syms}(\mathcal{G}) &\mapsto \text{symbols used in } \mathcal{G} \\
\mathrm{P}(\mathcal{G}) &\mapsto \text{productions of } \mathcal{G} \\
\mathrm{Pr}(\mathcal{G}) &\mapsto \text{priorities of } \mathcal{G} \\
\mathrm{R}(\mathcal{G}) &\mapsto \text{restrictions of } \mathcal{G}
\end{aligned}
$$

3.3.2 Normalization

As an example of the normal form, consider the grammar in Figure 3.3. It completely describes the lexical and context-free syntax of expressions with identifier, multiplication and addition—the same language described in the example in §3.2. In fact, this grammar is derived from that grammar by application of a normalization procedure. We briefly discuss the elements of this normalization that is formally specified in Part II. Refer to Figure 3.3 for examples of the normalization rules.

Lexical and Context-free Syntax The most important aspect of the normalization for this chapter is the integration of lexical and context-free syntax. The productions of lexical and context-free syntax are merged. In order to avoid interference of lexical and context-free syntax the symbols in productions are renamed. The symbols in the lexical syntax—except for character classes and literals—are renamed using the symbol constructor ⟨_`-LEX`⟩. For instance, `Id` becomes ⟨`Id-LEX`⟩ and `[\97-\122]+` becomes `<[\97-\122]+-LEX>`. Similarly, the symbols in the context-free syntax are renamed using ⟨_`-CF`⟩. Furthermore, the symbols on the left-hand side of context-free productions are separated by $\langle$`LAYOUT?-CF`$\rangle$, which entails that layout can occur at that position. In this way two disjunct sets of symbols are created. The interface between lexical and context-free syntax is now expressed by an injection $\langle A\texttt{-LEX}\rangle \to \langle A\texttt{-CF}\rangle$ for each symbol A used both in the lexical and the context-free syntax.

Top Symbol A syntax definition defines a number of symbols. A text over such a definition can be one produced by any of its symbols. For context-free parsing we need a single start symbol from which all strings are generated. For this purpose for each sort A a production

$$\langle\texttt{LAYOUT?-CF}\rangle\ \langle A\texttt{-CF}\rangle\ \langle\texttt{LAYOUT?-CF}\rangle \to \langle\texttt{START}\rangle$$

is added to the grammar, defining the start symbol $\langle$`START`$\rangle$. The production also defines that a string can start and end with layout. Furthermore, to express the termination of a string the production

$$\langle\texttt{START}\rangle\ [\backslash\texttt{EOF}] \to \langle\texttt{Start}\rangle$$

defines that a string consists of a string generated by $\langle$`START`$\rangle$ followed by the end of file character.

Character Classes Character classes are expressions of the form $[cr_1 \ldots cr_n]$ where the cr_i are either characters or character ranges of the form $c{-}c'$. Character classes are normalized to a unique normal form by translating the characters to a numeric character code—the ASCII code—and by ordering and merging the ranges such that they are in increasing order and do not overlap. This normalization is formally specified and proven correct with respect to the set interpretation of character classes in Visser (1997b).

Literals Literals are abbreviations for fixed lists of characters. Literals are defined in terms of a production with the literal as result and singleton character classes corresponding to the characters as arguments. For example, the production

```
[\108] [\101] [\116] -> "let"
```

defines the literal `"let"` as the sequence of characters `l`, `e` and `t` in ASCII.

Regular Expressions An extensive set of regular expressions including optional, alternative, tupling, several kinds of iteration and permutation are expressed by means of defining productions. For instance, consider the definition of `<[\97-\122]+-LEX>` in Figure 3.3, which defines a list of one or more lowercase letters.

Priorities Priorities can be declared using chains of > declarations and associativities of productions can be declared using groups and attributes. These are all defined in terms of binary priority and associativity declarations.

```
sorts Id Exp
syntax
  [\9-\10\32]                                -> <LAYOUT-LEX>
  <LAYOUT-LEX>                               -> <LAYOUT-CF>
  <LAYOUT-CF> <LAYOUT-CF>                    -> <LAYOUT-CF> {left}
                                             -> <LAYOUT?-CF>
  <LAYOUT-CF>                                -> <LAYOUT?-CF>

  [\42]                                      -> "*"
  [\43]                                      -> "+"

  [\97-\122]                                 -> <[\97-\122]+-LEX>
  <[\97-\122]+-LEX> <[\97-\122]+-LEX>        -> <[\97-\122]+-LEX>
                                                            {left}
  <[\97-\122]+-LEX>                          -> <Id-LEX>
  <Id-LEX>                                   -> <Id-CF>

  <Id-CF>                                      -> <Exp-CF>
  <Exp-CF> <LAYOUT?-CF> "*" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> {left}
  <Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> {left}

  <LAYOUT?-CF> <Id-CF>  <LAYOUT?-CF>           -> <START>
  <LAYOUT?-CF> <Exp-CF> <LAYOUT?-CF>           -> <START>
  <START> [\EOF]                               -> <Start>
priorities
  <[\97-\122]+-LEX> <[\97-\122]+-LEX>        -> <[\97-\122]+-LEX>
  left
  <[\97-\122]+-LEX> <[\97-\122]+-LEX>        -> <[\97-\122]+-LEX>,
  <LAYOUT-CF> <LAYOUT-CF>                    -> <LAYOUT-CF> left
  <LAYOUT-CF> <LAYOUT-CF>                    -> <LAYOUT-CF>,
  <Exp-CF> <LAYOUT?-CF> "*" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> >
  <Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>,
  <Exp-CF> <LAYOUT?-CF> "*" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> left
  <Exp-CF> <LAYOUT?-CF> "*" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>,
  <Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> left
  <Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>
```

Figure 3.3: Expression grammar in normal form. The grammar contains no restrictions or reject productions.

Modules An SDF2 definition can be divided over a number of modules. Modules can import other modules. This is used to share common syntax definitions in several language definitions. Renamings of symbols and productions can be used to adapt the definition in a module to some application. Furthermore, symbol aliases can be used to abbreviate long regular expressions. Modular syntax definitions are completely expanded by the normalization function.

3.3.3 Semantics

A syntax definition defines a language, i.e., a set of strings, and the structure that is assigned to those strings. The strings of the language are important to its users who write down programs. The structure of those strings is important for the definition of language processors such as compilers, interpreters and typecheckers. The productions of an SDF2 definition describe both the language and the structure assigned to strings in the language. The semantics of a syntax definition is a set of parse trees from which a set of strings can be derived. The mapping from trees to strings is achieved by taking the yield of a tree. The reverse mapping from strings to trees is called parsing. At this point, we formally define the semantics of context-free grammars without considering disambiguation rules such as priorities, reject productions and follow restrictions.

A context-free grammar $\mathcal{G}$ generates a family of sets of *parse trees* $\mathcal{T}(\mathcal{G}) = (\mathcal{T}(\mathcal{G})(X) \mid X \in \mathrm{Syms}(\mathcal{G}))$, which contains the minimal sets $\mathcal{T}(\mathcal{G})(X)$ such that

$$\frac{c \in cc}{c \in \mathcal{T}(\mathcal{G})(cc)} \tag{Char}$$

$$\frac{A_1 \ldots A_n \to A \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}(\mathcal{G})(A_1),\ \ldots\ ,\ t_n \in \mathcal{T}(\mathcal{G})(A_n)}{[t_1 \ldots t_n \to A] \in \mathcal{T}(\mathcal{G})(A)} \tag{Prod}$$

In rule (Char) c is a character and cc a character class. We will write t_α for a list $t_1 \ldots t_n$ of trees where α is the list of symbols $X_1 \ldots X_n$ and $t_i \in \mathcal{T}(\mathcal{G})(X_i)$ for $1 \leq i \leq n$. Correspondingly we will denote the set of all lists of trees of type α as $\mathcal{T}(\mathcal{G})(\alpha)$. Using this notation $[t_1 \ldots t_n \to A]$ can be written as $[t_\alpha \to A]$ and the concatenation of two lists of trees t_α and t_β is written as $t_\alpha t_\beta$ and yields a list of trees of type $\alpha\beta$.

The *yield* of a tree is the concatenation of its leafs. The *language* defined by a grammar $\mathcal{G}$ is the family $L(\mathcal{G}) = (L(\mathcal{G})(X) \mid X \in \mathrm{Syms}(\mathcal{G}))$ of sets of strings that are yields of trees over the grammar, i.e., $L(\mathcal{G})(X) = \mathrm{yield}(\mathcal{T}(\mathcal{G})(X))$. A *parser* is a function Π that maps a string of characters to a set of parse trees. A parser Π *accepts* a string w if $|\Pi(w)| > 0$. A parser for a context-free grammar $\mathcal{G}$ that accepts exactly the sentences in $L(\mathcal{G})$ is defined by

$$\Pi(\mathcal{G})(w) = \{t \in \mathcal{T}(\mathcal{G})(X) \mid X \in \mathrm{Syms}(\mathcal{G}),\ \mathrm{yield}(t) = w\}$$

A parser Π is *deterministic* if $|\Pi(w)| \leq 1$ for all strings w. A grammar is ambiguous if there are strings with more than one parse tree, i.e, $|\Pi(\mathcal{G})(w)| > 1$.

3.4 Disambiguation

Disambiguation methods are used to select the intendend tree from a set of possible parse trees for an ambiguous string. SDF2 provides three disambiguation methods. Priority and associativity declarations are used to disambiguate concise expression grammars. Follow restrictions and reject productions are used to express lexical disambiguation. In this section we discuss these methods.

3.4.1 Disambiguation by Priorities

By using priority and associativity declarations, fewer grammar symbols have to be introduced and a more compact abstract syntax can be achieved. Consider the following grammar of expressions in a functional programming language with binary function application and let binding.

```
sorts Var Term
lexical syntax
  [a-z]+   -> Var
  [\ \t\n] -> LAYOUT
context-free syntax
  Var                          -> Term
  Term Term                    -> Term {left}
  "let" Var "=" Term "in" Term -> Term
  Term "=" Term                -> Term {non-assoc}
  "(" Term ")"                 -> Term {bracket}
context-free priorities
  Term Term                    -> Term >
  Term "=" Term                -> Term >
  "let" Var "=" Term "in" Term -> Term
```

An example term over this grammar is

```
let sum = foldr plus zero in sum lst
```

The grammar is disambiguated by means of priorities. The binary application operator is declared as left-associative. This entails that $x\ y\ z$ should be read as $(x\ y)\ z$ and not as $x\ (y\ z)$. This is illustrated in Figure 3.4 that shows the right- and left-associative parse trees for three adjacent terms. The priority declaration defines applications to have higher priority than equalities. Consider the trees in Figure 3.5. According to the priority declaration, the first tree has a priority conflict and therefore only the second tree is a correct parse tree. The following definition formally defines the notion of priority conflicts.

Definition 3.4.1 Given some grammar $\mathcal{G}$ with priority declarations $\mathrm{Pr}(\mathcal{G})$, the set $\mathrm{conflicts}(\mathcal{G})$ of priority conflicts over grammar $\mathcal{G}$ is the smallest set of parse tree patterns of the form $[\alpha[\beta \to B]\gamma \to A]$ such that:

$$\frac{\alpha B\gamma \to A > \beta \to B \in \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})} \tag{CF1}$$

$$\frac{\gamma \neq \epsilon,\ \beta \to B\ (\mathtt{right} \cup \mathtt{non\text{-}assoc})\ B\gamma \to A \in \mathrm{Pr}(\mathcal{G})}{[[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})} \tag{CF2}$$

$$\frac{\alpha \neq \epsilon,\ \beta \to B\ (\mathtt{left} \cup \mathtt{assoc} \cup \mathtt{non\text{-}assoc})\ \alpha B \to A \in \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B] \to A] \in \mathrm{conflicts}(\mathcal{G})} \tag{CF3}$$

A parse tree over a grammar $\mathcal{G}$ has a priority conflict if one of its nodes matches a pattern $[\alpha[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})$. □

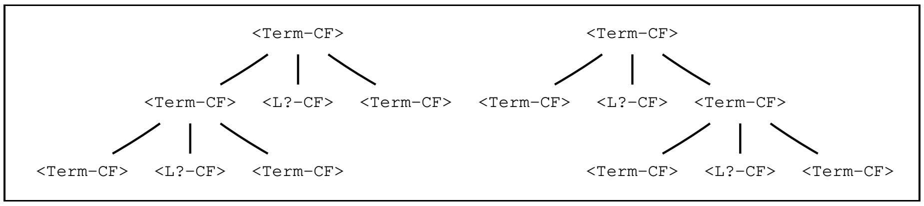


Figure 3.4: Left- and right-associative parse trees for binary term application.

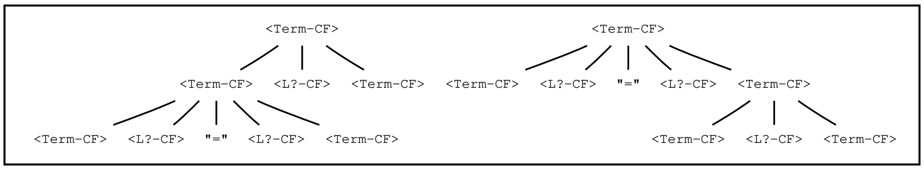


Figure 3.5: Two parse trees for application and equality.

Using the notion of priority conflicts we can define a filter on sets of parse trees that selects the trees without a conflict. For example, according to rule (CF3) and because of the declaration of application as a left-associative operator, the pattern

$$[\langle\texttt{T-CF}\rangle\ \langle\texttt{L?-CF}\rangle\ [\langle\texttt{T-CF}\rangle\ \langle\texttt{L?-CF}\rangle\ \langle\texttt{T-CF}\rangle \rightarrow \langle\texttt{T-CF}\rangle] \rightarrow \langle\texttt{T-CF}\rangle]$$

describes a tree with a conflict. (`Term` and `LAYOUT` are abbreviated to `T` and `L`, respectively.) Therefore, the second tree in Figure 3.4 has a conflict and the first one is selected by the disambiguation method. According to rule (CF1) and because application has higher priority than equality, the pattern

$$[[\langle\texttt{T-CF}\rangle\ \langle\texttt{L?-CF}\rangle\ \texttt{"="}\ \langle\texttt{L?-CF}\rangle\ \langle\texttt{T-CF}\rangle \rightarrow \langle\texttt{T-CF}\rangle]\ \langle\texttt{L?-CF}\rangle\ \langle\texttt{T-CF}\rangle \rightarrow \langle\texttt{T-CF}\rangle]$$

is a member of the conflicts generated by the functional language grammar. This means that the first tree in Figure 3.5 has a priority conflict. The second tree has no conflict.

3.4.2 Lexical Disambiguation

If we consider the example of functional expressions again we see that it contains two occurrences of lexical ambiguities.

Longest Match In the first place there is a longest match problem caused by the syntax-less binary application operator. Two adjacent letters could be the concatenation of two letters forming a variable, or it could be the application of two single letter variables. Figure 3.6 shows two parse trees for the string `fa`. In the first tree the concatenation of letter lists is used to make them into a single variable. In the second tree each letter is interpreted as a variable on its own. We want to solve this ambiguity by means of the longest match rule that prefers the longest possible lexical token. In this case the string `fa` as a single variable. We define the longest match notion formally by comparing the lengths of tokens. For this definition we first need the notion of the token stream associated to a parse tree.

Definition 3.4.2 (Token Stream) The *token stream* associated with a parse tree is the list of subtrees that have as root either an injection $\langle A$-`LEX`$\rangle \to \langle A$-`CF`$\rangle$ or a literal defining production. The length $|t|$ of a token t is the number of characters in its yield. □

According to this definition the token streams for the trees in Figure 3.6 are the single token

[[[[f → `<[a-z]+-LEX>`][a → `<[a-z]+-LEX>`] → `<[a-z]+-LEX>`]
→ `<Var-LEX>`] → `<Var-CF>`]

and the tokens

[[[f → `<[a-z]+-LEX>`] → `<Var-LEX>`] → `<Var-LEX>`]
[[[a → `<[a-z]+-LEX>`] → `<Var-LEX>`] → `<Var-LEX>`]

The idea of longest match disambiguation is to compare two token streams from left to right. While the tokens have the same length the streams are similar. The first token that differs in length solves the ambiguity by taking the tree associated with the longer token. In the example above, the first token stream is larger because its first token has length 2 while the first token of the second stream has length 1.

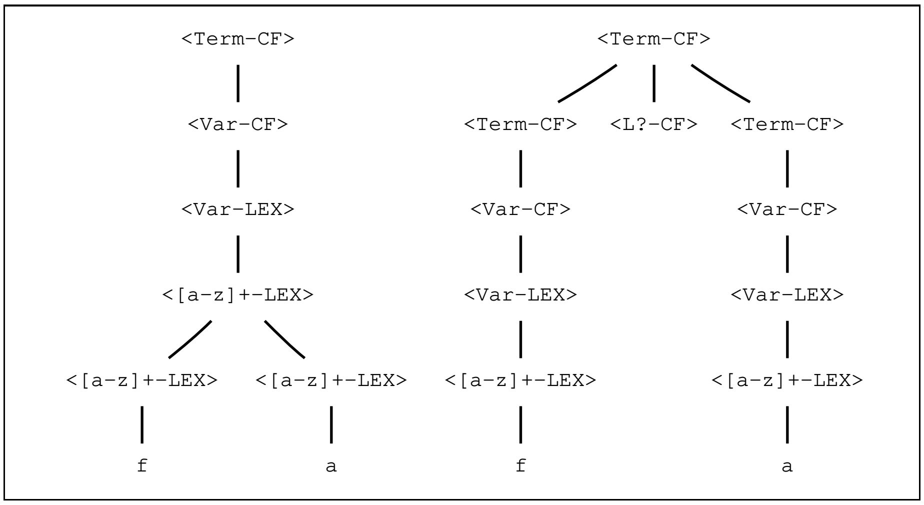


Figure 3.6: Two parse trees for `fa` over the functional expression grammar.

Formally we have the following definition of longest match disambiguation:

Definition 3.4.3 (Longest Match) Given the token streams $t_1 \ldots t_n$ associated with the tree t and $s_1 \ldots s_m$ associated with tree s, tree t is larger in the *longest match ordering* $>_{\mathrm{lm}}$ than s ($t >_{\mathrm{lm}} s$), if there is some $1 \leq i \leq \min(n, m)$ such that $|t_j| = |s_j|$ for $1 \leq j < i$ and $|t_i| > |s_i|$. □

This definition can be used as a method to filter parse forests by selecting the largest trees according to the longest match ordering. However, because a longest match ambiguity causes an exponential explosion of the parse forest this is not feasible. We need a method that can be applied during parsing, if possible as a filter on parse tables. A naive solution for the longest match problem in the example above is to require non-empty layout as a separator between the two terms of an application. In the example this would indeed solve the problem because the second tree would be forbidden. However, this solution is immediately refuted by considering the expression `f(a)`, where brackets are used around the argument.

A method that works in all cases we have encountered so far is that of follow restrictions. A follow restriction of the form $A_1 \ldots A_n \not\rightharpoonup cc$ declares that the symbols A_i should not be followed by any of the characters in the character class cc. In the example above the restriction

```
lexical restrictions
  Var -/- [a-z]
```

forbids a variable to be followed by a letter. This entails that the second tree in Figure 3.6 violates the follow restrictions and the desired first tree is selected.

Prefer Literals The second problem in the functional expression grammar is the overlap between the literals `"let"` and `"in"` and variables. This is particularly problematic in combination with the `=` operator on terms. A let binding `let x = t1 in t2` can be interpreted also as an equality `(let x) = (t1 in t2)`, where `let` and `in` are now read as variables. We clearly want to declare `let` and `in` as reserved words of the language that should not be used as variables. This lexical disambiguation rule is called 'prefer literals' and can be defined formally as follows.

Definition 3.4.4 (Prefer Literals) A tree violates the prefer literals rule if it contains a subtree with function $\langle A\text{-}\mathtt{LEX}\rangle \rightarrow \langle A\text{-}\mathtt{CF}\rangle$ and the yield of that tree is also used as literal in the grammar. □

This rule can be expressed by means of reject productions. A reject production is a production $\alpha \rightarrow A$ attributed with the attribute `reject`. It declares that a string is not of type A if it can also be derived from α. For example to disambiguate the grammar above we add the following productions.

```
lexical syntax
  "let" -> Var {reject}
  "in"  -> Var {reject}
```

This creates an ambiguity: `let` can be a variable in two ways, via the lexical definition or via the production above. Because this is a reject production both derivations are forbidden, i.e., `let` can only occur in the context of a `let` binding. We also need the restrictions

```
lexical restrictions
  "let" "in" -/- [a-z]
```

to prevent `letter` to be interpreted as the literal `let` and the variable `ter`. We will further discuss some properties of reject productions in §3.7.

Automatic Lexical Disambiguation We have defined two extensions of context-free grammars that enable us to express lexical disambiguation rules on grammars for integrated lexical and context-free syntax. However, it is desirable to derive the rules for lexical disambiguation automatically from the grammar. In §3.6 we will discuss this issue, after we have discussed parser generation.

3.5 Parser Generation

We have discussed a grammar formalism with disambiguation methods for concise definition of lexical and context-free syntax of languages. Now we turn our attention to deriving parsers from such syntax definitions. In this section we present the rules for the generation of parse tables for a shift-reduce parser. The rules constitute a modification of the well known SLR(1) algorithm. We first discuss shift-reduce parsing.

3.5.1 Shift-Reduce Parsing

A shift-reduce parser is a transition system that manipulates its state consisting of a stack and an input stream by repeatedly shifting a symbol from the input to the stack or reducing a number of elements on top of the stack to a single element until it enters an accepting state. The transitions between parse configurations are determined by the functions 'actions' and 'goto' as defined by the following transition rules:

$$\frac{\text{actions}(s_m, a_i) \ni \texttt{shift}(s_{m+1})}{(s_0 t_1 s_1 \ldots t_m s_m \bullet a_i \ldots a_n) \Rightarrow (s_0 t_1 s_1 \ldots t_m s_m a_i s_{m+1} \bullet a_{i+1} \ldots a_n)} \qquad \text{(Shi)}$$

$$\frac{\begin{array}{c}\text{actions}(s_{m+k}, a_i) \ni \texttt{reduce}(p, k),\\ s = \text{goto}(s_m, p),\ t = \text{tree}(p, [t_{m+1}, \ldots, t_{m+k}])\end{array}}{\begin{array}{c}(s_0 t_1 \ldots t_m s_m t_{m+1} s_{m+1} \ldots t_{m+k} s_{m+k} \bullet a_i \ldots a_n)\\ \Rightarrow (s_0 t_1 \ldots t_m s_m t s \bullet a_i \ldots a_n)\end{array}} \qquad \text{(Red)}$$

$$\frac{\text{actions}(s_1, \texttt{\textbackslash EOF}) \ni \texttt{accept}}{(s_0 t_1 s_1 \bullet \texttt{\textbackslash EOF}) \Rightarrow \texttt{accept}(t)} \qquad \text{(Acc)}$$

Here a configuration $(s_0 t_1 s_1 \ldots t_m s_m \bullet a_i \ldots a_n)$ consists of a stack on the left side of the $\bullet$ and a list of input characters on the right side of the $\bullet$. The stack is filled alternatingly with states s and trees t. Parsing starts in the configuration $C_0 = (s_0 \bullet a_1 \ldots a_n)$, where s_0 is the initial state of the parser. Parsing succeeds if there is some sequence of steps $C_0 \Rightarrow C_1 \Rightarrow \ldots \Rightarrow \texttt{accept}(t)$ that ends in the accepting configuration $\texttt{accept}(t)$.

There are various ways to define the actions and goto functions that drive a shift-reduce parser. The SLR(1) algorithm of DeRemer (1971) and Anderson *et al.* (1973) is a simplification of the LR(k) parsing algorithm of Knuth (1965). It works by first constructing an LR(0) parse table. This involves no lookahead sets in the parse items. The lookahead of reductions is constrained to the follow set of the nonterminal defined by the production being reduced.

In the rest of this section we describe a modification of the SLR(1) algorithm that incorporates priorities and follow restrictions. This modification is based on the derivation in Chapter 5, where starting with a schema for Earley's parsing algorithm, a parsing schema is derived such that the parser does not build trees with priority conflicts. Other changes are the use of character classes, the use of productions instead of symbols in follow and goto and the interpretation of follow restrictions to restrict the lookahead set of reductions.

3.5.2 First

The first set for a symbol contains those symbols with which a phrase for the symbol can start. Given some grammar $\mathcal{G}$, define for each list of symbols α and each character class cc the first characters in α followed by cc is the smallest character class $\mathrm{first}(\alpha, cc)$ such that:

$$\mathrm{first}(\epsilon, cc) = cc \tag{Fi1}$$

$$\mathrm{first}(cc'\ \alpha, cc) = cc' \tag{Fi2}$$

$$\frac{\alpha \to A \in \mathrm{P}(\mathcal{G})}{\mathrm{first}(A\beta, cc) \supseteq \mathrm{first}(\alpha\beta, cc)} \tag{Fi3}$$

The definition of the first set can be extended to the set of symbols that starts a sentence derived from a list of symbols.

$$\frac{\alpha \to A \in \mathrm{P}(\mathcal{G})}{\mathrm{first}_s(A\beta, \Phi) \supseteq \{A\} \cup \mathrm{first}(\alpha\beta, \Phi)} \tag{Fi4}$$

3.5.3 Follow

In the conventional SLR(1) algorithm the follow set is computed for each nonterminal of the grammar. It maps a nonterminal to the set of terminals that can follow that nonterminal in a sentence, i.e.,

$$\frac{\alpha A \beta \to^* \langle\texttt{Start}\rangle}{\mathrm{follow}(A) \supseteq \mathrm{first}(\beta, \emptyset)}$$

This can be computed as the closure of

$$\frac{\beta A \gamma \to B \in \mathrm{P}(\mathcal{G})}{\mathrm{follow}(A) \supseteq \mathrm{first}(\gamma, \mathrm{follow}(B))} \tag{Fo1}$$

that adds the characters in the first set of γ to the follow set of A if γ follows A in some production. The follow of B is added in case γ can also produce the empty string.

This notion can be refined to the follow-set of productions. The rule

$$\frac{\beta A\gamma \to B \in \mathrm{P}(\mathcal{G})}{\mathrm{follow}(\alpha \to A) \supseteq \mathrm{first}(\gamma, \mathrm{follow}(\beta A\gamma \to B))} \tag{Fo2}$$

defines the follow-set of production $\alpha \to A$ as those characters that can follow A in some context. In case of plain context-free grammars, rule (Fo2) has the same effect as rule (Fo1). But if we consider priorities, the rule is extended to

$$\frac{\beta A\gamma \to B \in \mathrm{P}(\mathcal{G}), [\beta[\alpha \to A]\gamma \to B] \notin \mathrm{conflicts}(\mathcal{G})}{\mathrm{follow}(\alpha \to A) \supseteq \mathrm{first}(\gamma, \mathrm{follow}(\beta A\gamma \to B))} \tag{Fo3}$$

Here the follow-set of a production is restricted to those contexts where it can actually be used without causing a priority conflict. For instance, in the expression grammar of §3.2, the follow-set of the addition production does not contain the character $*$ because addition can not occur as a direct descendant of multiplication.

Finally, if the grammar also defines follow restriction rules $A \not\rightharpoondown cc$, the follow-set of a production for A can be further restricted as

$$\frac{\beta A\gamma \to B \in \mathrm{P}(\mathcal{G}), [\beta[\alpha \to A]\gamma \to B] \notin \mathrm{conflicts}(\mathcal{G}), A \not\rightharpoondown cc \in \mathrm{R}(\mathcal{G})}{\mathrm{follow}(\alpha \to A) \supseteq \mathrm{first}(\gamma, \mathrm{follow}(\beta A\gamma \to B)) \setminus cc} \tag{Fo4}$$

The production can be followed by the difference of the first set of the right context and the character class cc.

To see the effect of the last rule consider the follow-set of the production `[a-z]+ -> Var` in the functional expression grammar of the previous section. Because of the application production and the injection of variables into terms, the follow-set of `[a-z]+ -> Var` is `[\EOF\t\n\ \(\)\=a-z]`. The lexical restriction `Var -/- [a-z]` removes the character class `[a-z]` from this follow-set, resulting in `[\EOF\t\n\ \(\)\=]`. This entails that a variable cannot directly be followed by a letter.

3.5.4 Goto Table

The states of an LR parser are formed by item-sets. An item is an object of the form $[\alpha \bullet \beta \to A]$, i.e., a context-free production with a $\bullet$ somewhere in between the symbols on the left-hand side. Such an item indicates that a sentential form of type α has already been recognized.

The initial state of the parser for grammar $\mathcal{G}$ is the item-set $\mathrm{init}(\mathcal{G})$ defined as

$$\mathrm{init}(\mathcal{G}) = \mathrm{closure}([\bullet\langle\texttt{START}\rangle\ [\backslash\texttt{EOF}] \to \langle\texttt{Start}\rangle])$$

This state expresses that a sentence can be recognized by recognizing a string of sort $\langle$`START`$\rangle$ followed by the special end of file character that indicates the end of strings.

The closure of a set of items adds all initial items to an item set for which the result symbol is predicted by one of the items in the set.

$$\mathcal{I} \subseteq \text{closure}(\mathcal{I}) \tag{Cl0}$$

$$\frac{[\alpha \bullet B\beta \to A] \in \text{closure}(\mathcal{I}),\ \gamma \to B \in \text{P}(\mathcal{G})}{[\bullet\gamma \to B] \in \text{closure}(\mathcal{I})} \tag{Cl1}$$

In the presence of priorities the closure is restricted to those items that do not cause a priority conflict.

$$\frac{\begin{array}{c}[\alpha \bullet B\beta \to A] \in \text{closure}(\mathcal{I}),\ \gamma \to B \in \text{P}(\mathcal{G}),\\ [\alpha[\gamma \to B]\beta \to A] \notin \text{conflicts}(\mathcal{G})\end{array}}{[\bullet\gamma \to B] \in \text{closure}(\mathcal{I})} \tag{Cl2}$$

For example, the item $[\bullet E{+}E \to E]$ is not added to the closure as a result of the item $[E + \bullet E \to E]$ if this production is left-associative, because $[E + [E + E \to E] \to E]$ is a conflict pattern

The parsing of a string starts with the parser in the initial state. Upon recognition of a symbol, either by reading a character or by completing a production, the parser can enter other states as prescribed by the transitions of the goto graph. The goto function maps an item-set to another item-set, given the symbol that has been recognized. The function 'goto' is defined by

$$\text{goto}(X, \mathcal{I}) = \text{closure}(\text{shift}(X, \mathcal{I}))$$

i.e., create a new item-set by shifting the $\bullet$ over the symbol X and produce the closure of the resulting item-set. In normal LR(0) parsing a shift with a symbol B creates an item-set containing all items of the previous set that have the $\bullet$ before a B symbol.

$$\frac{[\alpha \bullet B\beta \to A] \in \mathcal{I}}{[\alpha B \bullet \beta \to A] \in \text{shift}(B, \mathcal{I})} \tag{Sh1}$$

We refine the definition of shift to shifting with characters and shifting with productions. Shifting an item-set with a character or character-class is defined by the rule

$$\frac{[\alpha \bullet cc'\ \beta \to A] \in \mathcal{I},\ cc \subseteq cc'}{[\alpha\ cc' \bullet \beta \to A] \in \text{shift}(cc, \mathcal{I})} \tag{Sh2}$$

The character-class cc induces a shift of each item that predicts a character-class cc' that is a superset of cc.

Shifting with nonterminals is refined to shifting with complete productions. A shift is only successful if the production do not cause a priority conflict as a direct descendant at the position of the predicted symbol.

$$\frac{[\alpha \bullet B\beta \to A] \in \mathcal{I},\ [\alpha[\gamma \to B]\beta \to A] \notin \text{conflicts}(\mathcal{G})}{[\alpha B \bullet \beta \to A] \in \text{shift}(\gamma \to B, \mathcal{I})} \tag{Sh3}$$

For example, the production $E + E \to E$ cannot be used to shift the item $[E + \bullet E \to E]$ if this production is left-associative, because $[E + [E + E \to E] \to E]$ is a conflict. This restriction of the closure and goto functions guarantees that we can never enter a state where we have built a parse tree with a priority conflict.

3.5.5 Action Table

The action table declares the actions to be taken in each state. Given an item-set, the function 'actions' maps a character to the set of actions that the parser can take. If the set of actions is empty the parser has reached an erroneous state. If the set contains more than one action there is more than one way to proceed.

$$\frac{[\alpha \bullet cc\,\beta \to A] \in \mathcal{I},\ c \in cc}{\text{actions}(\mathcal{I}, c) \ni \texttt{shift}(\text{goto}([c], \mathcal{I}))} \tag{Shi}$$

$$\frac{[\alpha \bullet \to A] \in \mathcal{I},\ c \in \text{follow}(\alpha \to A)}{\text{actions}(\mathcal{I}, c) \ni \texttt{reduce}(\alpha \to A, |\alpha|)} \tag{Red}$$

$$\frac{[\langle\texttt{START}\rangle \bullet [\backslash\texttt{EOF}] \to A] \in \mathcal{I}}{\text{actions}(\mathcal{I}, \backslash\texttt{EOF}) \ni \texttt{accept}} \tag{Acc}$$

Note that $\texttt{shift}(\mathcal{I})$ denotes the shift *action* to state $\mathcal{I}$, whereas $\text{shift}(X, \mathcal{I})$ is the application of the shift *function* defined above.

The following proposition states that the actions and goto functions defined above constitute a correct shift-reduce parser.

Proposition 3.5.1 (Correctness) *Given the actions and goto functions for a grammar $\mathcal{G}$, we have that $(\text{init}(\mathcal{G}) \bullet w) \Rightarrow^* \texttt{accept}(t)$ iff $t \in \Pi(\mathcal{G})(w)$ and t contains no priority conflicts according to $\text{Pr}(\mathcal{G})$ and violates no follow restrictions in $\text{R}(\mathcal{G})$.*

3.5.6 Remarks

The transition rules for shift-reduce parsing are non-deterministic. If more than one action is possible in some configuration more than one transition is possible. If the actions function is deterministic, at most one transition path is possible for a string. Traditional parsing techniques only accept grammars that have a deterministic action function. In §3.8 we will discuss an efficient implementation for non-deterministic actions functions.

The rules for parser generation above ignore reject productions, i.e., they are treated just like other productions. In §3.7 we will discuss how reject productions can be interpreted by means of a filter on parse forests. In §3.8 we will discuss how reject productions can be interpreted during parsing by means of an adaptation of the GLR algorithm. For this purpose, an item-set $\mathcal{I}$ is marked as *rejectable* if it can be reached using a reject production, i.e., $\mathcal{I}$ is rejectable, if there is an $\mathcal{I}'$ such that $\text{goto}(\alpha \to A, \mathcal{I}') = \mathcal{I}$ and $\alpha \to A$ is a reject production.

3.6 Automatic Lexical Disambiguation

In §3.4 we discussed the specification of lexical disambiguation by means of follow restrictions and reject productions. Although this is an effective way to express lexical disambiguation, it is rather tedious to write down the rules. Therefore, it would be desirable to derive lexical disambiguation rules automatically from the other grammar rules such that the grammar is disambiguated according to the longest match and prefer literals criteria. Here we discuss some possibilities. The perfect rules for longest match disambiguation have not been found yet. It is a question whether this is possible at all, since it is undecidable whether a context-free grammar is ambiguous.

3.6.1 Prefer Literals

The prefer literals disambiguation rule can be expressed by generating reject productions according to the following rule:

$$\frac{\texttt{"}c_1 \ldots c_n\texttt{"} \in L(\langle A\texttt{-LEX}\rangle)}{\texttt{"}c_1 \ldots c_n\texttt{"} \to \langle A\texttt{-LEX}\rangle \ \{\texttt{reject}\} \in \mathrm{P}(\mathcal{G})}$$

i.e., if the literal is a lexical phrase of sort $\langle A\texttt{-LEX}\rangle$—there is an overlap—the reject rule is added to the grammar. This implements the reserved keywords rule. The only (implementation) problem is that a parser is needed to recognize the literals as lexicals. This can be solved by first generating a parser for the grammar without reject rules and using that parser to determine overlap between literals in the grammar and lexical categories. Reject rules can then be added to the grammar accordingly and a new parser can be generated for the extended grammar.

State Explosion A problem with reject productions to exclude keywords as lexicals is that it can add many items to item-sets. For instance, if a language contains 200 keywords that match with the identifiers of the language, each item-set containing an item $[\alpha \bullet \langle \texttt{Id-LEX}\rangle \beta \to A]$ would be expanded with 200 items $[\bullet\texttt{"}c_1 \ldots c_n\texttt{"} \to \langle \texttt{Id-LEX}\rangle \ \{\texttt{reject}\}]$ and 200 items $[\bullet[c_1] \ldots [c_n] \to \texttt{"}c_1 \ldots c_n\texttt{"}]$ along with many extra transitions. To prevent this expansion, we define the rejection of literals in an indirect way, as follows:

$$\frac{\texttt{"}c_1 \ldots c_n\texttt{"} \in L(\langle A\texttt{-LEX}\rangle)}{\begin{array}{c}() \ \langle A\texttt{-LIT}\rangle \to \langle A\texttt{-LEX}\rangle \ \{\texttt{reject}\} \in \mathrm{P}(\mathcal{G}) \\ \texttt{"}c_1 \ldots c_n\texttt{"} \to \langle A\texttt{-LIT}\rangle \in \mathrm{P}(\mathcal{G})\end{array}}$$

where the symbol () denotes the empty phrase, i.e., there is a production $\to ()$. The sort $\langle A\texttt{-LIT}\rangle$ is used to collect all literals to be rejected from $\langle A\texttt{-LEX}\rangle$. The production $() \ \langle A\texttt{-LIT}\rangle \to \langle A\texttt{-CF}\rangle \ \{\texttt{reject}\}$ defines the rejection for all literals at once. The effect of the empty symbol () in the second production is that only the item $[\bullet() \ \langle A\texttt{-LIT}\rangle \to \langle A\texttt{-LEX}\rangle]$ is added when $\langle A\texttt{-LEX}\rangle$ is predicted. This will cause a reduction with the production $\to ()$ to an item-set where $\langle A\texttt{-LIT}\rangle$ is predicted. This item-set is only computed once and is reused for all other item-sets that predict $\langle A\text{-}\mathtt{LEX}\rangle$. It is the initial state of a finite automaton for the matching of literals.

As an example, consider how the prefer literals rule for our functional language example is expressed using this modified rule:

```
syntax
  () <Var-LIT> -> <Var-LEX> {reject}
  "let"        -> <Var-LIT>
  "in"         -> <Var-LIT>
```

Local Exclusion An alternative for the expression of the prefer literals rule is the rule

$$\frac{\{[\bullet\langle A\text{-}\mathtt{LEX}\rangle \to \langle A\text{-}\mathtt{CF}\rangle],\ [\bullet[c_1]\ldots[c_n] \to "c_1\ldots c_n"]\} \subseteq \mathrm{closure}(\mathcal{I})}{[\bullet "c_1\ldots c_n" \to \langle A\text{-}\mathtt{CF}\rangle\ \{\mathtt{reject}\}] \in \mathrm{closure}(\mathcal{I})}$$

that locally forbids predicted literals as lexicals by extending the parser generator. This does not implement the reserved keywords rule in the sense of forbidding the use of a keyword as a lexical in all positions. Only when a lexical and a literal can appear in the same place, the literal is preferred. Therefore, it might still lead to ambiguities.

3.6.2 Longest Match

It is less straightforward to find a general rule to express 'longest match' using follow restrictions. An attempt is the rule

$$\frac{\langle B\text{-}\mathtt{LEX}\rangle \in \mathrm{follow}_{\mathrm{s}}(\langle A\text{-}\mathtt{LEX}\rangle \to \langle A\text{-}\mathtt{CF}\rangle)}{\langle A\text{-}\mathtt{LEX}\rangle \not\to \mathrm{first}(\langle B\text{-}\mathtt{LEX}\rangle) \cap \mathrm{last}(\langle A\text{-}\mathtt{LEX}\rangle) \in \mathrm{R}(\mathcal{G})} \quad \text{(FR)}$$

This restricts the follow set of $\langle A\text{-}\mathtt{LEX}\rangle$ by excluding the elements of the first set of $\langle B\text{-}\mathtt{LEX}\rangle$ that can also be used at the end of $\langle A\text{-}\mathtt{LEX}\rangle$ for those $\langle B\text{-}\mathtt{LEX}\rangle$s that can follow the injection $\langle A\text{-}\mathtt{LEX}\rangle \to \langle A\text{-}\mathtt{CF}\rangle$. Here $\mathrm{follow}_{\mathrm{s}}$ is the extension of the follow function to produce all symbols that can follow a production.

This rule is adequate in many cases. Consider for instance the functional expression grammar. The follow restriction for `Var` in §3.4.2 is derived exactly using this rule. However, the rule is not general enough. One counter example is the following grammar of expressions with single character variables and implicit multiplication operator. This describes mathematical expressions such as xy that denotes the multiplication of x and y.

```
lexical syntax
  [a-z]     -> Var
  [\ \t\n] -> LAYOUT
context-free syntax
  Var       -> Exp
  Exp Exp  -> Exp {left}
```

Rule (FR) would forbid `xy` as an expression forcing the use of whitespace, i.e., `x y`. Although this example shows that rule (FR) is unsound if considered as an analytic rule, one could also consider it as a normative rule forcing a clearer style of language definition.

Rule (FR) generates follow restrictions for lexicals. We also need restrictions for literals overlapping with lexicals. For instance, the restrictions

```
lexical restrictions
  "let" "in" -/- [a-z]
```

forbids the interpretation of `letter` as the literal `let` and the variable `ter`. The following rule adds restrictions to prevent this overlap.

$$\frac{\langle A\texttt{-LEX}\rangle \in \text{follow}(\texttt{"}c_1 \ldots c_n\texttt{"}), \quad c \in \text{first}(\langle A\texttt{-LEX}\rangle),\ \texttt{"}c_1 \ldots c_n c\texttt{"} \in L(\langle B\texttt{-LEX}\rangle)}{\texttt{"}c_1 \ldots c_n\texttt{"} \not\to [c] \in \mathrm{R}(\mathcal{G})}$$

If the literal $L = \texttt{"}c_1 \ldots c_n\texttt{"}$ followed by some character c from the first set of a lexical $\langle A\texttt{-LEX}\rangle$ that is a member of the follow set of L can form a lexical $\langle B\texttt{-LEX}\rangle$, there is a longer match than the literal L. Therefore, c is restricted from the follow set of the literal.

This rule is stronger than the longest match filter we formulated before. It can forbid sentences that have a single unambiguous interpretation. For instance, consider the string `let x = 1 int`. Here `int` is forced to be read as a variable and not as the juxtaposition of the literal `in` and the variable `t`.

It is clear that these rules are not the final word about fully automatic lexical disambiguation. Further research is needed to decide what is sufficient.

3.7 Reject Productions

In §3.4 we introduced reject productions to express 'prefer literals' lexical disambiguation. The parser generator discussed in §3.5 treats reject productions as normal productions. This will cause ambiguous parses for those cases where a normal production and a reject production overlap. In this section we first define the semantics of context-free grammars with reject productions, then we investigate several properties of such grammars including an interpretation of rejects to solve such ambiguities. [The author thanks Jan van Eijck and Annius Groenink for the email discussion that led to the results in this section.]

3.7.1 Semantics

The semantics of reject productions is obtained by refining the inductive definition of parse trees from §3.4. The inductive rule (Prod) is restricted to exclude the construction of parse trees that have a yield that could be obtained via a reject production.

A context-free grammar $\mathcal{G}$ with reject productions generates a family of sets of *parse trees* $\mathcal{T}_{\mathrm{r}}(\mathcal{G}) = (\mathcal{T}_{\mathrm{r}}(\mathcal{G})(X) \mid X \in \mathrm{Syms}(\mathcal{G}))$, which contains the minimal sets $\mathcal{T}_{\mathrm{r}}(\mathcal{G})(X)$ such that

$$\frac{c \in cc}{c \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(cc)} \tag{CharR}$$

$$\frac{\begin{array}{c} A_1 \ldots A_n \to A \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(A_1),\ \ldots,\ t_n \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(A_n), \\ \neg\exists \beta \to A\ \{\texttt{reject}\} \in \mathrm{P}(\mathcal{G}), t_\beta \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(\beta) : \mathrm{yield}(t_\beta) = \mathrm{yield}(t_1 \ldots t_n) \end{array}}{[t_1 \ldots t_n \to A] \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(A)} \tag{ProdR}$$

The second condition of (ProdR) excludes from $\mathcal{T}_{\mathrm{r}}(\mathcal{G})(A)$ those trees for which an A tree with the same yield could be built using a reject production at its root. This second condition is the only difference with the definition of $\mathcal{T}(\mathcal{G})$, i.e., we have $\mathcal{T}_{\mathrm{r}}(\mathcal{G}) \subseteq \mathcal{T}(\mathcal{G})$. Note that only trees in $\mathcal{T}_{\mathrm{r}}(\mathcal{G})(\beta)$ are excluded. That is, if there are nested reject productions such that some tree in $\mathcal{T}(\mathcal{G})(\beta)$ is rejected and thus not part of $\mathcal{T}_{\mathrm{r}}(\mathcal{G})(\beta)$, then it is not used to exclude trees using $\beta \to A$ {reject}.

Unfortunately, this definition is inconsistent for grammars with a cycle containing a reject production. For instance, consider the grammar

```
syntax
  [a] -> A
  A   -> B
  B   -> A {reject}
```

and consider whether the string a is a member of the language of this grammar: if $[a \to A] \in \mathcal{T}_{\mathrm{r}}(A)$, then $[[a \to A] \to B] \in \mathcal{T}_{\mathrm{r}}(B)$ and hence $[a \to A] \notin \mathcal{T}_{\mathrm{r}}(A)$. Conversely, if $[a \to A] \notin \mathcal{T}_{\mathrm{r}}(A)$, then $[[a \to A] \to B] \notin \mathcal{T}_{\mathrm{r}}(B)$ and hence $[a \to A] \in \mathcal{T}_{\mathrm{r}}(A)$. For this reason, we restrict the class of grammars that we want to consider to grammars that do not contain a cycle (disregarding the rejects) for which one of the transitions is via a reject production.

3.7.2 Expressive Power

In the rest of this section we explore some of the properties of reject productions.

$a^n b^n c^n$ First of all context-free grammars with reject productions can be used to describe some non-context-free languages. Consider for example, the language $a^n b^n c^n$ with $n \geq 0$, which is a standard example of a non-context-free language. The following grammar, due to Van Eijck (1997), defines this language using reject productions. The first four productions define the language $a^* b^* c^*$. The next four productions define the sorts D and E denoting, respectively, $a^n b^n$ and $b^n c^n$. The last four productions exclude from sort S all strings for which one of the pairs $x^n y^m$ have unequal numbers of xs and ys.

```
A* B* C* -> S              -> D       D  B+ C* -> S {reject}
"a"      -> A        A D B -> D       A+ D  C* -> S {reject}
"b"      -> B              -> E       A* B+ E  -> S {reject}
"c"      -> C        B E C -> E       A* E  C+ -> S {reject}
```

Difference Given a context-free grammar defining sorts A and B we can define the difference of the languages of these sorts by adding the following productions.

```
A  -> AminB
B  -> AminB {reject}
```

The first adds all A trees to AminB, the second excludes from this all A trees that match with a B tree.

Intersection Extending this result, we can express the intersection between sorts A and B by adding two new sorts AminB and AandB and by adding the following productions:

```
A -> AminB                  A     -> AandB
B -> AminB {reject}         AminB -> AandB {reject}
```

This defines AminB as the difference A − B, and AandB as the difference A − (A − B), i.e., the intersection of A and B.

We can generalize the results above. Given two context-free languages, we can express the difference and intersection of those languages using context-free grammars with reject productions. Take the union of the context-free grammars for the two languages, after renaming symbols to prevent interference. Then add productions for the sorts to be intersected as explained above.

Weak Complement If we are only interested in the strings that can be generated from a grammar (and not in their structure), the complement of a the language generated by sort A is defined by extending a grammar with the following rules:

```
~[]* -> NotA
A    -> NotA {reject}
```

The first production defines the complement of A as a string of arbitrary characters. The complement ~[] of the empty character class is the character class with all characters. The second production excludes from this language all strings in the language of A. Using this complement we can of course also express the weak intersection of two sorts.

Decidable We have seen that context-free grammars with reject productions are very expressive. It is now appropriate to ask whether it is even decidable whether a string is in the language of such a grammar. The following theorem states that this is indeed the case. The proof uses the notion of a parse forest that will be discussed in the next section. For the proof of the theorem we need the following proposition about generalized-LR parsers.

Proposition 3.7.1 *Let $\mathcal{G}$ be a context-free grammar. If $t_1, t_2 \in \mathcal{T}(\mathcal{G})(A)$ and $\mathrm{yield}(t_1) = \mathrm{yield}(t_2) = w$, then a GLR parse of w will result in an ambiguity node with t_1 and t_2 as possibilities.*

Theorem 3.7.2 *The parsing problem for context-free grammars with reject productions (without rejects in cycles) is decidable.*

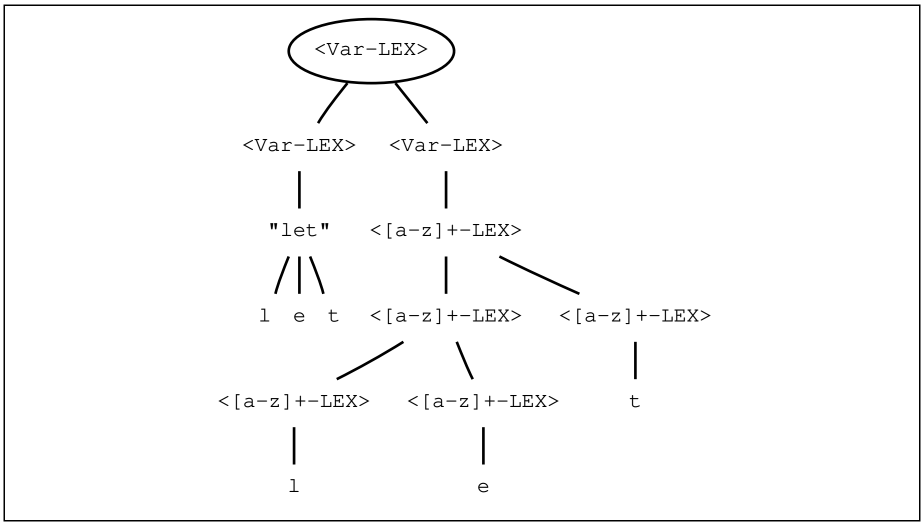


Figure 3.7: Ambiguity node caused by overlap between syntax for `<Var-LEX>` and reject production `"let" -> <Var-LEX>`

Proof. Given a context-free grammar $\mathcal{G}$ with reject productions (without rejects in cycles), construct a generalized-LR parser for $\mathcal{G}$ ignoring the reject annotations. The result is a parser for a possibly ambiguous context-free grammar. Now, given a string, parse it with this parser. If parsing fails, the string is also not in the language of the grammar with reject productions. Otherwise, the result of parsing is a parse forest. Since cycles do not contain rejects, these can be removed from the forest.

Now, if a tree $t = [t_1 \ldots t_n \to A]$ should be rejected according to the second condition of rule (ProdR), there is a reject production $\beta \to A$ and trees t_β such that $\text{yield}(t_\beta) = \text{yield}(t_1 \ldots t_n)$. But then, $\text{yield}([t_\beta \to A]) = \text{yield}([t_1 \ldots t_n \to A])$ and hence, according to the proposition above, the parse forest contains an ambiguity node on top of t also containing $[t_\beta \to A]$ as possibility.

Reject productions are now interpreted by traversing the forest, in a bottom-up manner, marking tree nodes according to the following rules: (1) Leafs are not marked. (2) A reduction node is marked if any of its direct descendants is marked. (3) An ambiguity node is marked if either all its direct descendants are marked, or if it contains an unmarked tree with as root label a reject production. Since the parse forest is finite, this procedure terminates.

If the root of the parse forest is marked after this procedure, the string is not accepted by the grammar, otherwise it is accepted and the forest without marked nodes represents all parse trees for the string. □

The tree in Figure 3.7 illustrates the proof. The overlap between the literal `"let"` and the syntax for variables causes an ambiguity. The ambiguity node is marked because `"let" -> <Var-LEX>` is a reject production. Therefore, the interpretation of `let` as a variable is dismissed.

This shows that we can construct a complete implementation of parsers for grammars with reject productions. In the next section we will discuss how reject productions can be interpreted *during* parsing to influence parse decisions to prevent trees with rejected subtrees from being built at all.

Expressive Power From the above we can conclude that context-free grammars with reject productions are stronger than pure context-free grammars, but have a decidable parsing problem. This gives a lower bound and upper bound for the expressive power of the formalism, but it is an open question what class of languages is described by context-free grammars with reject productions.

Regular Rejects We introduced reject productions in order to express the prefer literals rule. This means that only a regular language is excluded from a context-free one. This gives us the guarantee that the resulting language is still context-free. We could exploit this property and restrict the formalism to such regular reject productions and implement these by means of a grammar transformation. However, such a grammar transformation would probably yield large grammars. Furthermore, our implementation gives a general way to express the prefer literals rule and it allows the expression of other interesting grammars that have not been in the reach of declarative specification. This feature can give rise to as yet unforeseen applications.

3.8 Generalized-LR Parsing

In §3.5 we have defined the generation of shift-reduce parsers from context-free grammars with priority declarations and follow restrictions. If the actions function derived from a grammar is deterministic then the shift-reduce parser is also deterministic and can be implemented in a standard way.

However, since we do not restrict the class of grammars, it is not guaranteed that the actions function is deterministic. This can have two causes: (1) The lookahead needed for the grammar is more than provided by the parser generator. (2) The grammar is ambiguous. In the case of scannerless parsing we will frequently see grammars for which unbounded lookahead is needed. This entails that no variant of the LR parser generation algorithms will produce a deterministic actions function. Therefore, we need a non-deterministic implementation of the shift-reduce parsing algorithm. When a configuration is reached where more than one action is possible, all possibilities should be tried. In case of unbounded lookahead only one of the possible transitions leads to an accepting configuration. In case of an ambiguous string, multiple accepting configurations will be reached giving all possible parse trees for the string.

The advantage of such a non-deterministic approach is, first of all, the unbounded lookahead that it provides. Furthermore, a parser producing all parse trees for an ambiguous string can be used as a front-end for a disambiguation filter that selects the correct tree according to some disambiguation method. Finally, it is undecidable whether a grammar is ambiguous or has lookahead problems. Having a parser that yields all possible parses can help in detecting the ambiguities and resolve them in a much easier way than by inspecting conflicts in a parse table.

A naive way to implement such a non-deterministic parsing algorithm is to copy the entire configuration at each point where two or more actions are possible and to continue parsing with each those configurations. This will not be very efficient because of the memory requirements and because it will not reuse parses for substrings that are the same in two forked off configurations. Generalized-LR parsing is an efficient implementation of non-deterministic shift-reduce parsing. A GLR parser deals with conflicts in the parse table by splitting the parser into as many parsers as there are conflicts. If the conflict was due to a lack of lookahead, some of the parsers will not succeed in parsing the sentence and will die. If several parsers succeed in parsing, the grammar was ambiguous. In that case parse trees for all possible parses are built.

Generalized-LR parsing was developed for natural language processing by Tomita (1985). It is a specialization of the more general framework of Lang (1974) (later also described in Billot and Lang (1989)) for creating generalized parsers. The algorithm was improved by Rekers (1992) and applied to parsing of programming languages. The feasibility of GLR parsing for parsing of programming languages has been shown by the experience with GLR in the ASF+SDF Meta-Environment (Klint, 1993). More experience with GLR parsing of programming languages using an adaptation of Reker's algorithm is reported by Wagner and Graham (1997).

Besides the non-determinism in the parse table, we also need to interpret the reject productions in the grammar. In the previous section we showed how reject productions can be interpreted as a disambiguation filter after parsing. But we would rather interpret them earlier. In this section we explain GLR parsing and present an adaptation of the algorithm to interpret reject productions during parsing.

3.8.1 Parse Forest

A generalized parser deals with ambiguous grammars by producing all possible parse trees for an ambiguous string. In GLR parsing the possible parse trees are represented by means of a parse forest. This is a compact representation of a set of parse trees. A parse tree is constructed using application and ambiguity nodes. An application node represents the application of a production to a list of subtrees. An ambiguity node represents a set of possible parse trees for a (sub)string. By packing all trees for a substring into an ambiguity node, these parses can be shared in all trees for strings containing the substring.

For example, consider the following grammar of simple expressions with ambiguous addition and multiplication operator.

```
sorts Exp
syntax
  [a-z]         -> Exp
  Exp "+" Exp -> Exp
  Exp "*" Exp -> Exp
  Exp           -> <START>
```

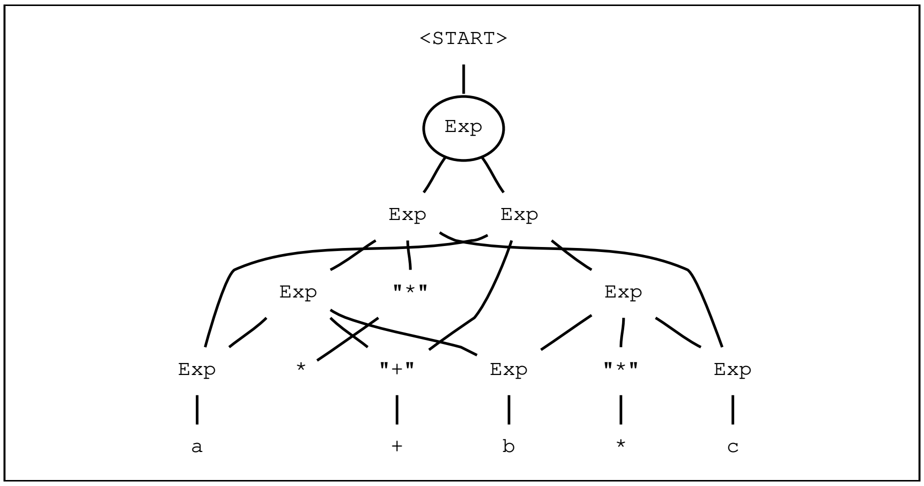


Figure 3.8: Parse forest with sharing for ambiguous string `a+b*c`.

To keep the example small, layout is not allowed between the tokens. The parse forest for the ambiguous string `a+b*c` is shown in Figure 3.8. The ellipse represents an ambiguity node. Observe that various subtrees are shared in the forest.

3.8.2 Graph Structured Stack

A GLR parser deals with conflicts in the parse table by maintaining a number of stacks in parallel. Each time a parse stack leads to n conflicting actions, n new stacks are created that continue the parse with those actions. These stacks are not copies of the old stack. The new top nodes have pointers to the old stack. If in a later stage two stacks get into the same state, the stacks are merged again. In this manner a graph structured stack is built in which parses for ambiguous substrings are shared.

A graph structured stack node consists of a state number and a list of links. Each link contains a reference to a node in the parse forest and a reference to the previous stack.

As an example of the working of a GLR parser, consider the sequence of stack configurations during parsing the string `a+b*c` in Figure 3.9. This is the parse that created the parse forest in Figure 3.8. The figure shows the stacks during each cycle of the parsing algorithm. After shifting a character, all possible reductions are performed and then the next character is shifted. The trees pointed to by the stack links are abbreviated by their yield using square brackets to show the structure. The symbol after the colon denotes the main type of the tree at the link. We consider the configurations one by one.

(a) The initial stack with state 0 is created. The character `a` is shifted.

(+) The character `a` reduces to an expression using the production `[a-z] ->`

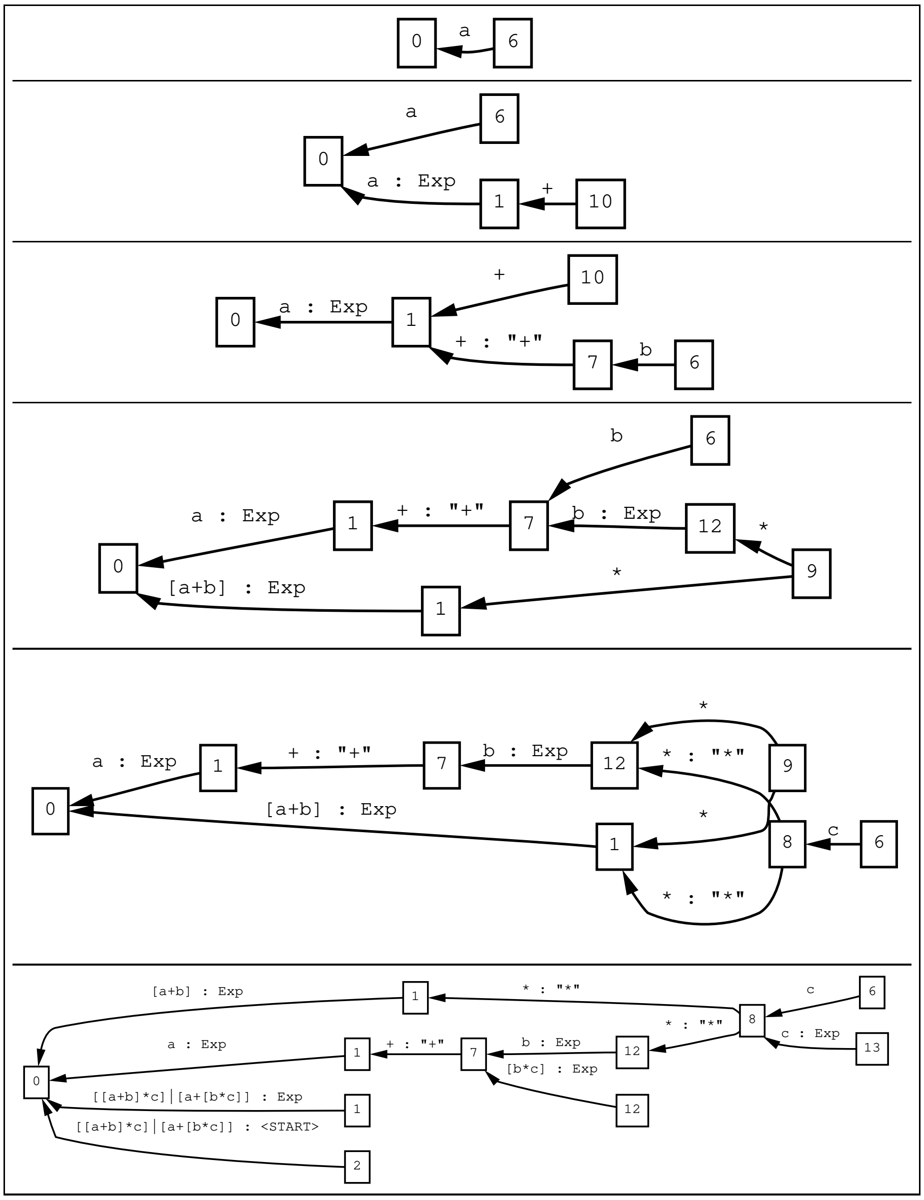


Figure 3.9: Parse configurations for the parse of string `a+b*c`.

`Exp`. The symbol `+` is shifted.

(b) The character `+` reduces to the literal `"+"`. The character `b` is shifted.

(*) The character `b` reduces to an expression. The sequence `[a+b]` reduces to an expression, resulting in a link from state 1 to state 0. From states 1 and 12 a shift can be performed with the next character `*`. Because both shifts lead to a stack with state 9, a single stack is created that has links to the two stacks.

(c) The character `*` reduces to the literal `"*"`. The character `c` is shifted.

(`\EOF`) The character `c` reduces to an expression. Now there are two possible reductions from the stack with state 13. First reduce `[b*c]` and then reduce `[a+[b*c]]`, or reduce `[[a+b]*c]`. Both reductions result in the creation of a stack with state 1 with a link to the initial stack. These stacks are shared and an ambiguity node is created that represents the two possible parse trees. At this point the entire string has been read and the next symbol is `\EOF`. Therefore, the expression is reduced to ⟨`START`⟩ and the string is accepted. The stack with state 2 is the accepting stack and the tree pointed to by its link is the parse tree for the entire string.

3.8.3 Reject Reductions

In §3.4 disambiguation with reject productions was introduced in order to express the prefer literals rule. In §3.7 we outlined a procedure for interpreting reject productions after parsing by pruning the parse forest. We would rather interpret reject productions during parsing to prevent trees containing reject productions from being built.

To understand how this can be achieved, recall the parse forest in Figure 3.7 that shows the ambiguity that is created when parsing the substring `let` in the functional expression grammar defined in §3.4.1. It can be interpreted using the lexical productions for variables or using the reject production `"let" -> <Var-LEX>` {`reject`}. Pruning this forest causes the ambiguity node to be eliminated from the parse forest, thereby rejecting the reading of `let` as a variable.

The parse configurations for this parse in Figure 3.10 show how this ambiguity is created. In the first three configurations the letters `l`, `e` and `t` are read. The fourth configuration is the interesting one. There are three parses for the substring `let`: as a variable constructed with `<[a-z]+-LEX>`, as the literal `"let"` of the reject production `"let" -> <Var-LEX>`, and as the literal `"let"` as part of the let construct. The reduction of the literal results in a stack with state 12. The reduction of the lexical and the reject rule lead to a merged stack with state 10 from where another reduction first leads to a stack with state 9 and then leads to a term and a stack with state 14. From the states 9, 12 and 14 parsing continues with a shift of the space character ␣ (32) to state 6.

The idea for the implementation of the reject rule is to forbid further actions with a state that has been reached using a reject reduction. The link that is created when reducing with a reject production is marked as rejected. If all links of a stack are marked as rejected all shifts and reductions from that state are forbidden.

In the last configuration of Figure 3.10 this would entail that the link from the stack with state 10 to the stack with state 4 is rejected. Therefore, the reduction

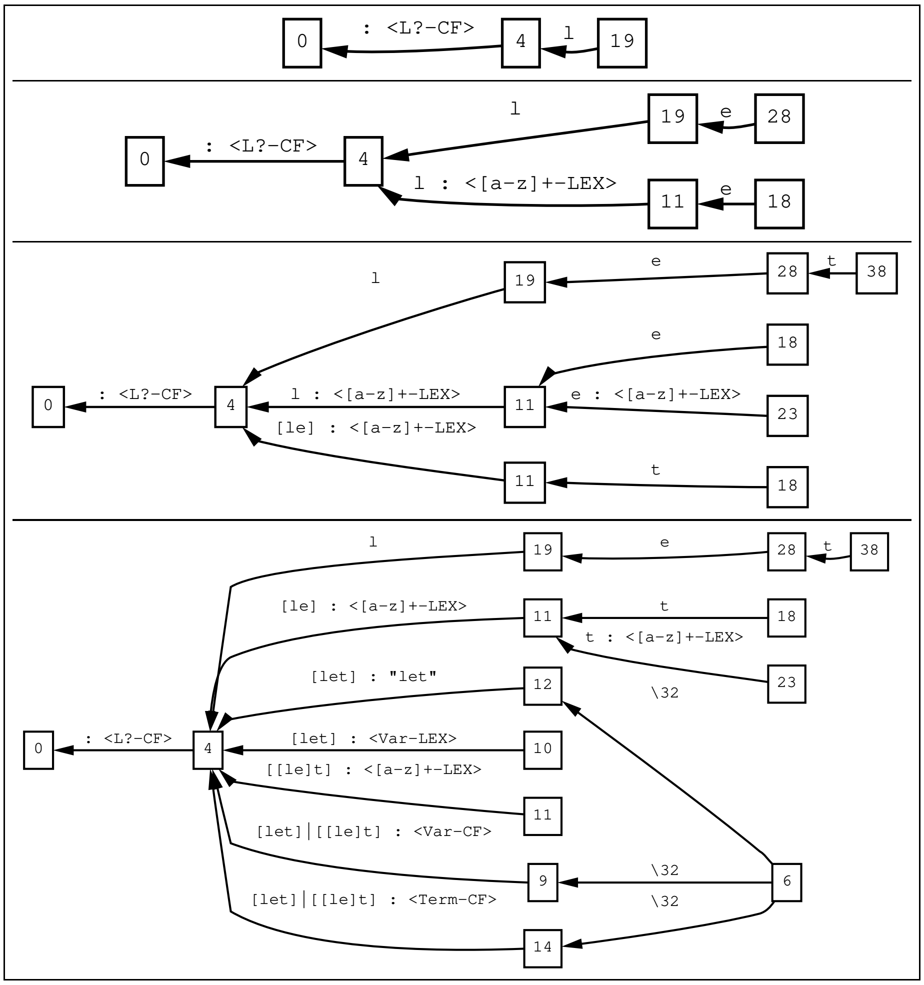


Figure 3.10: Parse of `let␣x...` when reject is ignored.

to state 14 and the shift to state 6 would be forbidden. This is exactly what happens in the parse shown in Figure 3.11. The dotted link is rejected and no actions are performed from its stack. The parse of `let` as variable is preempted. In the next configuration parsing continues only with the stack with state 12, corresponding to a parse of `let` as a literal.

3.8.4 The Algorithm

Below the complete SGLR algorithm is presented. The differences with the GLR algorithm of Rekers (1992) are the use of productions in the goto function and the handling of reject reductions. Furthermore, the parser does not make use of a scanner, but reads characters from a file or string. This could of course

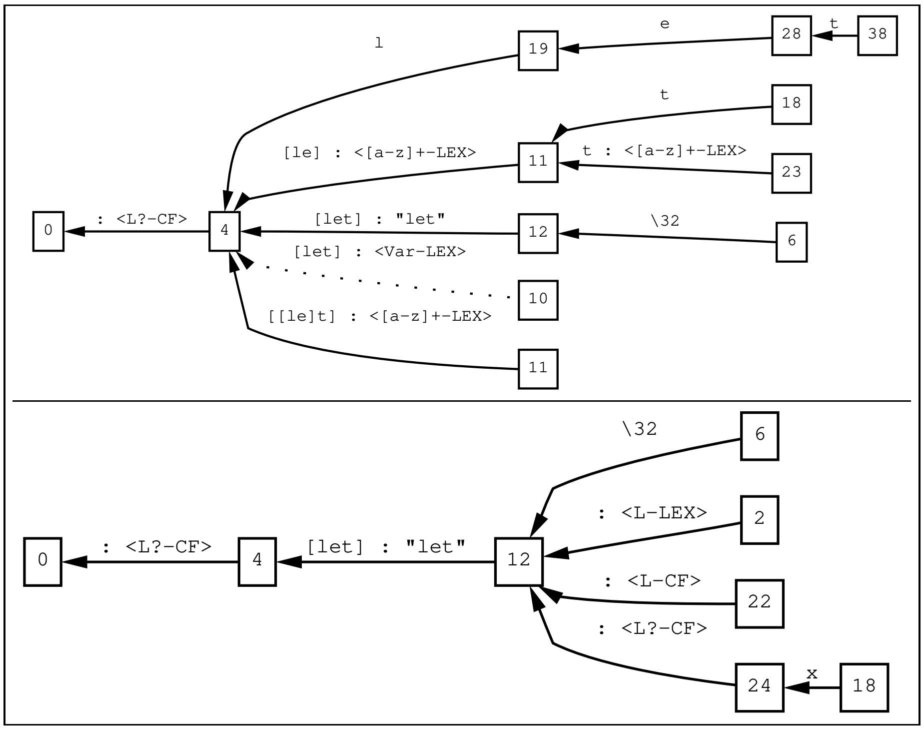


Figure 3.11: Parse of `let␣x...` with reject production that forbids `let` as a variable.

be a stream of token codes and does not make a difference to the algorithm. As we discussed in the §3.5 character classes are handled in the parse table and are thus transparent to the parser.

Algorithm 3.8.1 (SGLR) Given the parse table *table* for some grammar, parse the string of characters in *file*. If the string is a sentence in the language described by the grammar, return the parse forest for the string, and an error message otherwise.

Parse The function parse reads the characters from a file and returns a parse tree if the text is syntactically correct, an error message otherwise. The list of active stacks is initialized to contain a single stack with the initial state of the parse table as its state. For each character in the input, the parser handles all actions for each active stack. The shifts for each stack are stored and performed by the shifter after all possible reductions have been performed. When all characters have been read or when no more stacks are alive, parsing terminates. If parsing succeeded, the accepting stack has a direct link to the initial state. This link has a reference to the parse forest with all possible parse trees for the entire string. If parsing failed an error term is returned.

PARSE(*table*, *file*)
  global *accepting-stack* := $\emptyset$
  global *active-stacks* := {new stack with state init(*table*)}
  do
    global *current-token* := get-next-char(*file*)
    PARSE-CHARACTER()
    SHIFTER()
  while *current-token* $\neq$ `\EOF` $\wedge$ *active-stacks* $\neq \emptyset$
  if *accepting-stack* contains a link to the initial stack with tree t **then**
    return t
  else
    return `parse-error`

Parse Character The list of active stacks is moved to the list of stacks of the actor that performs the actions for a stack unless the stack is rejected. The list of stacks for the actor is extended when reductions are performed. If actions for newly added stacks are performed before all links to it have been created, a stack that becomes rejected might escape. Therefore, new stacks are added to *for-actor-delayed* if they are rejectable and are only considered when all stacks on *for-actor* are exhausted. Then stacks are taken from the delayed list in order of priority. The operation 'pop' removes the stack with the highest priority from a list of stacks.

PARSE-CHARACTER()
  global *for-actor* := *active-stacks*
  global *for-actor-delayed* := $\emptyset$
  global *for-shifter* := $\emptyset$
  while *for-actor* $\neq \emptyset \wedge$ *for-actor-delayed* $\neq \emptyset$ **do**
    if *for-actor* $= \emptyset$ **then**
      for-actor := {pop(*for-actor-delayed*)}
    for each stack *st* $\in$ *for-actor* **do**
      if $\neg$ all links of stack *st* rejected **then**
        ACTOR(*st*)

Actor Handle the actions for stack *st* and the current token. A reduce action is immediately handled. Shift actions are saved on *for-shifter* for handling if after all reductions have been performed. An accept action results in saving the current stack as the accepting stack. An error action is ignored because the current stack can be a wrong attempt while other stacks are still alive. The entire parse fails if all stacks lead to error actions. This will become apparent after shifting because no more active stacks will be alive.

ACTOR(*st*)
  for each action $a \in$ actions(s, *current-token*) **do**
    case a **of**
      `shift`(s) $\Rightarrow$ *for-shifter* := $\{\langle st, s\rangle\} \cup$ *for-shifter*
      `reduce`($\alpha \rightarrow A$) $\Rightarrow$ DO-REDUCTIONS(st, $\alpha \rightarrow A$)
      `accept` $\Rightarrow$ *accepting-stack* := *st*

Reductions Function *do-reductions* performs a reduction for stack *st* with production $\alpha \to A$. For each path of length $|\alpha|$ following the links from *st* to some stack st_0 the trees along the path are collected and the reducer is called to handle the reduction.

```
DO-REDUCTIONS(st, α → A)
    for each path from stack st to stack st₀ of length |α| do
        kids := the trees of the links which form the path from st to st₀
        REDUCER(st₀, goto(state(st₀), α → A), α → A, kids)
```

Reducer Given a stack *st*, a state *s*, a production $\alpha \to A$ and a list of trees *kids*, the reducer creates the application node for the production and the list of direct descendants *kids* and creates a new stack with state *s* and a link to stack *st*. However, because there might already exist as stack with state *s*, the list of active stacks is searched. If there is no such stack a new stack is created (else branch) and added to the list of active stacks and the list of stacks for the actor. The new stack has state *s* and a link with a pointer to the newly created tree. If a stack with state *s* already exists and there is a direct link *nl* from st_1 to st_0, an ambiguity has been found. The tree *t* is added to the ambiguity node of the link. If there is no direct link, a new link is created from st_1 to st_0 with *t* as parse tree. Because this new link entails that new reductions from already inspected stacks might be possible, all active stacks are reconsidered. In all cases, the link that is created or extended is marked as rejected if the production is a reject production.

```
REDUCER(st, s, α → A, kids)
    t := application of α → A to kids
    if ∃st₁ ∈ active-stacks : state(st₁) = s
        if ∃ a direct link nl from st₁ to st₀ then
            add t to the possibilities of the ambiguity node at tree(nl)
            if α → A is a reject production then mark link nl as rejected
        else
            add a link nl from st₁ to st₀ with tree t
            if α → A is a reject production then mark link nl as rejected
            for each st₂ ∈ active-stacks
                such that ¬ all links of st₂ rejected
                  ∧ st₂ ∉ for-actor ∧ st₂ ∉ for-actor-delayed do
                for each reduce(α → A) ∈ actions(state(st₂), current-token)
                do
                    DO-LIMITED-REDUCTIONS(st₂, α → A, nl)
    else
        st₁ := new stack with state s
        add a link nl from st₁ to st₀ with tree t
        active-stacks := {st₁} ∪ active-stacks
        if rejectable(state(st₁)) then
            for-actor-delayed := push(st₁, for-actor-delayed)
```

else
for-actor := $\{st_1\} \cup$ *for-actor-delayed*
if $\alpha \to A$ is a reject production **then** mark link nl as rejected

Limited Reductions The function *do-reductions* is used to do all reductions for some state and production that involve a path going trough link *nl*.

DO-LIMITED-REDUCTIONS(st, $\alpha \to A$, l)
for each path from stack st to stack st_0 of length $|\alpha|$ going through link l
do
kids := the trees of the links that form the path from st to st_0
REDUCER(st_0, goto(state(st_0), $\alpha \to A$), $\alpha \to A$, *kids*)

Shifter After all possible reductions have been performed, *for-shifter* contains a list of stacks that can do a shift. Only these stack make it into the next cycle of the parse. The list of active stacks is reinitialized to the empty list. For each stack st_0 in *for-shifter* a new stack is created with a link to st_0 with as tree the current token. That is, if a stack with state s was already created only a link from that stack to st_0 is created.

SHIFTER()
active-stacks := $\emptyset$
t := *current-token*
for each $\langle s, st_0\rangle \in$ *for-shifter* **do**
if $\exists st_1 \in$ *active-stacks*: state(st_1) = s **then**
add a link from st_1 to st_0 with tree t
else
st_1 := new stack with state s
add a link from st_1 to st_0 with tree t
active-stacks := $\{st_1\} \cup$ *active-stacks*
end

3.8.5 Remarks

The algorithm above does not actually mark stacks as rejected, but the link from a stack that is created with a reject production. Further action on a stack is forbidden if all links from that stacks are rejected. This is done because, in principle, there could be situations where two links are created from the same stack that are not merged (as is the case when the links are to the same stack) and only one is rejected. It is not clear whether such a situation can occur. But there is no proof of the contrary either.

The ordering on states that is assumed in the priority pop operation used in procedure PARSE-CHARACTER() is needed to ensure that nested reject productions are treated properly. For example, consider again a grammar extended with productions expressing the intersection of sorts `A` and `B`.

```
A -> AminB           A     -> AandB
B -> AminB {reject}  AminB -> AandB {reject}  AandB -> <Start>
```

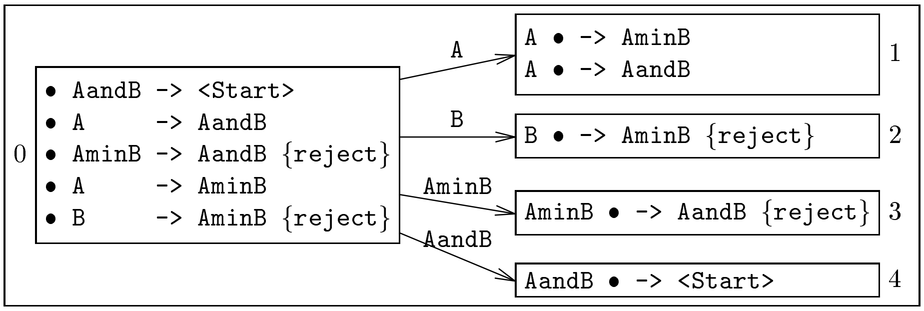


Figure 3.12: Goto graph for grammar with nested reject productions.

This gives rise to the goto graph in Figure 3.12. States 3 and 4 are rejectable because they can be reached with a reject production. When parsing a string that is in `A` and in `B`, state 2 is reached using the reduction for `B`. The next reduce action with the reject production `B -> AminB {reject}` leads a stack with state 3, which is rejected. No further action is taken from that stack. The reduction of `A -> AandB` leads to a stack with state 4 and then, correctly, to acceptance of the string.

Now consider the case where a string is in `A`, but not in `B`. Then there is no reduction to state 2 and hence state 3 is not rejected, but there is a reduction to state 3 using `A -> AminB` and a reduction to state 4 using `A -> AandB`. Now there are two rejectable stacks on the *for-actor-delayed* list. If the stack with state 3 is released first a reduction with `AminB -> AandB {reject}` occurs and the stack with state 4, which is still on *for-actor-delayed*, is rejected; and parsing fails as it should. However, if state 4 is released first, parsing succeeds before the stack with state 4 is rejected. It is clear that in this case state 3 has higher priority than state 4.

It is not clear how the ordering on states should be determined in general. It would seem that a state s_1 with productions that are reachable from the productions in a state s_2 has higher priority. This is only a guess, however, and should be worked out more carefully. For single, i.e., non-nested reject productions the ordering plays no role. Therefore, the implementation of exclusion by means of reject productions, of which prefer literals is a special case, is not dependent on finding an ordering on states.

3.9 Implementation

In the previous sections we have presented an approach to scannerless parsing. These techniques are implemented as part of the SDF2 tools. The tools have been used to construct parsers for a number of languages including SDF2 itself. Although no detailed data on the performance of the implementation are available at the time of this writing, a couple of preliminary observations can be made nonetheless.

Grammar Normalizer The syntax definition formalism SDF2 is completely specified in ASF+SDF. Part of the definition is the grammar normalizer discussed in §3.3. This specification has been compiled to an executable term rewriter, which has a reasonable performance. The literate specification of SDF2 and the normalization of syntax definitions is presented in Part II. The specification also defines the format of parse trees encoded in the ATerm format of Van den Brand *et al.* (1997a).

Parser Generator The parser generator described in §3.5 has been completely specified in ASF+SDF. The compiled specification of the parser generator is too inefficient. It is probably necessary to implement this component in an imperative language that allows direct access instead of lookup in lists.

There are several factors that make parser generation more difficult compared to normal SLR(1) parser generation for context-free grammars. There are more item-sets because of the productions for the lexical syntax. Extra productions are added because of the reject productions expressing the prefer literals rule, this increases the number of items in item-sets. The goto table contains a transition for each production instead of a transition for each nonterminal. The last factor can be reduced by sharing transitions to the same state.

Productions and item-sets are encoded by numbers. Character classes are important for reducing the size of the parse table. A set of actions that is shared by several characters is stored efficiently by means of a character class, i.e., 'actions' is a mapping from item-sets and *character classes* to sets of actions.

Parser The SGLR parsing algoritm has been implemented in C. The implementation makes use of the C implementation of ATerms (Van den Brand *et al.*, 1997a) to represent stacks and trees.

The parser includes visualization tools for parse forests and graph structured parse stacks that were used to produce the pictures in this chapter. The forest vizualization might be used as basis for an interactive disambiguation tool.

The C implementation of the SGLR parsing algorithm seems reasonably efficient, although sharing of trees can be improved. Output of parse trees is not optimal because sharing of subtrees is completely lost when writing out a parse forest in a linear term format. This can solved by using a linear encoding of graphs such as the graph exchange language GEL of Kamperman (1994). Furthermore, a mark-scan garbage collector for stack and tree nodes is used. This entails that all stack and tree nodes are visited on a collect, which is too expensive, since a large amount of the heap will not change status. A reference count garbage collector should make a difference.

Complexity of Lexical Analysis We have performed a few experiments to get an idea of the complexity of lexical analysis with scannerless generalized-LR parsers. The experiments were based on the simple expression grammar in §3.2. The experiments that were performed were of the form: (a) Parsing a single identifier of increasing length (up to 425KB). (b) Parsing an expression consisting of ten additions with identifier arguments of increasing length (up to 325KB). (c) Parsing an expression consisting of an increasing number of additions (up to 16K arguments with length 490KB).

For all these experiments we saw an almost linear behaviour for small files deteriorating to square behaviour for the large files. However, when garbage collection was turned off, this behaviour changed into linear for all experiments. This confirms the observation about the inappropriateness of the garbage collection algorithm. It also confirms the idea that lexical analysis will behave linearly for simple, i.e., regular lexical syntax. The prototype implementation should be further optimized before its performance can meaningfully be compared to scanner/parser combinations such as LEX/YACC. Nonetheless, these experiments show the feasibility of the scannerless generlized-LR parsing approach.

3.10 Related Work

The syntax definition formalism SDF2 is formally specified in Part II. The specification in ASF+SDF comprises the syntax of the formalism, the normalization procedure and the parse tree format defined by a grammar.

The syntax definition formalism SDF of Heering *et al.* (1989) was the starting point for the work discussed in this chapter. The definition of SDF2 grew out of the specification of SDF in ASF+SDF. A number of generalizations where applied to make the formalism more orthogonal and uniform and a number of improvements and new features were added based on the experience with SDF in the ASF+SDF Meta-Environment (Klint, 1993). SDF introduced the integration of lexical syntax and context-free syntax, but only at the formalism level. In the implementation, an SDF definition is mapped to a regular grammar defining the lexical syntax and a context-free grammar definining the context-free syntax. The scanners produced for the lexical syntax yield a graph structured stream of all possible tokenizations of the input filtered by a set of lexical disambiguation rules. Although this is a fairly advanced setup, the interface suffers from several of the problems that we discussed in §3.1 and §3.2.

The generalized-LR parsing algorithm was first developed by Tomita (1985) for application in natural language processing. It was later improved by Rekers (1992) and applied in the ASF+SDF Meta-Environment for parsing of programming languages. The algorithm presented in §3.8 is based on Rekers' version. Wagner and Graham (1997) describe the use of GLR parsing in incremental parsing of programming languages. Earley (1970) described the first generalized parsing algorithm that is closely related to the LR algorithm of Knuth (1965). A more recent approach to parsing with dynamic lookahead is the extension of top-down parsing with syntactic predicates of Parr and Quong (1994).

Scannerless parsing was introduced by Salomon and Cormack (1989, 1995). They define an extension of SLR(1) parsing in which the lack of lookahead is repaired by extending item-sets if conflicts are found. This non-canonical SLR(1) parser generation works only for a limited set of grammars, making grammar development difficult. The follow restrictions presented in this chapter are a simplification of the adjacency restriction rule of the NSLR(1) approach in which arbitrary grammar symbols can be forbidden to be adjacent. Our reject productions are called exclusion rules by Salomon and Cormack (1989, 1995).

We have presented a complete implementation for follow restrictions and reject productions, whereas the adjacency restrictions and exclusion rules are only partially implemented in NSLR(1) parsing.

A similar approach using GLR parsing is tried in the area of natural language processing. Tanaka *et al.* (1996) discuss the integration of morphological and syntactic analysis of Japanese sentences in a single GLR parser. The morphological rules describe how words can be formed from characters. Segmentation of a string of characters into a string of words is guided by a connection matrix restricting the categories that can be adjacent in a sentence. These rules do usually not suffice to find an unambiguous segmentation. by integrating morphological composition into the context-free grammar of the syntactic phase, such 'contextual' ambiguities can be avoided. This creates the problem of disambiguating the combined context-free grammar usingthe morphological connection matrix. This is partly done as a filter on the generated LR table and partly dynamically during parsing.

Disambiguation by means of priority and associativity declarations was introduced simultaneously by Aho *et al.* (1975) and Earley (1975). The former describe the solution of conflicts in LR parse tables by means of a restricted form of priorities. Aasa (1991, 1992) describes the solution of LR table conflicts by means of precedence declarations. Thorup (1992, 1994a, 1994b) describes the solution of parse table conflicts by means of a collection of excluded subtrees. The method is more expressive than the priorities of SDF, but only succeeds if all conflicts are solved, which is not guaranteed.

In Chapter 4 logical disambiguation methods are formalized as *disambiguation filters* on sets of parse trees. Based on this approach an efficient implementation of disambiguation by priorities is derived in Chapter 5 from the disambiguation filter for priorities. This derivation forms the foundation for the parser generator algorithm presented in this chapter.

3.11 Conclusions

In this chapter we have presented a new approach to parsing that has several advantages over conventional techniques. It overcomes the drawbacks of the traditional scanner/parser interface by abolishing the scanner completely (hence the name scannerless parsing). The lexical and context-free syntax of a language are described in a single integrated uniform grammar formalism. Lexical ambiguities can frequently be solved by means of the parsing context. Lexical structure and layout are preserved in the parse tree and thus accessible in semantic tools. A more expressive formalism for lexical syntax is obtained, such that for example nested comments can be expressed.

The approach encompasses an expressive syntax definition formalism. A grammar normalizer to reduce the complexity of the formalism by simplifying syntax definitions to context-free grammars with a few extensions. An SLR(1) parser generator that deals with character-classes, follow restrictions and priority and associativity rules. A generalized-LR parser that can be used for arbitrary context-free grammars with reject productions, at least if they are not nested. In parsing unambiguous languages, the GLR parser is used to dynamically handle lookahead problems by forking off parsers in parallel.

Reject productions turn out to be a very expressive device that brings us out of the domain of context-free languages. It is as yet unclear how expressive this formalism is exactly, but we have a lower bound—stronger than context-free because it describes $a^n b^n c^n$—and an upper bound because the parsing problem is decidable.

Priorities are compiled into the parse table such that no parse trees with priority conflicts can be produced by the parser. This reduces the size of the parse forest (in case of ambiguous binary expressions the parse forest grows exponentially) and decreases the number of paths in the graph structured stack. The technique is more general than conventional techniques for this kind of disambiguation and works even if there remain conflicts in the parse table due to other causes. For instance, if the grammar requires more lookahead than the parser generator provides.

An open issue is the fully automatic derivation of lexical disambiguation rules from the grammar that would make the method still easier to use. Apart from this minor point, scannerless generalized-LR parsing is a feasible parsing method that makes syntax definition more expressive and solves a number of problems with conventional parsing approaches.

4

Disambiguation Filters

An ambiguous context-free grammar defines a language in which some sentences have multiple interpretations. For conciseness, ambiguous context-free grammars are frequently used to define even completely unambiguous languages and numerous disambiguation methods exist for specifying which interpretation is the intended one for each sentence. The existing methods can be divided in 'parser specific' methods that describe how some parsing technique deals with ambiguous sentences and 'logical' methods that describe the intended interpretation without reference to a specific parsing technique.

We propose a framework of *filters* to describe and compare a wide range of disambiguation problems in a parser-independent way. A filter is a function that selects from a set of parse trees (the canonical representation of the interpretations of a sentence) the intended trees. The framework enables us to define several general properties of disambiguation methods. The expressive power of filters is illustrated by several case studies.

4.1 Introduction

In the last two decades we have seen the successful development of theory and implementation techniques for efficient, deterministic, parsing of languages defined by context-free grammars. As a consequence, the LL(k) and LR(k) grammar classes and associated parsing algorithms are now dominating the field.

Using parsing techniques based on these subclasses of the context-free grammars has, however, several drawbacks. First of all, syntax definitions may need to be brought into an acceptable, but often unnatural, form that obeys the restrictions imposed by the grammar class being used. More importantly, subclasses of the context-free grammars are not closed under composition, e.g., composing two LR(1) grammars does not necessarily yield an LR(1) grammar. Only the class of context-free grammars itself can support the composition of grammars which is essential for the support and development of modular grammar definitions.

The use of natural, modular, grammars is becoming feasible due to the recent advances in parsing technology for arbitrary context-free grammars. Unfortunately, when leaving the established field of deterministic parsing one encounters a next obstacle: the language defined by a grammar may become *ambiguous* and mechanisms are needed to disambiguate the parse forest (rather than the single parse tree) that will be produced by the parser. Disambiguation encompasses the whole spectrum from simple priority declarations for resolving ambiguities in arithmetic expressions to the use of semantic (e.g., type) information for pruning the parse forest. As a last resort, the *user* of the parser may have to resolve ambiguities interactively.

In this chapter we explore disambiguation mechanisms for general context-free grammars and their impact on parsing. We will concentrate on "logical" disambiguation that can be modeled by a *filter* on sets of parse trees. This excludes disambiguation methods that are inherently intertwined with a specific parsing technique. We study the expressiveness of various filters and their interaction with parsing: as a general rule simpler filters can be applied earlier (during parsing or even during parser generation).

This research was motivated by our experience with the modular syntax definition formalism SDF (Heering *et al.*, 1989) and its implementation based on generalized LR parsing (Rekers, 1992). Although quite elaborate disambiguation techniques are being used (local conflict detection based on priority and associativity, and a multi-set ordering for pruning the parse forest) we keep encountering examples where more fine-tuned filtering would be useful. This suggests an approach based on extensible, user-defined, disambiguation filters. For efficiency reasons, it will be advantageous to apply these filters as early as possible.

The rest of this chapter is structured as follows. In §4.2 we consider several characteristics of disambiguation methods. In §4.3 we introduce some preliminary terminology about context-free grammars and parsing. In §4.4 we define the notion *filter* on sets of parse trees, the disambiguation of a context-free grammar by a filter and several properties of filters. In §4.4.4 through §4.8 we discuss several examples illustrating the expressive power of filters. Finally in §4.9 we discuss related work and related issues.

4.2 Disambiguation

A disambiguation mechanism for context-free languages is a procedure that chooses from a range of possible parses for a sentence the most appropriate one according to some criterion. The architecture we propose to use for disambiguation consists of three parts (see Figure 4.1):

Language description: A *context-free grammar* and a set of *disambiguation rules*. Disambiguation rules concern *lexical disambiguation rules* (e.g., preference for a longest match, preference for keywords over identifiers), *context-free disambiguation rules* (e.g., precedence relations between operators), and *static semantic disambiguation rules* (e.g., type or declaration dependent rules).

Generation phase: A grammar transformer and a parser generator. Typical grammar transformations are the elimination of left/right recursion, and the

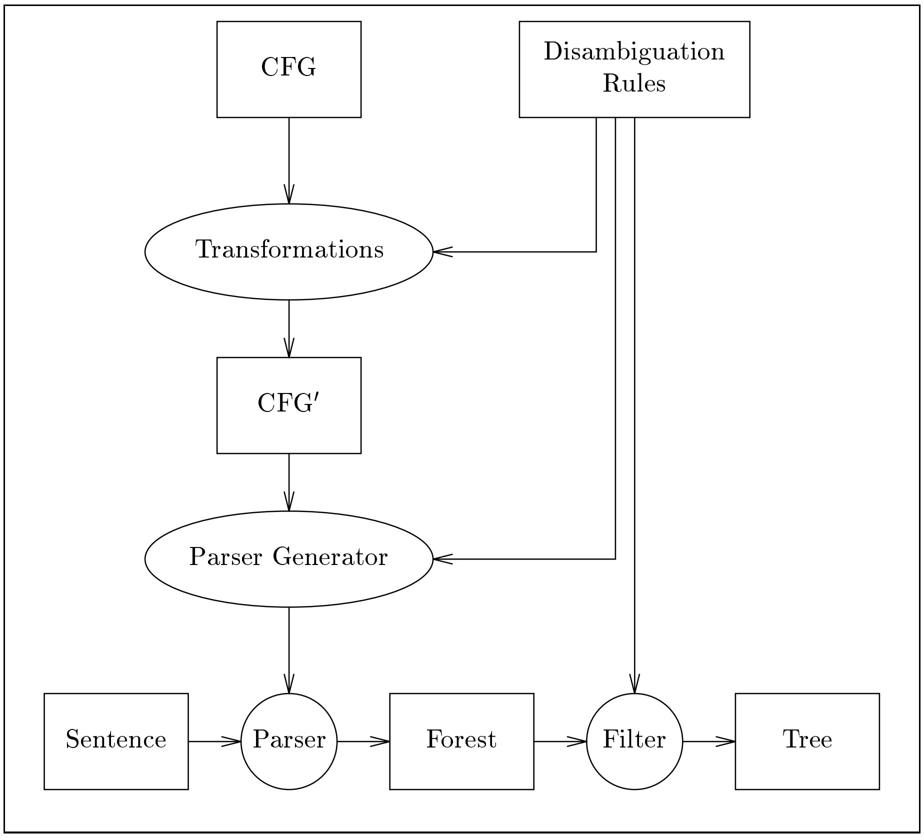


Figure 4.1: Phases in parsing with ambiguous grammars

coding of priority and associativity information in grammar rules. Parser generation is most likely based on standard Generalized-LR techniques (Tomita, 1985, Rekers, 1992).

Parsing phase: A parser/filter pipeline that transforms input sentences into a single (unambiguous) parse tree.

Given this architecture, we can classify disambiguation methods according to the following characteristics:

Interference of context-free grammar and disambiguation rules. Figure 4.1 suggests that the given context-free grammar and disambiguation rules are completely disjoint. In many grammar formalisms, however, they will interfere with each other. For instance, disambiguation rules may be embedded in grammar rules, or the order of grammar rules may have a significance for disambiguation. In this chapter, we will keep them disjoint.

Dependence on parsing method Disambiguation can be defined in terms of parse actions (and is then closely intertwined with parsing) or it can be understood independently from the parsing method used. We will adopt this latter view and we consider the first approach only when it is an implementation method of the latter.

Moment of disambiguation Disambiguation can take place during grammar transformation, during parser generation, and during post-parse filtering. As a general rule, deferring disambiguation is expensive but can be used to implement very general methods. Our strategy will be to define *all* disambiguation methods as post-parse filters and to seek implementation techniques that apply them (transparently but more efficiently) at an earlier moment.

Semantic assumptions An issue in the disambiguation of grammars is the question whether the derivations of an ambiguous sentence should all have the same meaning. In natural language parsing, this is clearly not true. In some other approaches such as for example the approach of Thorup (1994a, 1994b) this seems an essential assumption. In addition, it is not clear whether each sentence generated by the underlying context-free grammar should also be a sentence of the disambiguated grammar. This property is called 'completeness' in Thorup (1994a). But if we consider the language of type-correct Pascal programs we see that this property does not hold.

4.3 Preliminaries

4.3.1 Context-free Grammars

Definition 4.3.1 (Context-free Grammar) A *context-free grammar* $\mathcal{G}$ is a triple $\langle V_N, V_T, \mathcal{P}\rangle$, where V_N is a finite set of nonterminal symbols, V_T a finite set of terminal symbols, V the finite set of symbols of $\mathcal{G}$ is $V_N \cup V_T$, and $\mathrm{P}(\mathcal{G}) = \mathcal{P} \subseteq V^* \times V_N$ a set of productions. We write $\alpha \to A$ for a production $p = \langle \alpha, A\rangle \in \mathrm{P}(\mathcal{G})$. (We will sometimes refer to a production by a number or by an abbreviation of its symbols, e.g., $E + E \to E$ is abbreviated as $+$.) □

The next definitions characterize the language generated by a context-free grammar by the parse trees it generates instead of by derivations. This method is as clear as derivations and has the advantage that the semantics of filters is easily definable.

Definition 4.3.2 (Parse Trees) A context-free grammar $\mathcal{G}$ generates a family of sets of *parse trees* $\mathcal{T}(\mathcal{G}) = (\mathcal{T}(\mathcal{G})(X) \mid X \in V)$, which contains the minimal sets $\mathcal{T}(\mathcal{G})(X)$ such that

$$\frac{X \in V}{X \in \mathcal{T}(\mathcal{G})(X)}$$

$$\frac{A_1 \ldots A_n \to A \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}(\mathcal{G})(A_1),\ \ldots,\ t_n \in \mathcal{T}(\mathcal{G})(A_n)}{[t_1 \ldots t_n \to A] \in \mathcal{T}(\mathcal{G})(A)}$$

We will write t_α for a list $t_1 \ldots t_n$ of trees where α is the list of symbols $X_1 \ldots X_n$ and $t_i \in \mathcal{T}(\mathcal{G})(X_i)$ for $1 \le i \le n$. Correspondingly we will denote the set of all lists of trees of type α as $\mathcal{T}(\mathcal{G})(\alpha)$. Using this notation $[t_1 \ldots t_n \to A]$ can be written as $[t_\alpha \to A]$ and the concatenation of two lists of trees t_α and t_β is written as $t_\alpha t_\beta$ and yields a list of trees of type $\alpha\beta$. The *signature* of a tree is the production used to construct the root of a tree: $\text{sign}([t_\alpha \to A]) = \alpha \to A$. □

We omit the argument $\mathcal{G}$ from $\mathcal{T}(\mathcal{G})$ when the grammar $\mathcal{G}$ is clear from context. We will identify $\mathcal{T}$ and $\bigcup_{X \in V} \mathcal{T}(X)$ when appropriate. According to this definition we should write $[[a \to E] + [b \to E] \to E]$ for a tree with yield $a + b$. When no ambiguity arises we will often write this as $[a + b]$, using only brackets to indicate the tree structure.

Definition 4.3.3 (Yield) The *yield* of a tree t is the string containing all leaves from left to right, i.e.,

$$\begin{aligned} \text{yield}(X) &= X, \text{ if } X \in V_T \\ \text{yield}([t_1 \ldots t_n \to A]) &= \text{yield}(t_1) \ \cdots \ \text{yield}(t_n) \end{aligned}$$

The pointwise extension of yield to sets of parse trees is defined by $\text{yield}(\Phi) = \{\text{yield}(t) \mid t \in \Phi\}$. □

Definition 4.3.4 (Language) The *language* $L(\mathcal{G})$ generated by a context-free grammar $\mathcal{G}$ is the set of strings $\text{yield}(\mathcal{T}(\mathcal{G}))$. The language $L(\mathcal{G})(A)$ generated by nonterminal A is the set $\text{yield}(\mathcal{T}(\mathcal{G})(A))$. □

A context-free grammar is *ambiguous* if it generates at least two different trees t and s such that $\text{yield}(t) = \text{yield}(s)$. Derivation in the classical semantics of context-free grammars and parse trees are similar notions as is witnessed by the following proposition.

Proposition 4.3.5 *For any context-free grammar $\mathcal{G}$ and any $A \in V_N$, $\alpha \in V^*$:* $\alpha \to_{\mathcal{G}}^* A \iff \alpha \in L(\mathcal{G})(A)$ □

4.3.2 Parse Forests

A parse forest is a compact representation of a set of parse trees with the same yield. Compaction is achieved by sharing common subtrees and by packing different trees for the same yield in one node. Parse forests can be described by contexts and sets of contexts.

Definition 4.3.6 (Contexts) A *context* $C[\bullet]$ is a parse tree with exactly one occurrence of a hole $\bullet$. The instantiation $C[t]$ of a context $C[\bullet]$ is constructed by replacing the hole $\bullet$ by the tree t. We denote the set $\{C[t] \mid t \in \Phi\}$ by $C[\Phi]$. Similarly, $\Gamma[\bullet]$ denotes a set of contexts and its instantiation $\Gamma[t]$ is defined as $\{C[t] \mid C[\bullet] \in \Gamma[\bullet]\}$. □

Sharing of a tree t by a set of trees is represented by the instantiation $\Gamma[t]$ of a set of contexts. Packing of a set of trees in a single node is represented by the instantiation $C[\Phi]$ of a context with a set of trees. Sharing of a packed node by a set of trees is denoted by $\Gamma[\Phi]$.

4.3.3 Parsing

Definition 4.3.7 (Parser) A *parser* is a function Π that maps each string $w \in V_T^*$ to a set of parse trees. A parser Π *accepts* a string w if $|\Pi(w)| > 0$. A parser Π is *deterministic* if $|\Pi(w)| \leq 1$ for all strings w. A parser for a context-free grammar $\mathcal{G}$ that accepts exactly the sentences in $L(\mathcal{G})$ is defined by

$$\Pi(\mathcal{G})(w) = \{t \in \mathcal{T} \mid \text{yield}(t) = w\} \qquad \square$$

We restrict our attention to pure parsers that do not modify parse trees during parsing. An example of an implementation of parsers for arbitrary context-free grammars is Tomita's generalized LR algorithm (Lang, 1974, Tomita, 1985, Rekers, 1992). Such a generalized parser produces a parse forest as representation of a set of trees by packing all trees for a subsentence v, the set $\Phi = \Pi(\mathcal{G})(v)$, in a single node and sharing that node in all trees built for the sentence in which v is embedded, i.e., if uvw is a sentence and parsing the sentence $u \bullet w$ produces $\Pi(\mathcal{G})(u \bullet w) = \Gamma[\bullet]$ then the forest for the whole sentence can be constructed as $\Pi(\mathcal{G})(uvw) \supseteq \Gamma[\Phi]$.

4.4 Filters

Ambiguous context-free grammars produce multiple interpretations for some of the sentences they generate. A programming language definition should unambiguously assign to each string a single interpretation. Therefore, if a language definition is based on a context-free grammar, it should select from the multiple interpretations given by the grammar the most appropriate one. We formalize the specification of selection of an appropriate interpretation by the notion of *parse tree filters*. It will turn out that most 'logical' disambiguation methods can be expressed by means of filters.

Definition 4.4.1 (Filter) A *filter* $\mathcal{F}$ for a context-free grammar $\mathcal{G}$ is a function $\mathcal{F} : \wp(\mathcal{T}) \to \wp(\mathcal{T})$ that maps sets of parse trees to sets of parse trees, where $\mathcal{F}(\Phi) \subseteq \Phi$ for any $\Phi \subseteq \mathcal{T}$. The *disambiguation* of a context-free grammar $\mathcal{G}$ by a filter $\mathcal{F}$ is denoted by $\mathcal{G}/\mathcal{F}$. The *language* generated by $\mathcal{G}/\mathcal{F}$ is the set

$$L(\mathcal{G}/\mathcal{F}) = \{w \in V_T^* \mid \exists \Phi \subseteq \mathcal{T}(\mathcal{G}) : \text{yield}(\Phi) = \{w\} \wedge \mathcal{F}(\Phi) = \Phi\},$$

i.e., a string w is a sentence if there exists some non-empty parse forest Φ with yield w from which no trees are removed by the filter $\mathcal{F}$. The *interpretation* of a string w by $\mathcal{G}/\mathcal{F}$ is the set of trees $\mathcal{F}(\Pi(\mathcal{G})(w))$. A filter $\mathcal{F}_2$ is also applicable to a disambiguated grammar $\mathcal{G}/\mathcal{F}_1$, which is denoted by $(\mathcal{G}/\mathcal{F}_1)/\mathcal{F}_2$ and is equivalent to $\mathcal{G}/(\mathcal{F}_2 \circ \mathcal{F}_1)$. $\square$

Given a set of parse trees Φ for some sentence w, a filter selects the 'correct' parse tree(s) in Φ yielding a reduced set of trees $\Phi' \subseteq \Phi$. The condition $\mathcal{F}(\Phi) \subseteq \Phi$ ensures that filters do indeed *reduce* the set of trees instead of inventing new ones. A trivial example of a filter that satisfies this condition is the identity function on sets of parse trees. Often we will define a filter in negative terms by specifying which trees are wrong in some sense and then throw away the wrong trees from a set of trees. Observe that a disambiguated context-free grammar $\mathcal{G}/\mathcal{F}$ generates a subset of the language generated by $\mathcal{G}$, i.e., a string w is only in the language generated by $\mathcal{G}/\mathcal{F}$ if there is at least one tree with yield w that is not rejected by the filter.

This is a very general definition allowing arbitrary functions as filters. Later in this chapter we will consider several classes of filters that use less powerful functions.

4.4.1 Properties of Filters

We now investigate several properties of filters.

Definition 4.4.2 (Completely Disambiguating) A filter is *completely disambiguating* when $|\mathcal{F}(\Pi(\mathcal{G})(w))| \leq 1$ for all $w \in V^*$. □

This is a useful property if the parse trees are input for a next, semantic, processing phase. No provisions have to be made for sets of trees in such a phase. A more restrictive property is completeness.

Definition 4.4.3 (Complete, Thorup (1994a)) A filter $\mathcal{F}$ is *complete* for a context-free grammar $\mathcal{G}$ if $w \in L(\mathcal{G}) \Rightarrow |\mathcal{F}(\Pi(\mathcal{G})(w))| = 1$. □

Actually, Thorup defines a *parser* to be complete if it produces exactly one 'canonical' parse tree for each sentence in the language of its underlying context-free grammar.

Corollary 4.4.4 *If $\mathcal{F}$ is complete for $\mathcal{G}$ then $L(\mathcal{G}) = L(\mathcal{G}/\mathcal{F})$* □

Definition 4.4.5 (Local) A filter $\mathcal{F}$ for a context-free grammar $\mathcal{G}$ is *local* if for each set of contexts $\Gamma[\bullet] \subseteq \mathcal{T}[\bullet]$ and each $\Phi \subseteq \mathcal{T}$

$$\mathcal{F}(\Gamma[\Phi]) \subseteq \Gamma[\mathcal{F}(\Phi)]$$

A filter is *global* if it is not local. □

Global filters are counter intuitive. Rejection by a global filter of a tree for some substring of a sentence does not imply that that tree can not be a subtree of a parse tree for the sentence. A local filter is transparent. A rejected tree can not be a subtree of any larger tree. This means that a local filter can be applied to a local ambiguity instead of to the entire set of complete parse trees for a sentence. It seems that a disambiguation method that can be defined in terms of a local filter is both intuitive and easy to implement.

Definition 4.4.6 (Incremental) A filter $\mathcal{F}$ is *incremental* if, for each pair of sets of parse trees Φ_1, Φ_2, we have $\mathcal{F}(\mathcal{F}(\Phi_1) \cup \mathcal{F}(\Phi_2)) = \mathcal{F}(\Phi_1 \cup \Phi_2)$ □

A generalized parser constructs sets of parse trees for local ambiguities in an incremental fashion. If a filter is incremental, it can be applied to a set whenever an element is added. If it is not incremental application to a set is only legal if the set is completed.

Definition 4.4.7 (Commutative) Two filters $\mathcal{F}_1$ and $\mathcal{F}_2$ are *commutative* if, for each set of trees Φ, we have that $\mathcal{F}_1(\mathcal{F}_2(\Phi)) = \mathcal{F}_2(\mathcal{F}_1(\Phi))$ i.e, if their composition commutes: $\mathcal{G}/\mathcal{F}_1 \circ \mathcal{F}_2 = \mathcal{G}/\mathcal{F}_2 \circ \mathcal{F}_1$ □

Definition 4.4.8 (Context-free) A filter $\mathcal{F}$ for a context-free grammar $\mathcal{G}$ is *context-free* if there is an unambiguous context-free grammar $\mathcal{G}'$ and a function $tr : \mathcal{T}(\mathcal{G}') \to \mathcal{T}(\mathcal{G})$ such that $L(\mathcal{G}') = L(\mathcal{G}/\mathcal{F})$, i.e., $\mathcal{G}'$ generates the same language as $\mathcal{G}/\mathcal{F}$, and $\{tr(\Pi(\mathcal{G}')(w))\} = \mathcal{F}(\Pi(\mathcal{G})(w))$. A filter is *context-dependent* if it is not context-free. □

4.4.2 Specification of Filters

Filters can be defined in many ways. We will consider two special classes of filters that are defined in terms of predicates and relations on trees.

Definition 4.4.9 (Property Filter) The property filter $\mathcal{F}^{\mathcal{E}}$ *generated by the unary predicate* $\mathcal{E}$ (exclude) on trees is defined by

$$\mathcal{F}^{\mathcal{E}}_A(\Phi) = \{t \in \Phi \mid \neg\mathcal{E}(t)\}$$

A predicate $\mathcal{E}$ is *compositional* if for each tree t and each context $C[\bullet]$ $\mathcal{E}(t) \Rightarrow \mathcal{E}(C[t])$. □

A filter $\mathcal{F}^{\mathcal{E}}$ selects all trees which do *not* have property $\mathcal{E}$. The predicate characterizes, for instance, trees with a *conflict*. Compositionality of a filter-predicate ensures that if a tree has a conflict, any tree composed from it has a conflict as well. This implies that to understand a conflict in a sentence one only has to consider the smallest part of the sentence that has the conflict.

Proposition 4.4.10 *A filter $\mathcal{F}^{\mathcal{E}}$ is local iff $\mathcal{E}$ is compositional.*

Proof. ($\Leftarrow$) Let $\Gamma[\bullet]$ a set of contexts and Φ a set of trees. If $t \in \mathcal{F}(\Gamma[\Phi])$, then $\neg\mathcal{E}(t)$ and $t = C[t']$ such that $C[\bullet] \in \Gamma[\bullet]$ and $t' \in \Phi$. Since $\mathcal{E}$ is compositional we have that $\neg\mathcal{E}(t')$ and therefore $t' \in \mathcal{F}(\Phi)$ and thus $t \in \Gamma[\mathcal{F}(\Phi)]$. ($\Rightarrow$) Assume $\mathcal{E}$ is not compositional, i.e., there is some tree t and context $C[\bullet]$, such that $\mathcal{E}(t)$ and not $\mathcal{E}(C[t])$. Then $\mathcal{F}(\{C[t]\}) = \{C[t]\} \not\subseteq \emptyset = C[\mathcal{F}(\{t\})]$ and thus $\mathcal{F}^{\mathcal{E}}$ is not local. □

Definition 4.4.11 The predicate $\overline{\mathcal{E}}$ is defined in terms of $\mathcal{E}$ by $\overline{\mathcal{E}}(t) = \exists s \in \mathrm{sub}(t) : \mathcal{E}(s)$, where $\mathrm{sub}(t)$ denotes the set of all subtrees of t. □

Note that $\overline{\mathcal{E}}$ is always compositional.

Definition 4.4.12 (Comparison Filter) The comparison filter $\mathcal{F}^{\prec}$ *generated by the relation* $\prec$ is defined by

$$\mathcal{F}^{\prec}(\Phi) = \{t \in \Phi \mid \neg\exists t' \in \Phi : t' \prec t\}$$

A relation $\prec$ is *compositional* if $\forall s, t, C[\bullet] : s \prec t \Rightarrow C[s] \prec C[t]$. □

A filter $\mathcal{F}^{\prec}$ selects the minimal trees in a set according to the order $\prec$. Note that if $\prec$ is reflexive or symmetric, the filter $\mathcal{F}^{\prec}$ rejects all trees. For instance, given any context-free grammar $\mathcal{G}$, $\mathcal{G}/\mathcal{F}^{=}$ defines the empty language. The notation $\prec$ suggests that the most useful filters of this kind are based on strict partial orders, i.e., if $\prec$ is transitive, irreflexive and antisymmetric. If $\prec$ is a strict partial order, $\mathcal{F}^{\prec}$ is monotonous, i.e., $\mathcal{F}^{\prec}(\Phi_1) \subseteq \mathcal{F}^{\prec}(\Phi_2)$ if $\Phi_1 \subseteq \Phi_2$, which adds to the clarity of a disambiguation method.

Proposition 4.4.13 *A filter* $\mathcal{F}^{\prec}$ *is local iff* $\prec$ *is compositional.*

Proof. ($\Leftarrow$) Assume $\mathcal{F}$ not local, i.e., there are $\Gamma[\bullet]$, Φ and $s = C[s'] \in \Gamma[\Phi]$ such that $s \in \mathcal{F}(\Gamma[\Phi])$ but $s \notin \Gamma[\mathcal{F}(\Phi)]$. Thus $\neg\exists t \in \Gamma[\Phi] : t \prec s$, i.e., $\forall t \in \Gamma[\Phi] : \neg t \prec s$ and especially $\forall t' \in \Phi : \neg C[t'] \prec C[s']$ then, by compositionality of $\prec$, $\forall t' \in \Phi : \neg t' \prec s'$ which is equivalent to $\neg\exists t' \in \Phi : t' \prec s'$ but this is in contradiction with $\exists t' \in \Phi : t' \prec s'$ which follows from $s \notin \Gamma[\mathcal{F}(\Phi)]$. ($\Rightarrow$) Assume that $\prec$ is not compositional, i.e., there are some s, t and $C[\bullet]$ such that $s \prec t \wedge \neg C[s] \prec C[t]$. But then $\mathcal{F}(C[\{s,t\}]) = C[\{s,t\}] \not\subseteq C[\{s\}] = C[\mathcal{F}(\{s,t\})]$, which contradicts the fact that $\mathcal{F}$ is local. □

4.4.3 Parsers for Disambiguated Grammars

By definition a filter can always be used as a post-parse procedure to prune the parse forest, i.e., $\Pi(\mathcal{G}/\mathcal{F}) = \mathcal{F} \circ \Pi(\mathcal{G})$. For efficiency reasons it is attractive to apply the disambiguation rules described by a filter as early in the parse process as possible.

The problem of producing the most efficient parser from an abstract specification of a filter is probably undecidable. However, for certain classes of filters efficient parsers are possible. By considering many disambiguation methods in this one framework of filters crossovers between implementation strategies might arise.

Definition 4.4.14 (Approximation) An *approximation* of a parser for $\mathcal{G}/\mathcal{F}$ is a parser π such that for any string w $\mathcal{F}(\Pi(\mathcal{G})(w)) \subseteq \pi(w) \subseteq \Pi(\mathcal{G})(w)$ □

If $\mathcal{F}$ is a local filter for a context-free grammar $\mathcal{G}$, we can construct an approximation π for $\mathcal{G}/\mathcal{F}$ by filtering any local ambiguity as soon as it is constructed. Formally, if $\Pi(\mathcal{G})(v)(A) = \Phi$ and $\Pi(\mathcal{G})(u \bullet_A w)(B) = \Gamma[\bullet_A]$ then $\pi(uvw) \subseteq \Gamma[\mathcal{F}(\Phi)]$. If there are no trees left in a local ambiguity the parser that corresponds to it can be stopped, yielding the empty set of trees.

Parsing schemata are abstract specifications of parsing algorithms. In Chapter 5 we start an investigation of the implementation of parsers for grammars disambiguated by filters based on parsing schemata.

4.4.4 Case Studies

In order to assess the feasibility of using filters for the disambiguation of context-free grammars we present several case studies that illustrate the expressive power of our method.

Priorities are a conventional tool for disambiguation and have been proposed in many forms. In §4.5 we study the disambiguation mechanism of SDF which consists of a filter for priority conflicts and a filter for priority comparisons, both derived from a single priority declaration.

Extensible languages are typical examples of languages that are not in the scope of context-free grammars disambiguated by filters. The definition of a filter presumes a set of possible trees from which it selects appropriate ones. A grammar for an extensible language must somehow describe how new productions, i.e., new tree forms, can be introduced. However, restricted forms of extensibility, like Prolog's user-defined operators, are in the range of filters as will be discussed in §4.6.

Landin's offside rule is a disambiguation method based on indentation. In §4.7 we define this method by a filter.

A restricted class of filters based on pattern matching is described in §4.8.

4.5 Priorities

Disambiguation by *precedences* or *priorities* is used by many grammar formalisms in various instantiations (Earley, 1975, Aho *et al.*, 1975, Johnson, 1975, Heering *et al.*, 1989, Aasa, 1992). In this and the next section we study priorities in the syntax definition formalism SDF of Heering *et al.* (1989). An SDF priority declaration induces a strict partial order on grammar productions combined with associativity declarations. From the priority and associativity declarations $\mathcal{R}$ two filters $\mathcal{F}^{\mathcal{E}^{\mathcal{R}}}$ and $\mathcal{F}^{\prec^{\mathcal{R}}}$ are derived. The first removes trees with priority conflicts and the second selects trees which are minimal with respect to a multi-set ordering on trees.

We do not use the notation of SDF for the declaration of priorities but a notation similar to Earley's notation for precedence rules in Earley (1975) that is more suitable for theoretical exposition as in this paper. The concrete notation of SDF can be translated to the abstract notation used here. There have been many proposals for the interpretation of SDF priorities. Here we follow Klint (1988).

4.5.1 Priority Conflicts

Definition 4.5.1 (Priority Declaration) A *priority declaration* $\mathcal{R}$ for a context-free grammar $\mathcal{G}$ is a tuple $\langle \mathrm{L}, \mathrm{R}, \mathrm{N}, > \rangle$, where $\oplus \subseteq \mathrm{P}(\mathcal{G}) \times \mathcal{P}$ for $\oplus \in$

$\{\mathrm{L}, \mathrm{R}, \mathrm{N}, >\}$, such that L, R and N are symmetric and $>$ is irreflexive and transitive. □

The relations L, R and N declare left-, right- and non-associativity, respectively, between productions. The relation $>$ declares priority between productions. A tree with signature p_1 can not be a child of a tree with signature p_2 if $p_2 > p_1$.

Definition 4.5.2 (Priority Conflict) A tree t has a *root priority conflict* $\mathcal{E}^{\mathcal{R}}(t)$ if one of the following rules applies

$$\frac{A \to B\alpha \ (\mathrm{R}^{\mathcal{R}} \cup \mathrm{N}^{\mathcal{R}}) \ B \to \beta}{\mathcal{E}^{\mathcal{R}}([[t_\beta \to B] \ s_\alpha \to A])}$$

$$\frac{A \to \alpha B \ (\mathrm{L}^{\mathcal{R}} \cup \mathrm{N}^{\mathcal{R}}) \ B \to \beta}{\mathcal{E}^{\mathcal{R}}([s_\alpha \ [t_\beta \to B] \to A])}$$

$$\frac{A \to \alpha B \gamma >^{\mathcal{R}} B \to \beta}{\mathcal{E}^{\mathcal{R}}([s_\alpha \ [t_\beta \to B] \ s_\gamma \to A])}$$

A tree t has a *priority conflict*, if $\overline{\mathcal{E}}^{\mathcal{R}}(t)$. □

According to Definition 4.4.9 we can now construct the filter $\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}$. Thus the semantics of the pair $\langle \mathcal{G}, \mathcal{R} \rangle$ is the disambiguated context-free grammar $\mathcal{G}/\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}$. By definition of $\overline{\mathcal{E}}$ in terms of $\mathcal{E}$ we have the following:

Corollary 4.5.3 $\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}$ *is a local filter.* □

Example 4.5.4 The usual example for priorities is the following grammar $\mathcal{G}_{exp}$ for arithmetic expressions

$$\begin{array}{rr} E + E \to E & -E \to E \\ E - E \to E & (E) \to E \\ E * E \to E & a \to E \\ E \uparrow E \to E & b \to E \end{array}$$

that is completely disambiguated by the priority relation $\mathcal{R}_{exp}$:

$$-E > \uparrow > * > \{+, -\}, \ + \mathrm{L} +, \ - \mathrm{L} -, \ + \mathrm{L} -, \ * \mathrm{L} *, \ \uparrow \mathrm{R} \uparrow$$

Now we have, for instance,

$$\begin{aligned} \mathcal{F}^{\mathcal{E}^{\mathcal{R}_{exp}}}(\Pi(\mathcal{G}_{exp})(a + b + c)) &= \mathcal{F}^{\mathcal{E}^{\mathcal{R}_{exp}}}(\{[a + [b + c]], [[a + b] + c]\}) \\ &= \{[[a + b] + c]\} \end{aligned}$$

because $+ \mathrm{L} +$ □

According to the definition above a root priority conflict of a tree can be detected by looking at the signature of the tree and at the signatures of its children. The following version of the predicate is somewhat stronger in that it looks through chain rules.

Definition 4.5.5 (Chain Rule Elimination) The function *ecr* (*chain rule elimination*) yields the first subtree that is not an application of a chain rule:

$$\begin{aligned} ecr(X) &= X \\ ecr([t_B \to A]) &= ecr(t_B) \\ ecr([t_\alpha \to A]) &= [ecr{*}(t_\alpha) \to A], \text{ if } |\alpha| \neq 1 \end{aligned}$$ □

Definition 4.5.6 (Priority Conflict / Chain Rule) A tree t has a root priority conflict *modulo chain rules* ($\mathcal{E}_c^{\mathcal{R}}(t)$) $\mathcal{E}_c^{\mathcal{R}}(t) \iff \mathcal{E}^{\mathcal{R}}(ecr(t))$ □

4.5.2 Multi-set Ordering

After selecting the conflict-free trees from a set there might still be more than one tree in the set. The next filter that is used by SDF selects trees by comparing trees with respect to a multi-set ordering $\prec$ on trees.

Definition 4.5.7 (Multi-set) A *multi-set* is a function $M : \mathrm{P}(\mathcal{G}) \to \mathbf{N}$ that maps productions to the number of their occurrences in the set. The union $M \uplus N$ of two multi-sets M and N is defined as $M \uplus N(p) = M(p) + N(p)$. The empty multi-set is denoted by $\emptyset$, i.e., $\emptyset(p) = 0$ for any p. We write $p \in M$ for $M(p) > 0$. A multi-set with a finite number of elements with a finite number of occurrences can be written as $M = \{p_1, p_1, \ldots, p_2, \ldots\}$, where $M(p)$ is the number of occurrences of p in the list. □

Definition 4.5.8 (Tree as Multi-set) A tree t is translated to a multi-set by $\overline{\bullet} : \mathcal{T} \to (\mathrm{P}(\mathcal{G}) \to \mathbf{N})$ as

$$\begin{aligned} \overline{X} &= \emptyset \\ \overline{[t_1 \ldots t_n \to A]} &= \{\alpha \to A\} \uplus \overline{t_1} \uplus \ldots \uplus \overline{t_n}, \\ &\quad \text{if } \alpha \to A = \mathrm{sign}([t_1 \ldots t_n \to A]) \end{aligned}$$ □

Definition 4.5.9 (Multi-set Order) Given some priority declaration $\mathcal{R}$, the order $\prec^{\mathcal{R}}$ on multi-sets is defined such that $M \prec^{\mathcal{R}} N$ iff

$$M \neq N \wedge \forall y \in M : M(y) > N(y) \Rightarrow \exists x \in N : y >^{\mathcal{R}} x \wedge M(x) < N(x)$$ □

The motivation for this ordering is that it prefers parse trees that are constructed with the smallest possible number of productions of the highest possible priority.

Given a priority declaration $\mathcal{R}$, we can now construct the filter $\mathcal{F}^{\prec^{\mathcal{R}}}$ using Definition 4.4.12 that selects those trees which are minimal with respect to the multi-set ordering induced by the priority declarations.

Proposition 4.5.10 (Klint (1988)) *The multi-set ordering $\prec^{\mathcal{R}}$ on trees is compositional.*

Proof. (a) If $t_1 \prec^{\mathcal{R}} t_2$ then $t_1 \neq t_2$ and thus $T_1 = C[t_1] \neq C[t_2] = T_2$. (b) Assume $\overline{T_1}(y) > \overline{T_2}(y)$, then $\overline{t_1}(y) > \overline{t_2}(y)$. Since $t_1 \prec^{\mathcal{R}} t_2$, $\exists x \in t_2 : y >^{\mathcal{R}} x \wedge \overline{t_1}(x) < \overline{t_2}(x)$, then also $\exists x \in T_2 : y >^{\mathcal{R}} x \wedge \overline{T_1}(x) < \overline{T_2}(x)$. From (a) and (b) we conclude that $C[t_1] \prec^{\mathcal{R}} C[t_2]$. □

Example 4.5.11 The following grammar is a typical example of the working of the multi-set order for the disambiguation of overloaded operators. Expressions over natural numbers,(n) and real numbers (r) with addition and multiplication defined on both types of numbers. The natural numbers are included in the real numbers by means of the injection $N \rightarrow R$.

$$\begin{array}{rr} r \rightarrow R & n \rightarrow N \\ R + R \rightarrow R & N + N \rightarrow N \\ R * R \rightarrow R & N * N \rightarrow N \\ N \rightarrow R & \end{array}$$

This grammar is disambiguated by means of the following priority declaration:

$$*^N > *^R > +^N > +^R,\ +^N \mathrm{L} +^N, +^R \mathrm{L} +^R$$

Given the string $n + n$ the following trees are generated by the grammar (with number of occurrences of productions):

$$[[n \rightarrow N] + [n \rightarrow N] \rightarrow N]$$
$$[[n \rightarrow R] + [n \rightarrow R] \rightarrow R]$$

The first tree contains one occurrence of addition on natural numbers, while the second contains an occurrence of addition on real numbers. Because of the priority $+^N > +^R$, the first tree is lower in the multi-set ordering and is therefore selected by the filter. Some other combinations of strings and trees:

$$\begin{array}{rl} n + n + n & [[n + n \rightarrow N] + n \rightarrow N] \\ n + n + r & [[[n + n \rightarrow N] \rightarrow R] + r \rightarrow R] \\ n + n * r & [[n \rightarrow R] + [[n \rightarrow R] * r \rightarrow R] \rightarrow R] \end{array}$$ □

4.5.3 Shortcomings of Priorities

The following examples give illustrations of grammars that can not or not appropriately be disambiguated with priority rules.

Example 4.5.12 A well-known example is the following grammar $\mathcal{G}_{ie}$ for conditional statements with a *dangling-else* construct.

$$\begin{array}{r} i\ S\ e\ S \rightarrow S \\ i\ S \rightarrow S \end{array}$$

It is disambiguated by the priority declaration $\mathcal{R}_{ie}$: $i\ e > i$. This disambiguation is correct according to the conventional solution of this problem in that it forbids an i as first descendant of $i\ e$, as we can see from the parses of the sentence $iiSeS$:

$$\mathcal{R}_{ie}(\Pi(\mathcal{G}_{ie})(iiSeS)) = \mathcal{R}_{ie}(\{[i[iS]eS]), [i[iSeS]]\}) = \{[i[iSeS]]\}$$

where the first parse is filtered out because it contains a priority conflict against $ie > i$. However, the sentence $iSeiS$ is not a member of $L(\mathcal{G}_{ie}/\mathcal{R})$ since $[iSe[iS]]$ is the only tree in $\mathcal{T}(S)$ with the right yield and it has a priority conflict against $ie > i$. □

Example 4.5.13 Another problem of precedences is posed by the following grammar that defines arithmetic expressions by one *generic production for binary operators.*

$$E\ O\ E \to E,\ + \to O,\ * \to O,\ \cdots$$

This grammar can not be disambiguated like the grammars in Example 4.5.4, although it is useful when generic operations on the trees have to be defined. □

4.6 Prolog Operators

Several languages have mechanisms for introducing new infix, prefix and postfix operators and declaring their precedence and associativity. Here we study a mechanism that allows the user to introduce new operators with relative priority instead of with absolute priority as in Prolog (Bratko, 1990). The meaning of the priorities is the same as in the previous sections, but since the priority declarations are part of the tree, the definition of the filter is more complicated.

The context-free grammar $\mathcal{G}_{prolog}$ describes a language of programs P that consist of a list of clauses C that are either operator declarations D or expressions E. There is an infinite supply of operators O and priority between operators can be declared by the relations R, L and > which have the same meaning as before. A declaration is valid from the point of introduction until the end of the program unless overruled by a new declaration.

$$\begin{array}{rrr}
+ \to O & * \to O & \ldots \\
[a-z]+ \to A & & \\
O\ \mathrm{R}\ O \to D & O\ \mathrm{L}\ O \to D & O > O, \to D \\
E\ O\ E \to E & A \to E & (E) \to E \\
D \to C & E \to C & \\
C.\ P \to P & \epsilon \to P &
\end{array}$$

4.6.1 Global filter

A filter for these programs selects those trees that have expression trees that do not violate the priority declarations earlier in the tree. The first method checks a program tree by traversing it from left to right, checking each expression tree with the priority information it has collected earlier in the traversal.

Definition 4.6.1 The predicate $\mathcal{E}^{\mathcal{R}}$ is defined as follows on program trees

$$\begin{aligned}\mathcal{E}^{\mathcal{R}}([e.p]) &\iff \overline{\mathcal{E}}^{\mathcal{R}}(e) \vee \mathcal{E}^{\mathcal{R}}(p)\\ \mathcal{E}^{\mathcal{R}}([d.p]) &\iff \mathcal{E}^{\mathcal{R}\cup\{d\}}(p)\end{aligned}$$

and for expression trees

$$\begin{aligned}\mathcal{E}^{\mathcal{R}}([[e_1 \oplus e_2] \otimes e_3]) &\Leftarrow \oplus \,\mathrm{R} \otimes \vee \otimes > \oplus\\ \mathcal{E}^{\mathcal{R}}([e_1 \oplus [e_2 \otimes e_3]]) &\Leftarrow \oplus \,\mathrm{L} \otimes \vee \oplus > \otimes\end{aligned}$$

The filter for sets of P-trees over $\mathcal{G}_{prolog}$ can now be defined as $\mathcal{F}(\Phi) = \{t \in \Phi \mid \neg\mathcal{E}^{\emptyset}(t)\}$ □

4.6.2 Local filter

Another approach to selecting the right tree is by means of a local filter. The basic idea of the filter as defined below is that it infers the priority constraints posed by each subtree of a tree. If these constraints form an inconsistent statement the subtree can never be correct with respect to any priority declaration.

Definition 4.6.2 The function pr maps trees in $\mathcal{T}(\mathcal{G}_{prolog})$ to first-order logical formulas.

$$\begin{aligned}\mathrm{pr}([x \to A]) = \forall\otimes,\oplus,\ominus:\ & \otimes > \oplus \Rightarrow \neg\oplus > \otimes \wedge \neg \oplus \mathrm{L} \otimes \wedge \neg \oplus \mathrm{R}\otimes\\ \wedge\ & \otimes > \oplus \wedge \oplus > \ominus \Rightarrow \otimes > \ominus\\ \wedge\ & \otimes \,\mathrm{L} \oplus \Rightarrow \oplus \,\mathrm{L} \otimes \wedge \neg \otimes \mathrm{R}\oplus\\ \wedge\ & \otimes \,\mathrm{R} \oplus \Rightarrow \oplus \,\mathrm{R} \otimes \wedge \neg \otimes \mathrm{L}\oplus\\ \mathrm{pr}([e_1 \otimes e_2 \to E]) = {} & (\mathrm{op}(e_1) > \otimes \ \vee\ \mathrm{op}(e_1)\,\mathrm{L}\otimes) \ \wedge\ \mathrm{pr}(e_1) \wedge\\ & (\mathrm{op}(e_2) > \otimes \ \vee\ \mathrm{op}(e_2)\,\mathrm{R}\otimes) \ \wedge\ \mathrm{pr}(e_2)\\ \mathrm{pr}([e.p \to P]) = {} & \mathrm{pr}(e) \wedge \mathrm{pr}(p)\\ \mathrm{pr}([d.p \to P]) = {} & d \wedge \mathrm{pr}(p)\end{aligned}$$

where $\mathrm{op}([x \to A]) = x$ and $\mathrm{op}([e_1 \oplus e_2 \to E]) = \oplus$. The filter can now be defined as $\mathcal{F}^{\mathrm{pr}}(\Phi) = \{t \in \Phi \mid \exists\mathcal{R} : \mathcal{R} \models \mathrm{pr}(t)\}$ □

Example 4.6.3 Sentence: $* > +.a + b * c$. Trees:

$$\begin{aligned}&\mathrm{pr}((* > +).((a + b) * c).) = * > + \wedge + > * = \bot\\ &\mathrm{pr}((* > +).(a + (b * c)).) = * > +\end{aligned}$$ □

Example 4.6.4 Sentence: $a + b * c + d$. Trees:

$$\begin{aligned}&\mathrm{pr}(((a + b) * (c + d))) = + > *\\ &\mathrm{pr}((((a + b) * c) + d)) = + \,\mathrm{L} *\\ &\mathrm{pr}(((a + (b * c)) + d)) = (* > + \vee * \,\mathrm{R} +) \wedge + \,\mathrm{L} +\\ &\mathrm{pr}((a + ((b * c) + d))) = * > + \wedge + \,\mathrm{R} +\\ &\mathrm{pr}((a + (b * (c + d)))) = + \,\mathrm{R} *\end{aligned}$$ □

It is clear that this disambiguation method can not be applied at parser-generation time, but can very well be applied at parse-time.

Aasa (1991, 1992) describes a disambiguation method for a limited class of context-free grammars with distfix operators based on a predicate on trees. This filter is used to transform context-free grammars into disambiguated context-free grammars which generate the same trees.

4.7 Offside Rule

Several languages use the offside rule to enforce uniform indentation and at the same time reduce the number of keywords for separating constructs. The rule was first formulated by Landin (1966) and later (but shorter) by Richards (1984) as:

> None of an expression's tokens can lie to the left of its first token.

In the following definition disambiguation by the offside rule is defined by means of a filter.

Definition 4.7.1 (Offside) Associate with each occurrence of a terminal $X \in V_T$ its horizontal position $h(X)$. Associate with each tree $t = [t_1 \ldots t_n \to A]$ its horizontal position $h(t) = h(t_1)$ and its minimal horizontal position $hm(t) = \min_{i=1}^{n}(hm(t_i))$. A tree t is *offside* ($o(t)$) if $hm(t) < h(t)$. The grammar $\mathcal{G}/\mathcal{F}^{\overline{o}}$ is disambiguated by the offside rule. □

4.8 Pattern Matching Filters

In §4.5 we saw how priorities can be defined in terms of a unary predicate that checks every node of a tree for a priority violation, i.e., if it matches some pattern that indicates a priority conflict. This method is part of a larger class of disambiguation methods based on pattern matching. This class is attractive since it is weak enough to implement efficiently and it is strong enough to resolve ambiguities in the area of precedence and associativity in an elegant way.

Definition 4.8.1 (Matching) A tree t *matches* a tree (pattern) q, if $q \diamond t$:

$$\begin{aligned} X &\diamond X \\ A &\diamond [t_\alpha \to A] \\ [q_1 \ldots q_n \to A] &\diamond [t_1 \ldots t_n \to A] \Leftarrow q_1 \diamond t_1, \ldots, q_n \diamond t_n \end{aligned}$$

If Q is a set of patterns then $Q \diamond t$ if there is some $q \in Q$ such that $q \diamond t$. □

This definition can easily be extended such that $\diamond$ yields a substitution of the variables—indexed nonterminals–in the pattern, if patterns are linear. We will write $\sigma = q \diamond t$ to indicate that σ is a substitution such that $q\sigma = t$.

4.8.1 Subtree Exclusion

Thorup (1994b) describes a disambiguation method that consists in specifying a set of tree patterns that are excluded from trees produced by a parser. In terms of filters this works according to the following definition.

Definition 4.8.2 (Subtree Exclusion) Given a set Q of tree patterns, the *subtree exclusion filter* $\mathcal{F}^Q$ is defined by

$$\mathcal{F}^Q(\Phi) = \{t \in \Phi \mid \neg\exists s \in \mathrm{sub}(t) : Q \ \diamond \ s\} \qquad \square$$

Disambiguation by priority conflicts as defined in §4.5 can be defined in terms of subtree exclusion by translating the rules in a priority declaration $\mathcal{R}$ to a set of tree patterns $Q_{\mathcal{R}}$ that characterize trees with priority conflicts. For example, if $* > + \in \mathcal{R}$, then the pattern $[[E + E] * E]$ is illegal and therefore the tree $[[a + b] * c]$ is illegal.

Definition 4.8.3 (Priority Conflict Patterns) A priority declaration $\mathrm{Pr}(\mathcal{G})$ derives a pattern set $Q_{\mathrm{Pr}(\mathcal{G})}$ as follows:

$$\frac{\alpha B \to A \ \ \mathrm{L} \cup \mathrm{N} \ \ \beta \to B \ \in \ \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B] \to A] \in Q_{\mathrm{Pr}(\mathcal{G})}}$$

$$\frac{B\alpha \to A \ \ \mathrm{R} \cup \mathrm{N} \ \ \beta \to B \ \in \ \mathrm{Pr}(\mathcal{G})}{[[\beta \to B]\alpha \to A] \in Q_{\mathrm{Pr}(\mathcal{G})}}$$

$$\frac{\alpha B\beta \to A \ > \ \gamma \to B \ \in \ \mathrm{Pr}(\mathcal{G})}{[\alpha[\gamma \to B]\beta \to A] \in Q_{\mathrm{Pr}(\mathcal{G})}}$$

Proposition 4.8.4 *A tree has a root priority conflict (Definition 4.5.2) according to a priority declaration* $\mathrm{Pr}(\mathcal{G})$ *iff it matches one of the patterns in* $Q_{\mathrm{Pr}(\mathcal{G})}$*, i.e.,* $\mathcal{E}^{\mathrm{Pr}(\mathcal{G})}(t) \iff Q_{\mathrm{Pr}(\mathcal{G})} \ \diamond \ t$ $\square$

Subtree exclusion is strictly more expressive than priorities as a disambiguation mechanism: Proposition 4.8.4 proves that each priority declaration can be expressed as a subtree exclusion filter. Example 4.5.13 showed the grammar with generic syntax for infix operators $EOE \to E$ that could not be disambiguated with priorities. By excluding patterns like

$$[E\ [* \to O]\ [E\ [+ \to O]\ E \to E] \to E]$$

the intended disambiguation can be achieved. This higher expressivity of subtree exclusion is due to the fact that arbitrarily deep patterns can be specified, while priorities provide fixed pattern templates—corresponding to associativity and precedence—that are always 2 levels deep. Like priorities, subtree exclusion is not expressive enough for a correct disambiguation of the dangling-else grammar in Example 4.5.12. This is due to the fact that this problem can not be solved with a finite number of fixed depth patterns. Below we will propose to solve this problem by the use of higher-order patterns.

4.8.2 Rewrite Rules

LaLonde and Rivieres (1981) describe a disambiguation method for operator grammars with productions of the form $E \oplus E \to E$. It works by translating a grammar to an unambiguous right-associative context-free grammar with productions $T\ O\ E \to E$ and $\oplus \to O$ and defining *tree transformations* that transform a tree over the unambiguous grammar to the correct tree over the ambiguous grammar. Such a transformed grammar is implemented by a deterministic parser that yields right-associative trees that are transformed after parsing to the correct form by generic rules such as

$$[[E_1\ [\oplus \to O]\ E_2 \to E]\ [\otimes \to O]\ E_3 \to E]$$
$$\Rightarrow [E_1\ [\oplus \to O]\ [E_2\ [\otimes \to O]\ E_3 \to E] \to E], \text{ if } \otimes > \oplus$$

The transformation system is specialized for operator precedence information. A generalization of this technique is achieved by applying an arbitrary *tree rewrite system* instead of operator transformations. For instance, to express that $* > +$, the rewrite system contains a rule

$$[[E_1 + E_2 \to E] * E_3 \to E] \Rightarrow [E_1 + [E_2 * E_3 \to E] \to E]$$

Thorup (1994a) uses this idea in a method for the disambiguation of context-free grammars by term-rewrite systems:

Definition 4.8.5 (Term Rewrite System) A *term rewrite system* (TRS) is a set E of tree pairs (s,t). A tree t *rewrites in one step* to a tree s in a TRS E ($t \to_E s$) if $t = C[t']$, $s = C[s']$ and there is a pair $(q,p) \in E$ such that $\sigma = q \diamond t'$ and $p\sigma\ \diamond\ s'$. A tree t *rewrites* to a tree s if $t \to_E^+ s$. A set of rewrite rules E is yield preserving if $\text{yield}(t) = \text{yield}(s)$ for each $t \to s$ in E. □

Definition 4.8.6 (Rewrite Filter) If E is a yield preserving TRS, then $\mathcal{F}^E$ is the filter defined by

$$\mathcal{F}^E(\Phi) = \{t \in \Phi \mid \neg\exists s \in \Phi : t \to_E^+ s\}$$ □

Proposition 4.8.7 *Rewrite filters $\mathcal{F}^E$ for yield preserving E are local.* □

Proof. Assume $t \in \mathcal{F}(\Gamma[\Phi])$ and $t \notin \Gamma[\mathcal{F}(\Phi)]$. The latter implies that $t = C[t']$ and $\exists s' \in \Phi : t' \to^+ s'$ but then $C[t'] \to +C[s']$ and therefore $t \notin \mathcal{F}(\Gamma[\Phi])$, which contradicts the first assumption. □

The grammar $\mathcal{G}/\mathcal{F}^E$ is not implemented by post-parse filtering, but the TRS is used for the solution of conflicts in LR parse tables. The input for the algorithm is a context-free grammar $\mathcal{G}$ and a TRS E, the output is a complete, linear time parser Π and a TRS $E' = E \cup E''$ if such a pair exists, indication of failure otherwise. The parser Π is a deterministic parser for $L(\mathcal{G})$, that produces for each sentence w a tree t in normal form with respect to E', i.e., there is no tree s such that $t \to_{E'}^+ s$.

Disambiguating rewrite rules can be derived from *semantic equations* $s = t$ that express that two trees (patterns) s and t have the same meaning. If the yields of the left-hand side and right-hand side of such an equation are the same, i.e., $\text{yield}(s) = \text{yield}(t)$, a disambiguation rule choosing either one can be derived. This is especially appropriate for associative operators as in $a \oplus (b \otimes c) = (a \oplus b) \otimes c$. Thorup (1994a) assumes $s = t$ if $\text{yield}(s) = \text{yield}(t)$ and neither $s \to^+ t$ nor $t \to^+ s$.

4.8.3 Higher-Order Patterns

Several disambiguation problems can not be described by fixed-depth patterns. We propose a language of higher-order patterns that adds expressive power to pattern matching. It allows the correct specification of the disambiguation of the dangling-else grammar from Example 4.5.12.

Definition 4.8.8 (Higher-Order Pattern) A *higher-order tree pattern* is an element from the set

$$\mathcal{H} = \mathcal{T} \cup \{\dot{\alpha}, \dot{\beta}, \dot{\gamma}, \ldots\} \cup (V_N \times \mathcal{H}^*)$$

We write $(A \to^* q_1 \ldots q_n)$ for an element of $V_N \times \mathcal{H}^*$. A tree t *matches* with a higher order pattern q if they are in the relation $q \diamond t$:

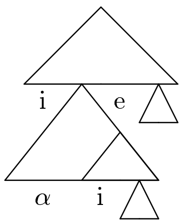
$$\begin{aligned}
X &\diamond X \\
A &\diamond A \to \alpha(t_\alpha) \\
[q_\alpha \to A] &\diamond [t_\alpha \to A] \Leftarrow \forall X \in \alpha : q_X \diamond t_X \\
[\vec{q} \to^* \ A] &\diamond [\vec{t} \to A] \Leftarrow \vec{q} \diamond \vec{t} \\
\dot{\alpha}\,\vec{q} &\diamond \vec{t}_1 \vec{t}_2 \Leftarrow \vec{q} \diamond \vec{t}_2 \\
q\vec{q} &\diamond t_1 \vec{t}_2 \Leftarrow q \diamond t_1 \wedge \vec{q} \diamond \vec{t}_2 \\
\vec{q} &\diamond \vec{t}_0[\vec{t}_1 \to A]\vec{t}_2 \ \Leftarrow \vec{q} \diamond \vec{t}_0 \vec{t}_1 \vec{t}_2
\end{aligned}$$

□

Example 4.8.9 The dangling-else grammar can now be disambiguated by excluding all subtrees that match with

$$[i\ [\dot{\alpha}\ \ i\ S \to^* \ S]\ e\ S \to S]$$

This pattern matches any tree of the form



and thus describes all situations where an if-then appears before an else in a string. □

4.9 Discussion

Many disambiguation methods for context-free grammars for programming languages have been proposed since the early seventies. We can only briefly sketch here some of the related work.

4.9.1 Implementation

Filters are an attractive method for the disambiguation of context-free grammars because they specify the interpretation of a sentence in a logical manner and can be implemented as post-parse filter. An implementation consisting of a standard generalized parser in combination with a post-parse filter allows fast prototyping of, and experimentation with, new disambiguation methods. However, deferring filtering until parsing is complete can be expensive, because many trees built during parsing are thrown away afterwards. If a tree is rejected by a filter after parsing we can look at the parse steps that created it and see at which point the reason for rejection is introduced. These facts can be used to apply filter rules during parsing or even when constructing the parser. In Chapter 5 an investigation is started into the derivation of efficient parsers for context-free grammars disambiguated with filters.

4.9.2 Operator Disambiguation

Disambiguation of arithmetic operators is most commonly done by assigning a priority to each operator and to resolve conflicting priorities during parsing. Traditionally, resolution of priority conflicts and parsing are closely intertwined. Techniques for disambiguation have been applied in all phases discussed earlier in §4.2: grammar transformations, heuristic resolution of table conflicts during parser generation, rule based resolution, and post-parse filtering of parse trees.

Typical grammar transformations are the elimination of left/right recursion, and the coding of priority and associativity information in grammar rules.

Aho *et al.* (1975) describe how parsers for ambiguous grammars of binary expressions can be disambiguated with associativity and precedence declarations. This technique is applied by Johnson (1975) in the parser generator YACC.

Earley (1975) describes a general scheme of precedence relations on context-free productions but only indicates how these could be used in static disambiguation. Precedences in the definition of programming languages are also discussed by Aasa (1991).

User definable disambiguation is, for instance, used in Prolog by declaring the absolute priority and associativity of operators.

The order of the productions in a context-free grammar is used by Wharton (1976) (backtracking) and in YACC (Johnson, 1975) (resolution of shift/reduce conflicts).

Wharton (1976) defines a backtracking parser that is guided by an ordering on parse steps. This ensures a single parse for any sentence over any grammar. However, this resolution of ambiguity is not based on the language being defined but on properties of the grammar productions.

In SDF (Heering *et al.*, 1989) a strict partial order on productions is used as well as relative associativity of productions. This involves the detection of priority conflicts, and a multiset ordering on trees.

Thorup (1992, 1994a, 1994b) describes a technique of resolving LR and LL conflicts based on a set of rewrite rules over parse trees. A consequence of this work is disambiguation by exclusion of a set of tree patterns from the set of legal trees generated by a grammar.

4.9.3 Semantic Disambiguation

Disambiguation can also be combined with the further semantic processing of parse trees. For instance, during static semantic checking (type checking) of a tree disambiguation can be done using type information. Examples of this approach based on attribute grammars can be found in Aasa (1992), Veldhuijzen van Zanten (1988a) and Veldhuijzen van Zanten (1988b), Oude Luttighuis and Sikkel (1992). Van den Brand (1992) describes parse time application of semantic predicates in affix grammars (a variant of attribute grammars). His technique can also be applied to lexical disambiguation. Parr and Quong (1994) describe a disambiguation method that mixes syntactic and semantic disambiguation in LL parsers. Static semantic restrictions on parse trees are also used by Bailes and Chorvat (1993) and McCrosky and Sailor (1993).

An even stronger form of semantics-directed disambiguation can be found in languages such as APL where execution and parsing of a program occur simultaneously and decisions regarding parsing can depend on the outcome of execution.

4.9.4 Filters

The notion of "filtering" as a means of disambiguation has been proposed by other authors as well. A separation between disambiguation and parsing is described by LaLonde and Rivieres (1981) where post-parse transformations on trees are used to produce the right parse tree. The idea is also described by Aasa (1992). In our approach, the treatment of filters and their properties is more abstract and completely independent from the underlying parsing techniques.

In the framework of parsing schemata of Sikkel (1993) the notion of filtering is used for describing refinement relations between parsing algorithms.

In several approaches the *user* is queried interactively to filter ambiguities. An application of user-directed filtering is described by Share (1988) where a modification of YACC is used that reports parse conflicts during parsing (instead of during parser generation) and lets the user solve them. This technique is proposed as a solution of parsing documents in various ambiguous mark-up languages. Tomita (1985) also describes resolution of ambiguities by the user. The implementation of SDF (Heering *et al.* (1989)) uses interactive dialogs to filter ambiguities that could not be resolved by priorities.

We did not propose a *formalism* for the specification of filters, since we mainly explored their semantics. §4.4.2, however, already suggests an approach to the specification of filters using predicates or partial orders, thus abstracting from the application of these to sets of trees or to parse forests.

4.9.5 Unparsing

Disambiguation does not only play a role in parsing but also in unparsing, i.e., generating a string for some semantic value. If parse trees are mapped to abstract syntax trees and in this process bracket functions are considered as identity functions (e.g., (x) and x are identified at the level of abstract syntax trees), there is a problem during the reverse mapping of abstract syntax trees to parse trees since the right brackets may have to be reintroduced. For disambiguation methods based on precedence relations there is some body of knowledge how to do this (e.g., Van den Brand and Visser, 1996). For arbitrary filters new theory is needed, indicating how to unparse in this general case. Blikle (1989) describes the derivation of concrete syntax from abstract syntax. The equations that translate abstract to concrete syntax are very similar to the algebraic specification of pretty printers in Van den Brand and Visser (1996). Blikle's method breaks down when the syntax becomes too concrete, i.e., when brackets become optional. The disambiguation method of Thorup (1994a) is aimed at solving that problem. However, unparsing is not addressed by him.

4.10 Conclusions

We have presented filters as a unifying framework for a large class of existing disambiguation methods. This framework can handle all 'logical' disambiguation methods but is not suited for defining parser-specific methods. All disambiguation methods expressed as filter can be implemented by post-parse filtering. This provides a way of experimenting easily with new methods without having to adapt a given parser generator. For us, the main merit of this framework is an increased understanding of the relationship between parsing and disambiguation. This insight may help during the design of new disambiguation methods and their integration with syntax definition formalisms. In the next chapter we explore techniques for deriving efficient parsers from the combination of a grammar and a filter. These initial ideas suggest that a separation of parsing and filtering at the conceptual level does not exclude the use of efficient parsing techniques at the implementation level.

5

A Case Study in Optimizing Parsing Schemata by Disambiguation Filters

Disambiguation methods for context-free grammars enable concise specification of programming languages by ambiguous grammars. A disambiguation filter is a function that selects a subset from a set of parse trees—the possible parse trees for an ambiguous sentence. The framework of filters provides a declarative description of disambiguation methods independent of parsing. Although filters can be implemented straightforwardly as functions that prune the parse forest produced by some generalized parser, this can be too inefficient for practical applications.

In this chapter the optimization of parsing schemata, a framework for high-level description of parsing algorithms, by disambiguation filters is considered in order to find efficient parsing algorithms for declaratively specified disambiguation methods. As a case study the optimization of the parsing schema of Earley's parsing algorithm by two filters is investigated. The main result is a technique for generation of efficient LR-like parsers for ambiguous grammars disambiguated by means of priorities.

5.1 Introduction

The syntax of programming languages is conventionally described by context-free grammars. Although programming languages should be unambiguous, they are often described by ambiguous grammars because these allow a more natural formulation and yield better abstract syntax. For instance, consider the following grammars. The first, ambiguous grammar gives a clearer and more concise description of arithmetic expressions than the second unambiguous one.

```
"a"         -> E                "a"         -> V    T -> E
E "+" E     -> E                E "+" T     -> E    V -> T
E "*" E     -> E                T "*" V     -> T
"(" E ")"   -> E                "(" E ")"   -> V
```

To obtain an unambiguous specification of a language described by an ambiguous grammar it has to be disambiguated. For example, the first grammar above can be disambiguated by associativity and priority rules that express that `E "*" E -> E` has higher priority than `E "+" E -> E` and that both productions are left associative. In the second grammar these disambiguation rules have been encoded in the grammar itself by means of extra non-terminals.

In Chapter 4 we have set up a framework for specification and comparison of disambiguation methods. In this framework a disambiguation method is described as a *filter* on sets of parse trees. A disambiguation filter is interpreted by parsing sentences according to the ambiguous context-free grammar with some generalized parsing method, for instance Generalized LR parsing (Tomita, 1985, Rekers, 1992), and then prune the resulting parse forest with the filter. Because this method of specification of disambiguation is independent of parsing, a language definition can be understood without understanding a parsing algorithm and it can be implemented by any generalized parser.

Although filters provide a uniform model for the description of disambiguation, they are too inefficient for several applications because all possible parse trees for a sentence have to be built before the intended ones are selected. (The number of possible parse trees for the first grammar above grows exponentially with the length of strings.) The *optimization problem* for filters is to find an efficient parser for the combination of a context-free grammar and a disambiguation filter. The filter can be used to prevent parse steps that lead to parse trees that would be removed by the filter after parsing. *Parsing schemata*, introduced by Sikkel (1993, 1994), are high-level descriptions of parsing algorithms that abstract from control- and data-structures and provide a suitable framework for the study of the interaction between filters and parsers.

Since it is not clear how to solve the optimization problem in general, if that is possible at all, an instance of the problem is studied in this chapter, i.e., the optimization of the underlying parsing schema of Earley's (1970) parsing algorithm by a filter for disambiguation by priorities. This method, which is the disambiguation method of the formalism SDF (Heering *et al.*, 1989), interprets a priority relation on context-free productions as two consecutive filters. The first selects trees without a priority conflict. The second selects trees that are minimal with respect to a multi-set ordering on trees induced by the priority relation.

The main result of this chapter is a parsing schema for parsing with priorities. The schema specifies a complete implementation of parsing modulo priority conflicts and a partial implementation for the multi-set order. The schema can be implemented as an adaptation of any parser generator in the family of LR parser generators. The resulting parsers yield parse trees without priority conflicts.

The method of specifying a disambiguation method by a filter and applying it to optimize the parsing schema of some parsing algorithm appears to be fertile soil for growing new parsing algorithms from old ones.

The rest of the chapter is structured as follows. In §5.2 some preliminary notions are defined. In §5.3 disambiguation filters are defined. In §5.4 parsing schemata are informally introduced. In §5.5 priority rules and the notion of priority conflict are defined and a parsing schema optimized for the priority conflict filter is derived. In §5.6 the relation between Earley parsing and LR parsing is discussed and it is shown how optimization results can be translated from the former to the latter. Furthermore, the results are extended to SLR(1) parsing. In §5.7 the multi-set filter induced by a priority declaration is defined and a partial optimization of the Earley schema for this filter is derived. The two optimizations can be combined in a single schema, obtaining an efficient implementation of disambiguation with priorities.

5.2 Preliminaries

Definition 5.2.1 (Context-free Grammar) A *context-free grammar* $\mathcal{G}$ is a triple $\langle V_N, V_T, \mathcal{P} \rangle$, where V_N is a finite set of nonterminal symbols, V_T a finite set of terminal symbols, V the set of symbols of $\mathcal{G}$ is $V_N \cup V_T$, and $\mathrm{P}(\mathcal{G}) = \mathcal{P} \subseteq V^* \times V_N$ a finite set of productions. We write $\alpha \to A$ for a production $p = \langle \alpha, A \rangle \in \mathcal{P}$. □

The $\alpha \to A$ notation for productions (instead of the traditional $A \to \alpha$) is a convention of the syntax definition formalism SDF to emphasize the use of productions as mixfix function declarations. The string rewrite relation $\to^*_{\mathcal{G}}$ induced by a context-free grammar is therefore also reversed, from a generation relation to a recognition relation. Repeated application of productions rewrites a string to its syntactic category. The statement $w \to^* A$ means that the string w can be reduced to the symbol A.

Observe that we do not distinguish a start symbol from which sentences are derived. Each nonterminal in V_N generates a set of phrases as is defined in the following definition.

Definition 5.2.2 (Parse Trees) A context-free grammar $\mathcal{G}$ generates a family of sets of *parse trees* $\mathcal{T}(\mathcal{G}) = (\mathcal{T}(\mathcal{G})(X) \mid X \in V)$, which contains the minimal sets $\mathcal{T}(\mathcal{G})(X)$ such that

$$\frac{X \in V}{X \in \mathcal{T}(\mathcal{G})(X)}$$

$$\frac{A_1 \ldots A_n \to A \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}(\mathcal{G})(A_1),\ \ldots,\ t_n \in \mathcal{T}(\mathcal{G})(A_n)}{[t_1 \ldots t_n \to A] \in \mathcal{T}(\mathcal{G})(A)}$$

We will write t_α for a list $t_1 \ldots t_n$ of trees where α is the list of symbols $X_1 \ldots X_n$ and $t_i \in \mathcal{T}(\mathcal{G})(X_i)$ for $1 \leq i \leq n$. Correspondingly we will denote the set of all lists of trees of type α as $\mathcal{T}(\mathcal{G})(\alpha)$. Using this notation $[t_1 \ldots t_n \to A]$ can be written as $[t_\alpha \to A]$ and the concatenation of two lists of trees t_α and t_β is written as $t_\alpha t_\beta$ and yields a list of trees of type $\alpha\beta$.

The *yield* of a tree is the concatenation of its leaves. The language $L(\mathcal{G})$ defined by a grammar $\mathcal{G}$ is the family of sets of strings $L(\mathcal{G})(A) = \mathrm{yield}(\mathcal{T}(\mathcal{G})(A))$. □

Definition 5.2.3 (Parsing) A *parser* is a function Π that maps each string $w \in V_T^*$ to a set of parse trees. A parser Π *accepts* a string w if $|\Pi(w)| > 0$. A parser Π is *deterministic* if $|\Pi(w)| \leq 1$ for all strings w. A parser for a context-free grammar $\mathcal{G}$ that accepts exactly the sentences in $L(\mathcal{G})$ is defined by

$$\Pi(\mathcal{G})(w) = \{t \in \mathcal{T}(\mathcal{G})(A) \mid A \in V_N, \text{yield}(t) = w\} \qquad \square$$

Example 5.2.4 As an example consider the ambiguous grammar

```
"a"         -> E
E "+" E     -> E
E "*" E     -> E
"(" E ")" -> E
```

from the introduction. According to this grammar the string $a + a * a$ has two parses:

$$\begin{aligned}\Pi(\mathcal{G})(a + a * a) = \{&[[[a \to E] + [a \to E] \to E] * [a \to E] \to E] \\ &[[a \to E] + [[a \to E] * [a \to E] \to E] \to E]\}\end{aligned} \qquad \square$$

5.3 Disambiguation Filters

Definition 5.3.1 (Disambiguation Filter) A *filter* $\mathcal{F}$ for a context-free grammar $\mathcal{G}$ is a function $\mathcal{F} : \wp(\mathcal{T}) \to \wp(\mathcal{T})$ that maps sets of parse trees to sets of parse trees, where $\mathcal{F}(\Phi) \subseteq \Phi$ for any $\Phi \subseteq \mathcal{T}$. The *disambiguation* of a context-free grammar $\mathcal{G}$ by a filter $\mathcal{F}$ is denoted by $\mathcal{G}/\mathcal{F}$. The *language* $L(\mathcal{G}/\mathcal{F})$ generated by $\mathcal{G}/\mathcal{F}$ is the set

$$L(\mathcal{G}/\mathcal{F}) = \{w \in V_T^* \mid \exists \Phi \subseteq \mathcal{T}(\mathcal{G}) : \text{yield}(\Phi) = \{w\} \ \wedge \ \mathcal{F}(\Phi) = \Phi\}$$

The *interpretation* of a string w by $\mathcal{G}/\mathcal{F}$ is the set of trees $\mathcal{F}(\Pi(\mathcal{G})(w))$. A filter $\mathcal{F}_2$ is also applicable to a disambiguated grammar $\mathcal{G}/\mathcal{F}_1$, which is denoted by $(\mathcal{G}/\mathcal{F}_1)/\mathcal{F}_2$ and is equivalent to $\mathcal{G}/(\mathcal{F}_2 \circ \mathcal{F}_1)$. $\square$

Several properties and examples of filters are discussed in Chapter 4. In §5.5 and §5.7 two examples of disambiguation filters will be presented. The optimization problem for disambiguation filters can be formulated as follows.

Definition 5.3.2 (Optimization by Filter) Given a context-free grammar $\mathcal{G}$ and a filter $\mathcal{F}$, a parser π is an optimization of $\Pi(\mathcal{G})$ if for any string w

$$\mathcal{F}(\Pi(\mathcal{G})(w)) \subseteq \pi(w) \subseteq \Pi(\mathcal{G})(w)$$

We say that π *approximates* $\mathcal{F} \circ \Pi(\mathcal{G})$. π is an optimal approximation if $\pi(w) = \mathcal{F}(\Pi(\mathcal{G})(w))$ for any w. $\square$

5.4 Parsing Schemata

Parsing schemata (Sikkel, 1993, 1997) abstract from the details of control- and data-structures of full parsing algorithms by only considering the intermediate results of parsing. A parsing system is a deduction system that specifies how from a set of hypotheses (the tokens of a sentence) assertions (the intermediate parser states) can be derived according to a set of deduction rules for some context-free grammar. A parsing schema is a parsing system parameterized with a context-free grammar and a sentence. Below parsing schemata are introduced informally by means of an example. A formal treatment can be found in Sikkel (1993, 1997). A related approach is the deductive parsing method of Shieber *et al.* (1995), where inference rules describing parsing algorithms much like parsing schemata are interpreted as chart parsers in Prolog.

Definition 5.4.1 defines a parsing schema for Earley's (1970) parsing algorithm. Its specification consists of an implicit definition of the set of hypotheses H, the definition of a set of items $\mathcal{I}$ and the definition of a set of deduction rule schemata. For each string $a_1 \ldots a_n$ the set of hypotheses H is the set containing the items $[a_i, i-1, i]$ for $1 \leq i \leq n$. The set of items $\mathcal{I}$ is the domain of the deduction system, i.e., the items are the subject of deductions. According to this definition, Earley items are of the form $[\alpha \bullet \beta \rightarrow A, i, j]$, where $\alpha\beta \rightarrow A$ is a production of grammar $\mathcal{G}$. The indices refer to positions in the string $a_1 \ldots a_n$. The intention of this definition is that an item $[\alpha \bullet \beta \rightarrow A, i, j]$ can be derived if $a_{i+1} \ldots a_j \rightarrow^*_{\mathcal{G}} \alpha$ and $a_1 \ldots a_i A\gamma \rightarrow^*_{\mathcal{G}} B$, for some non-terminal B and string of symbols γ. The deduction rules (I) through (C) describe how these items can be derived. Rule (I), the *initialization* rule, specifies that the item $[\bullet\alpha \rightarrow A, 0, 0]$ can always be derived. The *predict* rule (P), states that a production $\gamma \rightarrow B$ can be predicted at position j, if the item $[\alpha \bullet B\beta \rightarrow A, i, j]$ has already been derived. Finally, the rules (S) and (C) finalize the recognition of a predicted and recognized token or nonterminal—witnessed by the second premise—by shifting the $\bullet$ over the predicted symbol.

Definition 5.4.1 (Earley) Parsing schema for Earley's parsing algorithm (Earley, 1970).

$$\frac{\alpha\beta \rightarrow A \in \mathrm{P}(\mathcal{G}),\ 0 \leq i \leq j}{[\alpha \bullet \beta \rightarrow A, i, j] \in \mathcal{I}}$$

$$\frac{}{[\bullet\alpha \rightarrow A, 0, 0]} \tag{I}$$

$$\frac{[\alpha \bullet B\beta \rightarrow A, i, j]}{[\bullet\gamma \rightarrow B, j, j]} \tag{P}$$

$$\frac{[\alpha \bullet a\beta \rightarrow A, i, j],\ [a, j, j+1]}{[\alpha a \bullet \beta \rightarrow A, i, j+1]} \tag{S}$$

$$\frac{[\alpha \bullet B\beta \rightarrow A, h, i],\ [\gamma\bullet \rightarrow B, i, j]}{[\alpha B \bullet \beta \rightarrow A, h, j]} \tag{C}$$

□

A derivation according to a parsing schema is a sequence $I_0, \ldots, I_m$ of items such that for each i ($0 \leq i \leq m$) $I_i \in H$ or there is a $J \subseteq \{I_0, \ldots, I_{i-1}\}$ such that $J \vdash I_i$ is (the instantiation of) a deduction rule. (Observe that if J is empty this corresponds to the case of using a rule without premises, such as the initialization rule.) A string $w = a_1 \ldots a_n$ is in the language of context-free grammar $\mathcal{G}$ if an item $[\alpha\bullet \to A, 0, n]$ is derivable from the hypotheses corresponding to w in the instantiation of the parsing schema in Definition 5.4.1 with $\mathcal{G}$. An item of the form $[\alpha\bullet \to A, 0, n]$ is called a *final* item and signifies that the entire string is recognized as an A phrase. The predicate $w \vdash^P_{\mathcal{G}} I$ expresses that there is a derivation $I_0, \ldots, I_m = I$ of the item I from the hypotheses generated from string w in the instantiation of parsing schema P with grammar $\mathcal{G}$.

The schema in Example 5.4.1 only defines how strings can be recognized. Since disambiguation filters are defined on sets of trees and not on items, a way to relate items to trees is needed. Definition Definition 5.4.3 gives an extension of the schema in Definition 5.4.1 that describes how trees can be built as a result of the deduction steps. First we need a definition of partial parse tree

Definition 5.4.2 (Partial Parse Tree) A partial parse tree is a tree expression of the form $[t_\alpha \to A]$ where $t_\alpha \in \mathcal{T}(\mathcal{G})(\alpha)$ and such that the tree can be completed to a normal tree by adding a list of trees t_β, i.e., $[t_\alpha t_\beta \to A] \in \mathcal{T}(\mathcal{G})(A)$. (which requires $\alpha\beta \to A \in \mathrm{P}(\mathcal{G})$.) □

The items in the schema have the form $[\alpha\bullet\beta \to A, i, j] \Rightarrow [t_\alpha \to A]$ and express that from position i to position j a phrase of type α has been recognized and the partial parse tree $[t_\alpha \to A]$ has been built as a result. The set of hypotheses H is changed such that token items are annotated with trees, i.e., for each token a_i in the string $[a_i, i-1, i] \Rightarrow a_i \in H$. Note how the shift and complete rules extend partial parse trees.

Definition 5.4.3 (Earley with Trees) Parsing schema for Earley's algorithm with construction of parse trees.

$$\frac{\alpha\beta \to A \in \mathrm{P}(\mathcal{G}),\ 0 \leq i \leq j,\ t_\alpha \in \mathcal{T}(\mathcal{G})(\alpha)}{[\alpha \bullet \beta \to A, i, j] \Rightarrow [t_\alpha \to A] \in \mathcal{I}}$$

$$\frac{}{[\bullet\alpha \to A, 0, 0] \Rightarrow [\to A]} \tag{I}$$

$$\frac{[\alpha \bullet B\beta \to A, h, i] \Rightarrow [t_\alpha \to A]}{[\bullet\gamma \to B, i, i] \Rightarrow [\to B]} \tag{P}$$

$$\frac{[\alpha \bullet a\beta \to A, h, i] \Rightarrow [t_\alpha \to A],\ [a, i, i+1] \Rightarrow a}{[\alpha a \bullet \beta \to A, h, i+1] \Rightarrow [t_\alpha a \to A]} \tag{S}$$

$$\frac{[\alpha \bullet B\beta \to A, h, i] \Rightarrow [t_\alpha \to A],\ [\gamma\bullet \to B, i, j] \Rightarrow t_B}{[\alpha B \bullet \beta \to A, h, j] \Rightarrow [t_\alpha t_B \to A]} \tag{C}$$

□

Figure 5.1 shows the derivation of a parse tree for the string $a + a$ with the grammar from Example 5.2.4. The following theorem states that parsing

$[a, 0, 1]$	$\Rightarrow a$
$[+, 1, 2]$	$\Rightarrow +$
$[a, 2, 3]$	$\Rightarrow a$
$[\bullet E + E \to E, 0, 0]$	$\Rightarrow [\to E]$
$[\bullet a \to E, 0, 0]$	$\Rightarrow [\to E]$
$[a\bullet \to E, 0, 1]$	$\Rightarrow [a \to E]$
$[E \bullet +E \to E, 0, 1]$	$\Rightarrow [[a \to E] \to E]$
$[E + \bullet E \to E, 0, 2]$	$\Rightarrow [[a \to E]+ \to E]$
$[\bullet a \to E, 2, 2]$	$\Rightarrow [\to E]$
$[a\bullet \to E, 2, 3]$	$\Rightarrow [a \to E]$
$[E + E\bullet \to E, 0, 3]$	$\Rightarrow [[a \to E] + [a \to E] \to E]$

Figure 5.1: Derivation with the parsing schema in Definition 5.4.3 and the grammar from Example 5.2.4.

as defined in Definition 5.2.3 and derivation with Earley's parsing schema in Definition 5.4.3 are equivalent.

Theorem 5.4.4 (Correctness) *Parsing schema Earley with trees derives exactly the trees produced by a parser, i.e.,* $\{t \mid A \in V_N, w \vdash_{\mathcal{G}}^{5.4.3} [\alpha\bullet \to A, 0, n] \Rightarrow t\} = \Pi(\mathcal{G})(w)$

The following proposition states that the decoration of items with partial parse trees makes no difference to what can be derived. Items in a parsing schema can be annotated with trees as long as they do not affect the deduction.

Proposition 5.4.5 *Parsing schema Earley with trees preserves the derivations of parsing schema Earley, i.e.,* $w \vdash_{\mathcal{G}}^{5.4.1} [\alpha \bullet \beta \to A, i, j] \iff \exists t_\alpha \in \mathcal{T}(\mathcal{G})(\alpha) : w \vdash_{\mathcal{G}}^{5.4.3} [\alpha \bullet \beta \to A, i, j] \Rightarrow [t_\alpha \to A]$

The optimization problem can now be rephrased as:

Definition 5.4.6 (Optimizing Parsing Schemata) The optimization of a parsing schema P by a disambiguation filter $\mathcal{F}$ constitutes in finding a derived parsing schema P' such that

$$\mathcal{F}(\Pi(\mathcal{G})(w)) \subseteq \{t \mid w \vdash_{\mathcal{G}}^{P'} I \Rightarrow t\} \subseteq \{t \mid w \vdash_{\mathcal{G}}^{P} I \Rightarrow t\}$$

where I is some final item. $\square$

5.5 Priority Conflicts

We consider the optimization of parsing schema Earley by two disambiguation filters that are used to interpret the priority disambiguation rules of the formalism SDF of Heering *et al.* (1989). This disambiguation method is also used in the generalization of SDF to SDF2 presented in Part II. The subject of this section is a filter that removes trees with a priority conflict. This filter is similar to the conventional precedence and associativity filter. The declaration of priority rules will also be used in the definition of the multi-set filter in §5.7.

Definition 5.5.1 (Priority Declaration) A *priority declaration* $\mathrm{Pr}(\mathcal{G})$ for a context-free grammar $\mathcal{G}$ is a tuple $\langle \mathrm{L}, \mathrm{R}, \mathrm{N}, > \rangle$, where $\oplus \subseteq \mathcal{P} \times \mathcal{P}$ for $\oplus \in \{\mathrm{L}, \mathrm{R}, \mathrm{N}, >\}$, such that L, R and N are symmetric and $>$ is irreflexive and transitive. □

The relations L, R and N declare left-, right- and non-associativity, respectively, between productions. The relation $>$ declares priority between productions. A tree with signature p_1 can not be a child of a tree with signature p_2 if $p_2 > p_1$. The syntax of priority declarations used here is similar to that in Earley (1975). In SDF (Heering *et al.*, 1989) a formalism with the same underlying structure but with a less Spartan and more concise syntax is used. In SDF one writes `left` for L, `right` for R and `non-assoc` for N. We will use both notations.

Definition 5.5.2 (Priority Conflict) The set $\mathrm{conflicts}(\mathcal{G})$ generated by the priority declaration of a grammar $\mathcal{G}$ is the smallest set of partial trees of the form $[\alpha[\beta \to B]\gamma \to A]$ defined by the following rules.

$$\frac{\alpha B\gamma \to A > \beta \to B \in \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

$$\frac{\gamma \neq \epsilon,\ \beta \to B\ (\mathtt{right} \cup \mathtt{non\text{-}assoc})\ B\gamma \to A \in \mathrm{Pr}(\mathcal{G})}{[[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

$$\frac{\alpha \neq \epsilon,\ \beta \to B\ (\mathtt{left} \cup \mathtt{non\text{-}assoc})\ \alpha B \to A \in \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B] \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

This set defines the patterns of trees with a priority conflict. □

Using the definition of priority conflict we can define a filter on sets of parse trees.

Definition 5.5.3 (Priority Conflict Filter) A tree t has a *root priority conflict* if its root matches one of the tree patterns in $\mathrm{conflicts}(\mathcal{G})$. A tree t has a *priority conflict*, if t has a subtree s that has a root priority conflict. The filter $\mathcal{F}^{\mathrm{Pr}}$ is now defined by $\mathcal{F}^{\mathrm{Pr}}(\Phi) = \{t \in \Phi \mid t \text{ has no priority conflict}\}$. The pair $\langle \mathcal{G}, \mathrm{Pr} \rangle$ defines the disambiguated grammar $\mathcal{G}/\mathcal{F}^{\mathrm{Pr}}$. □

Example 5.5.4 Consider the following grammar with priority declaration

```
syntax
  "a"     -> E
  E "*" E -> E {left}
  E "+" E -> E {left}
```

```
priorities
  E "*" E -> E >
  E "+" E -> E
```

Here the attribute `left` of a production p abbreviates the declaration $p \mathrm{L} p$. The tree

$$[[[a \to E] + [a \to E] \to E] * [a \to E] \to E]$$

has a priority conflict over this grammar—it violates the first priority condition since multiplication has higher priority than addition. The tree

$$[[a \to E] + [[a \to E] * [a \to E] \to E] \to E]$$

does not have a conflict. These trees correspond to the (disambiguated) strings $(a + a) * a$ and $a + (a * a)$, respectively. The implication operator in logic is an example of a right associative operator: $a \to a \to a$ should be read as $a \to (a \to a)$. Non-associativity can be used to exclude unbracketed nested use of the equality operator in expressions using the production `E "=" E -> E`. □

The priority conflict filter induced by a priority declaration can be used to optimize the Earley parsing schema. By the following observation a more general optimization problem can be solved.

Definition 5.5.5 (Subtree Exclusion) A *subtree exclusion filter* based on a set Q of excluded subtrees is defined by

$$\mathcal{F}^Q(\Phi) = \{t \in \Phi \mid \neg t \triangleleft Q\}$$

where $t \triangleleft Q$ (t is excluded by Q) if t has a subtree that matches one of the patterns in Q. □

The optimized parsing schema should not derive trees that contain a subtree contained in Q. As is shown in definition 5.4.3 such patterns are constructed in the complete rule and predicted in the predict rule. The construction of trees with priority conflicts can be prevented by adding an extra condition to these rules. This leads to the following adaptation of the Earley parsing schema.

Definition 5.5.6 (Earley modulo Q) Parsing schema Earley modulo a set Q of parse trees of the form $[\alpha[\gamma \to B]\beta \to A]$, which are excluded as subtrees. The set of items $\mathcal{I}$ and the deduction rules (H), (I) and (S) are copied unchanged from Definition 5.4.3.

$$\frac{[\alpha \bullet B\beta \to A, h, i] \Rightarrow [t_\alpha \to A],\ [\alpha[\gamma \to B]\beta \to A] \notin Q}{[\bullet\gamma \to B, i, i] \Rightarrow [\to B]} \quad \text{(P)}$$

$$\frac{\begin{array}{c}[\alpha \bullet B\beta \to A, h, i] \Rightarrow [t_\alpha \to A],\ [\gamma \bullet \to B, i, j] \Rightarrow t_B,\\ [\alpha[\gamma \to B]\beta \to A] \notin Q\end{array}}{[\alpha B \bullet \beta \to A, h, j] \Rightarrow [t_\alpha t_B \to A]} \quad \text{(C)}$$

□

The following theorem states that parsing schema in Definition 5.5.6 is an optimal approximation of the composition of a subtree exclusion filter (with trees of the form $[\alpha[\gamma \to B]\beta \to A]$) and a generalized parser.

Theorem 5.5.7 (Correctness) *Parsing schema Earley modulo Q derives exactly the trees produced by the composition of a parser and a subtree exclusion filter for Q, i.e.,* $\{t \in \mathcal{T}(\mathcal{G})(A) \mid A \in V_N,\ w \vdash^{5.5.6}_{\mathcal{G},Q} [A \to \alpha\bullet, 0, n] \Rightarrow t\} = \mathcal{F}^Q(\Pi(\mathcal{G})(w))$

This is proved using two lemmas. The soundness lemma asserts that no intermediate parse tree derived with the deduction rules has an excluded subtree (i.e., a priority conflict). The completeness lemma states that every parse tree without a priority conflict can be derived. The completeness lemma is obtained by reverting the implication of the soundness lemma. In these lemmata we use the notion of a context $t_B[\bullet]$ that represents a tree context of type B with one subtree that is a hole $\bullet$. The instantiation $t_B[t_A]$ of a context $t_B[\bullet]$ is the tree obtained by replacing the $\bullet$ subtree by the tree t_A.

Lemma 5.5.8 (Soundness) *For all context-free grammars $\mathcal{G}$, strings $w = a_1 \ldots a_n \in V_T^*$, symbols $A \in V_N$ and $\alpha, \beta \in V^*$, natural numbers $i \leq j \in \mathbf{N}$, and trees $t_\alpha \in \mathcal{T}(\mathcal{G})(\alpha)$ such that $\alpha\beta \to A \in \mathrm{P}(\mathcal{G})$, and Q a set of parse tree patterns of the form $[\alpha[\gamma \to B]\beta \to A]$ we have that*

$$\frac{w \vdash^{5.5.6}_{\mathcal{G}} [\alpha \bullet \beta \to A, i, j] \Rightarrow [t_\alpha \to A]}{\begin{array}{c}\mathrm{yield}(t_A = [t_\alpha\beta \to A]) = a_{i+1} \ldots a_j\beta, \\ \exists t_B[\bullet] \in \mathcal{T}(\mathcal{G})(B) : \neg t_B[t_A] \lhd Q \wedge \mathrm{yield}(t_B[A]) = a_1 \ldots a_i A\delta\end{array}}$$

5.6 From Earley to LR

There is a close correspondence between Earley's algorithm and LR parsing (Knuth, 1965). In fact, parsing schema Earley in Definition 5.4.1 can also be considered the underlying parsing schema of an LR(0) parser. The main difference between the algorithms is that in LR parsing the instantiation of the parsing schema with a grammar is compiled into a transition table. Definition 5.6.1 defines a parsing schema for 'compiled' LR(0) parsing. The intermediate results of an LR parser, the LR states, are sets of LR items closed under prediction, defined by the function *closure*. The function *goto* computes the set of items that results from a state by shifting the dot in the items over a symbol X. The schema defines three deduction rules. Rule (I) generates the initial state consisting of the set of all items $[\bullet\alpha \to A]$ predicting all productions of the grammar. Rule (Sh) obtains a new state from a state by *shifting* a terminal. Rule (Re) reduces a number of states to a new state upon the complete recognition of a production $B_1 \ldots B_m \to B$. It is clear that the function *closure* corresponds to the predict rule (P) in Earley, that (Sh) corresponds to (S) and that (Re) corresponds to (C). A goto-graph is a precomputation of the goto function. Figure 5.2 shows a goto-graph for the grammar of Example 5.2.4.

Definition 5.6.1 (LR(0) Parsing) LR items are Earley items without indices. The items used in LR parsing are sets of LR-items with a pair of indices.

$$\mathcal{I}_{LR} = \{[\alpha \bullet \beta \to A] \mid \alpha\beta \to A \in \mathrm{P}(\mathcal{G})\} \quad \mathcal{I} = \{[\Phi, i, j] \mid \Phi \subseteq \mathcal{I}_{LR}\}$$

The closure of a set of items Φ is the smallest set of items containing Φ and closed under prediction, i.e.,

$$\Phi \subseteq \mathrm{closure}(\Phi)$$

$$\frac{[\alpha \bullet B\beta \to A],\ \gamma \to B \in \mathrm{P}(\mathcal{G})}{[\bullet\gamma \to B] \in \mathrm{closure}(\Phi)}$$

Given a symbol X the goto function maps a set of items to the closure of the set obtained by shifting all items with X.

$$\mathrm{goto}(X, \Phi) = \mathrm{closure}(\{[\alpha X \bullet \beta \to A] \mid [\alpha \bullet X\beta \to A] \in \Phi\})$$

Given these functions an LR parser is defined[1] by the following deduction rules.

$$\frac{}{[\{[\bullet\alpha \to A] \mid \alpha \to A \in \mathcal{G}\}, 0, 0]} \tag{I}$$

$$\frac{[\Phi, h, i],\ [a, i, i+1]}{[\mathrm{goto}(a, \Phi), h, i+1]} \tag{Sh}$$

$$\frac{[\Phi^{[\alpha\bullet B\beta\to A]}, h, i],\ [\Phi_1^{[B_1\bullet\ldots B_k\to B]}, i, i_1],\ \ldots\ [\Phi_k^{[B_1\ldots B_k\bullet\to B]}, i, i_k]}{[\mathrm{goto}(B, \Phi), h, i_k]} \tag{Re}$$

□

In the same way that an LR parser is derived from the Earley schema an LR parser can be derived from the optimized parsing schema of Definition 5.5.6 by adapting the closure and goto functions.

Definition 5.6.2 (LR(0) parsing modulo Q) Items are only predicted if they do not lead to a conflict.

$$\frac{[\alpha \bullet B\beta \to A],\ \gamma \to B \in \mathrm{P}(\mathcal{G}),\ [\alpha[\gamma \to B]\beta \to A] \notin Q}{[\bullet\gamma \to B] \in \mathrm{closure}(\Phi)}$$

Given a production $\gamma \to B$ the goto function maps a set of items to the closure of the set obtained by shifting all items with $\gamma \to B$ for which that does not lead to a conflict.

$$\begin{aligned}\mathrm{goto}(\gamma \to B, \Phi) = \mathrm{closure}(\{[\alpha B \bullet \beta \to A] \mid\ & [\alpha \bullet B\beta \to A] \in \Phi \\ & \wedge [\alpha[\gamma \to B]\beta \to A] \notin Q\})\end{aligned}$$

□

Note that the goto function has to be parameterized with the production that is recognized instead of with just the symbol. (For the (Sh) rule the old goto function is used.) Figure 5.3 shows the goto-graph for the disambiguated grammar from Example 5.5.4.

[1] This definition gives the intermediate results of an LR parser, not its exact control flow.

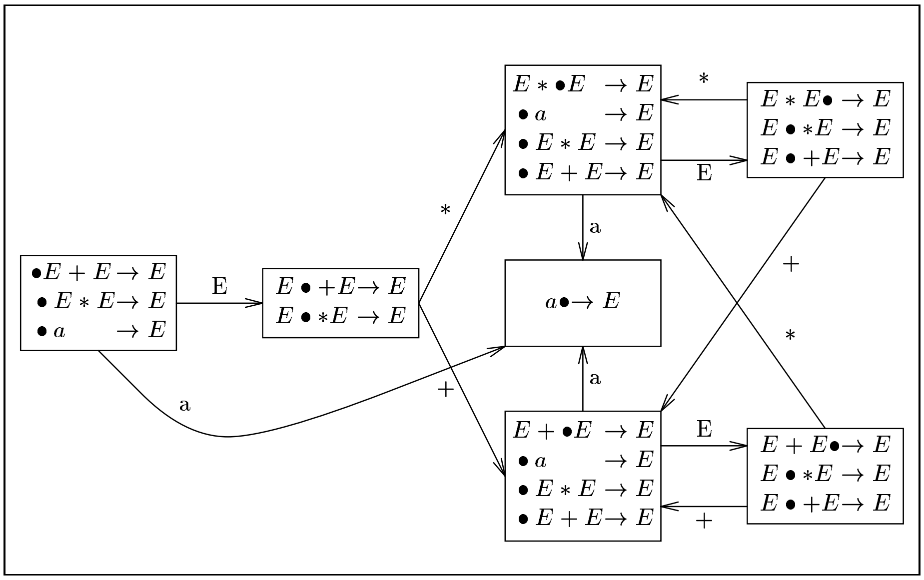


Figure 5.2: LR(0) goto graph for the grammar of Example 5.2.4

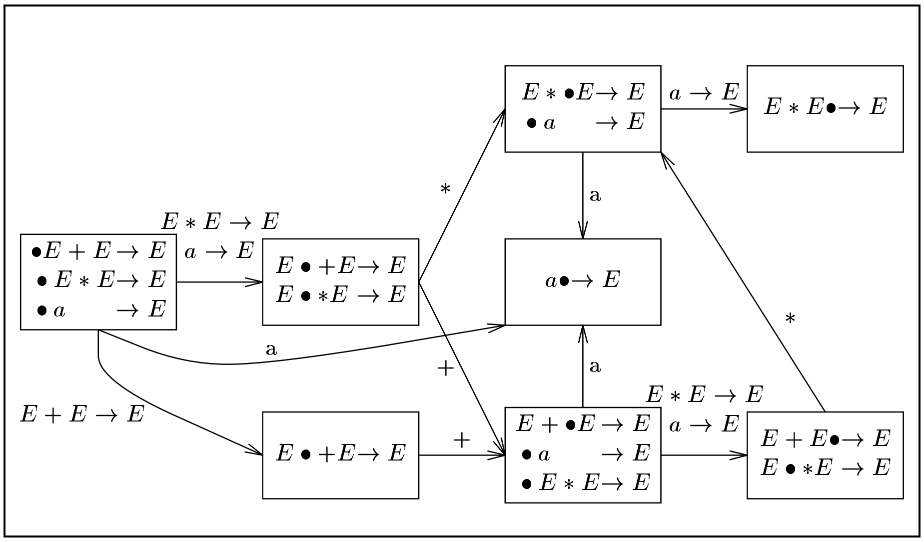


Figure 5.3: LR(0) goto graph for the grammar of Example 5.5.4.

5.6.1 SLR(1) Parsing

The LR(0) goto graph in Figure 5.3 contains conflicts that are easy to prevent with the SLR(1) (Simple LR(1)) extension of LR(0) parsing due to DeRemer (1971). The SLR algorithm is based on the observation that a reduction is only useful if the next symbol in the string can follow the symbol that is recognized by the reduction, i.e., the right hand-side of the production that is reduced. This is expressed in the following adaptation of the LR(0) parsing schema of Definition 5.6.1. The function $\mathrm{first}(\alpha, \Psi)$ yields the set of symbols that can start a phrase derived from a string of symbols α followed by a symbol from the set Ψ. The expression $\mathrm{follow}(B, \Psi)$ denotes the set of symbols that can follow symbol B in a phrase that is followed by a symbol from the set Ψ. The reduce rule now only applies if a production has been recognized *and* the next symbol in the string can follow the right-hand side of the production.

Definition 5.6.3 (SLR(1) Parsing) The set of first symbols of a phrase generated by a string α followed by an element from Ψ, is the smallest set $\mathrm{first}(\alpha, \Psi)$ such that

$$\begin{aligned}
\mathrm{first}(\epsilon, \Psi) &= \Psi \\
\mathrm{first}(a\alpha, \Psi) &= \{a\} \\
\mathrm{first}(A\alpha, \Psi) &= \textstyle\bigcup_{\beta \to A \in \mathrm{P}(\mathcal{G})} \mathrm{first}(\beta\alpha, \Psi)
\end{aligned}$$

The set of symbols that can follow a symbol B in a phrase generated by $\mathcal{G}$ followed by a symbol from Ψ is the smallest set such that

$$\frac{\alpha B \beta \to A \in \mathrm{P}(\mathcal{G})}{\mathrm{follow}(B, \Psi) \supseteq \mathrm{first}(\beta, \mathrm{follow}(A, \Psi))}$$

The reduce rule of the schema in Definition 5.6.1 is restricted by requiring that the next symbol in the string is an element of the follow set of B.

$$\frac{\begin{array}{c}[\Phi^{[\alpha \bullet B\beta \to A]}, h, i],\ [\Phi_1^{[B_1 \bullet \ldots B_k \to B]}, i, i_1],\ \ldots, [\Phi_k^{[B_1 \ldots B_k \bullet \to B]}, i, i_k], \\ [a, i_k, i_k + 1],\ a \in \mathrm{follow}(B, \{\$\})\end{array}}{[\mathrm{goto}(B, \Phi), h, i_k]}$$

□

The SLR(1) schema can be adapted in the same way as the LR(0) schema to account for priority conflicts (or subtree exclusion). However, the definition of follow above is too weak for this extended schema. For instance, in the grammar of Example 5.5.4, the token $*$ is an element of the follow set of E. However, $*$ can not follow an E if it is a $E + E \to E$, i.e., if a reduction is done with $E + E \to E$, no action for $*$ is possible. The following parsing schema optimizes the SLR(1) parsing schema by defining the follow set for a production instead of for a symbol and adapting the reduce rule accordingly. Figure 5.4 shows the SLR(1) table for the grammar of Example 5.5.4.

Definition 5.6.4 (SLR(1) Parsing Modulo Q) This schema defines SLR(1) parsing modulo a set Q of parse trees of the form $[\alpha[\beta \to B]\gamma \to A]$ using the definition of the closure and goto functions from the parsing schema in Definition 5.6.2 and the definition of first from Definition 5.6.3.

$$\frac{\alpha B \gamma \to A \in \mathrm{P}(\mathcal{G}),\ [\alpha[\beta \to B]\gamma \to A] \notin Q}{\mathrm{follow}(\beta \to B, \Psi) \supseteq \mathrm{first}(\beta, \mathrm{follow}(\alpha B \gamma \to A, \Psi))}$$

state	a	$*$	$+$	$\$$	1	2	3	4
0	s 1				3	3	4	2
1		r 1	r 1	r 1				
3		s 8	s 5	acc				
4			s 5	acc				
5	s 1				7	7		
7		s 8	r 3	r 3				
8	s 1				9			
9		r 2	r 2	r 2				

(1) $a \rightarrow E$
(2) $E * E \rightarrow E$
(3) $E + E \rightarrow E$
(4) $E\,\$ \rightarrow S$

(2) > (3)
(2) L (2)
(3) L (3)

Figure 5.4: SLR(1) table for the grammar of example 5.5.4. s n denotes *shift to state* n, r n denotes reduce with production n, acc denotes *accept*. The right part of the table contains the goto entries for the productions. This parse table corresponds to the goto graph of figure 5.3.

The reduce rule is adapted to the new definition of follow.

$$\frac{\begin{array}{c}[\Phi^{[\alpha\bullet B\beta\to A]}, h, i],\ [\Phi_1^{[B_1\bullet\ldots B_k\to B]}, i, i_1],\ \ldots, [\Phi_k^{[B_1\ldots B_k\bullet\to B]}, i, i_k],\\ [a, i_k, i_k+1],\ a \in \mathrm{follow}(B_1\ldots B_k \to B, \{\$\})\end{array}}{[\mathrm{goto}(B_1\ldots B_k \to B, \Phi), h, i_k]} \quad \text{(Re)}$$

□

5.6.2 Discussion

Conventional methods for disambiguating grammars that apply to LR parsing disambiguate the grammar by solving conflicts in an existing LR table. The classical method of Aho *et al.* (1975) uses associativity and precedence information of a limited form—a linear chain of binary operators that have non-overlapping operator syntax—to solve shift/reduce conflicts in LR tables. The method is based on observations on how such conflicts should be solved given precedence information, without a real understanding of the cause of the conflicts. Aasa (1991, 1992) describes filtering of sets of parse trees by means of precedences. Thorup (1994a) describes a method that tries to find a consistent solution for all conflicts in an LR table starting from, and producing a set of excluded subtrees.

All these methods fail on grammars that are inherently non-LR(k), i.e., for which there is no complete solution of all conflicts in any LR table for the grammar. An example is the grammar

```
syntax
                -> L
  L [\ \t\n]    -> L
  "a"           -> E
  E L "*" L E -> E {left}
  E L "+" L E -> E {left}
```

```
priorities
  E L "*" L E -> E >
  E L "+" L E -> E
```

that models arithmetic expressions with layout. The tokens of expressions can be separated by any number of spaces, tabs or newlines, which requires unbounded lookahead. Such grammars are the result of integrating the lexical syntax and context-free syntax of a language into a single grammar as is proposed in Chapter 3. Parsers for such grammars are called scannerless parsers because the tokens they read are the characters from the input file. This grammar is disambiguated completely (it has no ambiguous sentences) with priorities, resulting in an LR table that contains some LR-conflicts, but that does not produce trees with priority conflicts. In combination with a nondeterministic interpreter, e.g., Tomita's generalized LR algorithm (Tomita, 1985), of the parse tables this gives an efficient disambiguation method for languages on the border of determinism.

Thorup (1994b) describes a transformation on grammars based on a set of excluded subtrees to disambiguate a grammar. This method could be used to generate conflict free parse tables as far as possible. Because such a transformation introduces new grammar symbols, more states and transitions are needed in the parse table than for the original grammar. Since the method defined above also introduces some extra states, it would be interesting to compare the LR tables produced by both methods.

5.7 Multi-set Filter

The multi-set ordering on parse trees induced by a priority declaration solves ambiguities not solvable by priority conflicts. A certain class of ambiguities solved by the multi-set order does not need the full power of multi-sets, only a small part of both trees are actually compared. Based on this observation an optimization of the Earley schema that partially implements the multi-set filters can be defined.

Definition 5.7.1 (Multi-sets) A *multi-set* is a function $M : \mathrm{P}(\mathcal{G}) \to \mathbf{N}$ that maps productions to the number of their occurrences in the set. The union $M \uplus N$ of two multi-sets M and N is defined as $(M \uplus N)(p) = M(p) + N(p)$. The empty multi-set is denoted by $\emptyset$, i.e., $\emptyset(p) = 0$ for any p. We write $p \in M$ for $M(p) > 0$. A multi-set with a finite number of elements with a finite number of occurrences can be written as $M = \{p_1, p_1, \ldots, p_2, \ldots\}$, where $M(p)$ is the number of occurrences of p in the list. A parse tree t is interpreted as a multi-set of productions by counting the number of times a production acts as the signature of a subtree of t, where $\alpha \to A$ is the signature of $[t_\alpha \to A]$. □

The following definition due to Jouannaud and Lescanne (1982) defines an ordering on multi-sets.

Definition 5.7.2 (Multi-set Order) Given some priority declaration $\mathrm{Pr}(\mathcal{G})$, the order $\prec^{\mathrm{Pr}(\mathcal{G})}$ on multi-sets is defined as

$$M \prec^{\mathrm{Pr}(\mathcal{G})} N \iff$$
$$M \neq N \wedge \forall y \in M : M(y) > N(y) \Rightarrow \exists x \in N : y >^{\mathrm{Pr}(\mathcal{G})} x \wedge M(x) < N(x)$$

□

Definition 5.7.3 (Multi-set Filter) Given a priority relation $\mathrm{Pr}(\mathcal{G})$, the multi-set filter $\mathcal{F}^{\prec^{\mathrm{Pr}(\mathcal{G})}}$ is defined by

$$\mathcal{F}^{\prec^{\mathrm{Pr}(\mathcal{G})}}(\Phi) = \{t \in \Phi \mid \neg\exists s \in \Phi : s \prec^{\mathrm{Pr}(\mathcal{G})} t\}$$ □

The motivation for this filter is that it prefers parse trees that are constructed with the smallest possible number of productions of the highest possible priority.

Example 5.7.4 Consider the grammar

```
syntax
  "n"       -> N
  N "+" N -> N
  N         -> R
  "r"       -> R
  R "+" R -> R
```

that describes the language of 'naturals' and 'reals' with an overloaded addition operator. The sentence $n+n$ can be parsed as $[[n \to N]+[n \to N] \to N]$ and as $[[[n \to N] \to R]+[[n \to N] \to R] \to R]$. This ambiguity can be solved, choosing either the first or the second tree, by declaring one of the priority rules

```
  N "+" N -> N > R "+" R -> R
```

or

```
  R "+" R -> R > N "+" N -> N
```

Note that with the second priority rule, the production $N + N \to N$ is only used as a parse tree in a context where no R is allowed. Therefore, the first priority rule is assumed in further examples. □

The multi-set order is too strong for this kind of disambiguation. To solve the ambiguity there is no need to compare the complete trees, as the multi-set order does. Comparing the patterns $[[N+N \to N] \to R]$ and $[[N \to R]+[N \to R] \to R]$ is sufficient. The goto graph corresponding to the Earley parser for the example grammar (Figure 5.5) shows that the partial phrase $n+$ causes a conflict (in the left-most state at the bottom row) after completing the production $[n \to N]$. The parser can either shift with $+$ or complete with the chain rule of $N \to R$. However, only after having seen what follows the $+$ a decision can be made. In the following adaptation of the Earley parsing schema the cause of these early decision problems is solved by not predicting and completing chain production, but instead storing them in items.

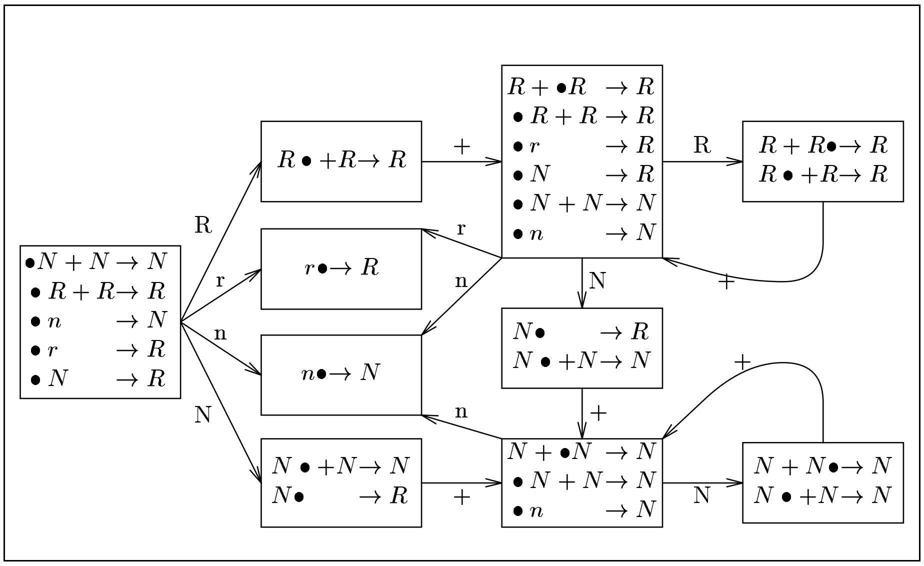


Figure 5.5: Goto graph for the grammar of Example 5.7.4

Definition 5.7.5 (Earley modulo Chain Rules) Let V_C be the set containing all chain symbols $[C \rightarrow B]$, where B and C are nonterminals in context-free grammar $\mathcal{G}$ and $B = C$ or $C \rightarrow_{\mathcal{G}} B_1 \rightarrow_{\mathcal{G}} \cdots \rightarrow_{\mathcal{G}} B_m \rightarrow_{\mathcal{G}} B$, $(m \geq 0)$. Symbols $[A \rightarrow A]$ and A are identified. A production with chain symbols $[C \rightarrow B]$ in its left-hand side is identifiable (member of grammar, priority relation) with a production where the chain symbols are replaced with their heads B. The (I) and (S) rules are as usual.

$$\frac{\alpha\beta \rightarrow A \in \mathcal{G},\ |\alpha\beta| \neq 1 \vee \alpha\beta = a \in V_T,\ 0 \leq i \leq j}{[\alpha \bullet \beta \rightarrow A, i, j] \in \mathcal{I}}$$

$$\frac{[\alpha \bullet B\beta \rightarrow A, h, i],\ [C \rightarrow B] \in V_C}{[\bullet\gamma \rightarrow C, i, i]} \quad \text{(P)}$$

$$\frac{[\alpha \bullet B\beta \rightarrow A, h, i],\ [\gamma\bullet \rightarrow C, i, j],\ |\beta| > 0}{[\alpha[C \rightarrow B] \bullet \beta \rightarrow A, h, j]} \quad \text{(C1)}$$

$$\frac{[\alpha \bullet B \rightarrow A, h, i], [\gamma\bullet \rightarrow C, i, j], \neg[\alpha'\bullet \rightarrow A', h, j],\ \alpha' \rightarrow A' > \alpha B \rightarrow A}{[\alpha[C \rightarrow B]\bullet \rightarrow A, h, j]} \quad \text{(C2)}$$

□

The negative premise $\neg[\alpha\bullet \rightarrow A, i, j]$ in combination with the condition $A' \rightarrow \alpha' > A \rightarrow \alpha B$ is used in rule (C2) to express that an item $[\alpha[C \rightarrow B]\bullet \rightarrow A, h, j]$ can be derived from $[\alpha \bullet B \rightarrow A, h, i]$ and $[\gamma\bullet \rightarrow C, i, j]$ only if no item $[\alpha'\bullet \rightarrow A', h, j]$ can be derived such that $A' \rightarrow \alpha'$ has higher priority than $A \rightarrow \alpha B$.

With the introduction of negative premises we leave the domain of parsing schemata as defined in Sikkel (1993) and this deserves a more thorough investigation than is possible in the scope of this chapter. However, two points about

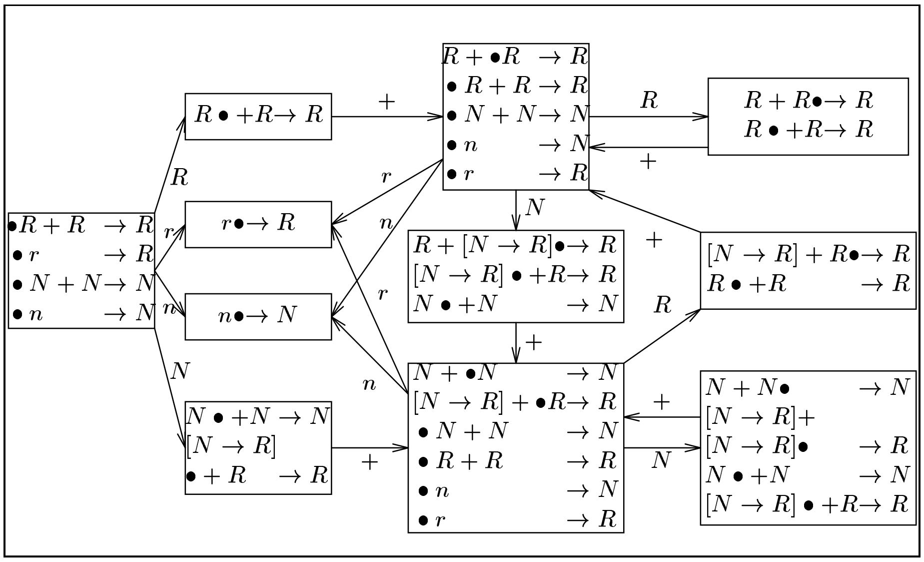


Figure 5.6: Goto graph for grammar of Example 5.7.4 corresponding to parsing schema in Definition 5.7.5. The item $[[N \to R] + [N \to R]\bullet \to R]$ is present if the negative premise of rule (C2) is absent.

this feature can be observed: (1) As used here the notion has a straightforward implementation in an LR-like compilation scheme: first construct the complete set of items and then choose the maximal items from it. (2) The priority relation $>$ on productions is irreflexive by definition, which entails that rule (C2) has no instantiation of the form $I_1, I_2, \neg I_3 \vdash I_3$ that would make the schema inconsistent.

Example 5.7.6 Figure 5.6 shows the goto graph for the grammar of Example 5.7.4 according to the parsing schema in Definition 5.7.5. The shift/reduce conflict between the items $[N \bullet +N \to N]$ and $[N\bullet \to R]$ is changed into a reduce/reduce conflict between the items $[N + N\bullet \to N]$ and $[[N \to R] + [N \to R]\bullet \to R]$. If the negative premise of rule (C2) is taken into account the item $[[N \to R] + [N \to R]\bullet \to R]$ can not be derived, and is not present in the goto-graph. The conflict is solved. □

The method does not help for grammars where the ambiguity is not caused by chain rules, for instance consider the following example due to Kamperman (1992)

```
syntax
  E E     -> E
  "-" E   -> E
  E "-" E -> E
priorities
```

```
E E     -> E >
"-" E   -> E >
E "-" E -> E
```

It defines expressions formed by concatenation, prefix minus and infix minus.

The methods developed in this chapter can be combined into a parsing schema that handles both priority conflicts and the partial implementation of multi-set filters by adding the subset exclusion conditions to the (P), (C1) and (C2) rules of the parsing schema in Definition 5.7.5. As a bonus this combined parsing schema handles priority conflicts modulo chain rules.

5.8 Conclusions

In this chapter two disambiguation methods specified as a filter on sets of parse trees were considered. These filters were used to optimize parsers for context-free grammars by adapting their underlying parsing schema.

The first optimization uses priority conflicts to prevent ambiguities. The resulting Earley parsers modulo priority conflicts are guaranteed not to produce trees with priority conflicts, even for grammars with overlapping operators, layout in productions or other problems that need unbounded lookahead. In combination with a GLR interpreter of the parse tables this gives an efficient disambiguation method for languages with unbounded lookahead. The second optimization covers a subset of the ambiguities solved by multi-set filters. Together these optimizations can be used in the generation of efficient parsers for a large class of ambiguous context-free grammars disambiguated by means of priorities.

Parsing schemata provide a high-level description of parsing algorithms that is suitable for the derivation of new algorithms. The introduction of negative items was needed to express the optimization for the multiset filter and needs more research. This first experiment in implemenation of disambiguation methods from formal specifications encourages research into a fuller optimization of multiset filters and application of this approach to other disambiguation methods.

The deductive parsing approach of Shieber *et al.* (1995) and its implementation in Prolog could be used to prototype such optimized schemata. Deductive parsing consists in computing the closure of a set of axiom items under the inference rules of a schema, resulting in all items derivable for a sentence. Compiling the inference rule of a schema into a parse table for a specific grammar increases the efficiency of an algorithm, since work is shifted from the parser into the parser generator. It seems feasible to generalize the compilation of Earley rules into LR tables to other schemata, thus obtaining a very declarative method for creating new parser generators.

Part II

Context-Free Syntax Definition

6

A Family of Syntax Definition Formalisms

In the next chapters we present the design and specification of a family of syntax definition formalisms. The kernel of this family of formalisms is formed by context-free grammars. A number of orthogonal extensions to the kernel is defined. Many of these extensions are defined in terms of the primitives of the kernel by means of normalization functions. This provides a framework for constructing new formalisms by adapting and extending previous ones.

Included in the family are the following extensions of context-free grammars: uniform definition of lexical and context-free syntax, variables, disambiguation by priorities, follow restrictions and reject productions, a rich set of regular expressions defined in terms of context-free productions, character classes, aliases, parameterized modules with hidden imports and renamings. The accumulation of these extensions is the syntax definition formalism SDF2.

This chapter provides an introduction to SDF2 and gives an overview of the design and specification of the family of formalisms.

6.1 Introduction

New programming, specification and special purpose languages are being developed continuously. Syntax definition formalisms play a crucial role in the design and implementation of new languages. Syntax definition formalisms also play a role embedded in other languages: regular expressions in edit operations, macro definitions for macro preprocessors, user definable infix or distfix operators in programming languages, grammars as signatures in algebraic specification formalisms, and documents that contain a description of their own syntax.

The core of many syntax definition formalisms is formed by context-free grammars, which are widely used in computer science since their introduction by Chomsky (1956). A context-free grammar is a set of string rewrite rules of the form $\alpha \to \mathcal{A}$ with α a string $\mathcal{A}_1 \ldots \mathcal{A}_n$ of zero or more symbols and $\mathcal{A}$ a symbol. A string (a sequence of symbols) w is a member of the language described by a grammar $\mathcal{G}$ if it can be rewritten to the start symbol S, i.e., if there is a sequence

$w = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_n = S$ such that each step has the form $\alpha_i \to \alpha_{i+1}$ with $\alpha_i = \beta_1\beta_2\beta_3$ and $\alpha_{i+1} = \beta_1 B \beta_3$ and $\mathcal{G}$ contains a production $\beta_2 \to B$.

Despite, or maybe due to, the simplicity of this basic structure there has never emerged a standard formalism for syntax definition. The Backus Naur Form (BNF), originally developed by Backus (1959) and Naur *et al.* (1960) for the definition of the syntax of Algol, is a commonly used notation for context-free grammars, but it does not have the status of a standard and many variants are in use. Several standard notations for syntax definition have been proposed (e.g., Wirth, 1977, Williams, 1982). None of these has been convincing, instead a number of similar or overlapping formalisms exist.

The reason for this divergence is that a practical syntax definition formalism serves not only to define languages, i.e, sets of strings. Syntax definitions are also interpreted as recognizers that decide whether a string belongs to a language, as parsers that map strings to parse trees, as mappings from parse trees to abstract syntax trees and as syntax directed editors. Plain context-free grammars are not adequate for this purpose. To support the compact definition of languages, formalisms can provide a variety of features as extensions to the basic structure: character classes, regular expressions, disambiguation by associativity and priority declarations, reuse by modularization, parameterization of language definitions, interfacing between the formalism and its environment, e.g., mapping to abstract syntax.

Various extensions of context-free grammars have been developed for attaching semantics to grammars: Attribute grammars (Knuth, 1968) attach attribute evaluation rules to productions. The computation of the semantics of a parse tree consists in computing the values of all attributes. This computation is orthogonal to parsing. Affix grammars (Koster, 1971) and extended affix grammars (Watt, 1977) are similar to attribute grammars, but predicates on affix values can play a role in disambiguation during parsing. Definite clause grammars (Pereira and Warren, 1980) are based on the Horn clauses of logic programming. Parsing is performed by the SLD resolution evaluation mechanism. Semantic values are represented by means of terms and passed around using unification and logic variables. Other approaches including algebraic specification use a separate formalism to define the semantics.

Traditionally, compiler construction is the main application area for syntax definition formalisms. The most well-known is the pair LEX/YACC. The formalism LEX (Lesk and Schmidt, 1986) is used to define the lexical syntax of a language using regular expressions. According to the regular expressions a string is analyzed and divided into tokens. In case more than one regular expression can be matched, a number of disambiguation rules such as prefer longest match and prefer regular expressions appearing earlier in the file. The 'compiler compiler' YACC (Johnson, 1975) is used for the definition of the context-free syntax of a language. An LALR(1) parser generator translates grammars to C programs if the grammar is LALR(1). Some conflicts in the parse table caused by ambiguous expression grammars can be solved by means of binary and unary precedences based on the ideas of Aho *et al.* (1975). Furthermore, the order of productions in the grammar is used to solve conflicts. Trees for a string can be constructed by calling C functions from the grammar productions.

Recent formalisms are generally based on the same deterministic parsing techniques, but extend the expressivity and declarativeness of syntax definition by providing mechanisms for building trees and coupling to other phases of compilation. Some examples are: The Cocktail compiler generator (Grosch, 1990) provides a BNF-like formalism with an LALR parser generator. The Eli system (Gray *et al.*, 1992) is a collection of tools for developing all aspects of compilers. The syntax definition formalism is based on context-free grammars. Attribute rules are added to define semantics computations. The tree transformation language TXL (Cordy and Carmichael, 1993) is a programming language for source to source transformations by means of transformation rules on parse trees. The syntax definition formalism of TXL is based on context-free grammars extended with some regular operators. Lexical syntax is defined by means of predefined lexical notions and by means of regular expressions over character classes. PCCTS (Parr and Quong, 1994) is a formalism based on top-down LL(1) parsing. The problems of unbounded lookahead are dealt with by means of backtracking and syntactic predicates that can be used to try out a variant before deciding which production to predict.

An application domain derived from compiler construction is the area of programming environments. A programming environment is a collection of tools for interactively developing and testing programs in some programming language. These tools are usually centered around an interactive syntax directed editor. A syntax editor has knowledge of the language of the programs being edited and provides support for checking the syntax of programs and for presenting and manipulating the structure of programs. In order to rapidly process changes to a program, incremental parsing and incremental compilation are used. Syntax definition formalisms developed for derivation of programming environments include the grammar formalism of the Synthesizer Generator (Reps and Teitelbaum, 1989), PSG (Bahlke and Snelting, 1986), METAL (Kahn *et al.*, 1983) and SDF (Heering *et al.*, 1989). The ASF+SDF Meta-Environment (Klint, 1993) is a programming environment for developing and generating programming environments from algebraic specifications. To speed up the development cycle for syntax definitions, incremental parser generation is used to only regenerate those parts of the parser that have been affected by a change.

Syntax definition in algebraic specification takes the form of grammars as algebraic signatures. The motivation here is to provide flexible notation for functions and constructors in abstract data type specifications and less the description of real programming languages. Therefore, the requirements on lexical syntax are not so strong. The correspondence of context-free grammars and many-sorted signatures was first described by Rus (see Hatcher and Rus, 1976). Goguen *et al.* (1977) showed that this correspondence could be used to define the semantics of programming languages. The correspondence was exploited in a number of algebraic specification formalisms to provide flexible, user-definable notation for functions and constructors. The first formalism to incorporate this was OBJ (Futatsugi *et al.*, 1985). Others are Cigale (Voisin, 1986), ASF+SDF (Heering *et al.*, 1989, Bergstra *et al.*, 1989a), the Meta Notation, used in action semantics (Mosses, 1992), and Elan (Vittek, 1994).

The combination of features that a formalism provides is, necessarily, rather arbitrary and strongly influenced by the expected application of definitions and the environment in which generated tools have to operate. Although it is not desirable to include all conceivable features in a formalism—some features can not be combined with others and too many features results in an unmanageable formalism—the similarities between different formalisms can be exploited by reusing parts of the design and implementation of old formalisms. However, formalisms are conventionally designed in a monolithic way, containing an intertwined mix of features, resulting in a formalism with a lack of orthogonality and uniformity that is difficult to implement, extend and use for other applications than the originally intended ones. Syntax definition formalisms form no exception to this rule.

Here we set out to design syntax definition formalisms in a modular way, as a family of formalisms each extending a small kernel with some feature for syntax definition. This approach should result in more orthogonal and uniform formalisms and should make it easier to (a) construct formalisms that use some subset of a set of known features, (b) adapt formalisms for use in other application areas, (c) implement tools for such formalisms and (d) design new formalisms that combine new features with existing ones.

As a first step to accomplish this goal we design a concrete formalism with a set of features that is useful in many application areas, but in particular in the application of grammars as signatures for algebraic specifications of programming languages. The result is the syntax definition formalism SDF2 that is a generalization of SDF. It incorporates several concepts and techniques introduced by Heering *et al.* (1989) in a more orthogonal and uniform way and adds several new features.

We use the algebraic specification formalism ASF+SDF to formally specify the family of syntax definition formalisms. For an introduction to ASF+SDF see Van Deursen *et al.* (1996).

In this chapter we outline the main features of SDF2 and examine the structure and design principles of the specification.

6.2 An Overview of SDF2

SDF2 is a syntax definition formalism based on context-free grammars, extended with character classes, sorts, literals, priorities, regular expressions, renamings, aliases and modules and combines the definition of lexical and context-free syntax into one formalism. The syntax definition in Figure 6.1 on page 121, taken from Visser (1997c), presents (the syntax of) a small untyped, first-order functional programming language, the data type environments and the evaluation function that interprets such functional programs using an instantiation of the environments data type. A program in this language might contain the following definition of a function `map` that applies a function `F` to all elements of a list `L`:

```
function map(F, L) is
         if(is-nil(L), nil(),
            cons(call(F, head(L)), map(F, tail(L))))
```

We sketch the main features of SDF2 and use the syntax definition in Figure 6.1 as running example.

6.2.1 Context-free Productions

The basis of the formalism is formed by *context-free productions*. These are rules of the form $\alpha \rightarrow \mathcal{A}$, where α is a list of symbols $\mathcal{A}_1 \ldots \mathcal{A}_n$ $(n \geq 0)$ and $\mathcal{A}$ a symbol. A production declares that a string of category $\mathcal{A}$ can be constructed by concatenating strings of the categories $\mathcal{A}_i$. For instance, the production

```
Fun "(" Terms ")" -> Term
```

defines that a term can be constructed by means of a function symbol followed by a list of terms separated by commas between parentheses. Conventionally, context-free productions are written as $\mathcal{A} \rightarrow \alpha$ or as $\mathcal{A} ::= \alpha$. In SDF2 productions are written the other way around to make the similarity to function declarations more apparent. This is useful because SDF2 definitions are used as signatures in algebraic specifications such that productions correspond to algebraic operators. For example, in a conventional signature one would declare the evaluation function that computes the value of a term with respect to a program and some environment by means of the function declaration

```
eval : Program # Term # Env -> Term
```

The production

```
eval "[[" Program "]]" "(" Term ")" "_" Env -> Term
```

not only defines a function with the same input types, but also the syntax for its applications, i.e., the program argument should be enclosed in double square brackets and the term argument should be enclosed in parentheses.

6.2.2 Character Clasess

Syntax definitions describe languages consisting of strings of *characters*, where the set of all characters can be encoded by a finite set of consecutive natural numbers. *Character Classes* are compact descriptions of sets of characters and are typically used in the definition of lexical categories such as layout, identifiers and numbers. The example contains the following character classes: the characters space, tab and newline `[\ \t\n]`, all characters except newline `~[\n]`, all uppercase letters `[A-Z]`, all lowercase letters `[a-z]`, all letters and digits and the hyphen character `[a-zA-Z0-9\-]`.

6.2.3 Literals

Literals are strings of characters between double quotes that stand for exactly that string of characters. These are used to represent keywords—such as `"function"` and `"program"`—and operators and other literal symbols—such as `"["`, `"|->"`, `"|>"`, and `"*"`. The definition of the function `eval` does not use quotes for the function name. This is an exception to the general rule: identifiers starting with a lowercase letter can also be used as literals.

6.2.4 Sorts

The basic nonterminal symbols used in productions are *sorts*, which are written as identifiers starting with a capital letter. Sorts should be declared in a sorts section. The example defines the sorts `Var`, `Fun`, `Term`, etc.

6.2.5 Regular Expressions

More complex nonterminal symbols can be formed by means of *regular expressions* that provide abbreviations for tupling (_), iteration _* and _+, optional constructs _?, and alternatives _|_. For example, `[a-zA-Z0-9\-]*` denotes a list of zero or more characters from the set of letters, digits and hyphens, `{Term ","}*` declares a list of terms separated by commas and the expression `(Key "|->" Value)*` denotes lists of zero or more tuples consisting of a key, the symbol `"|->"` and a value.

6.2.6 Aliases

Since such regular expressions can become quite tedious to type, it can be useful to introduce a shorter name for such symbols. This can be done by introducing a *symbol alias*. For example the declaration

```
aliases
  {Term ","}* -> Terms
```

introduces `Terms` as an alias for the regular expression `{Term ","}*`.

6.2.7 Priorities

Some productions that have a sensible type are syntactically ambiguous. For instance, the two productions for destructive and consistent environment update

```
Env "|>" Env -> Env
Env "*"  Env -> Env
```

are ambiguous with respect to themselves and to each other, e.g., the environment expression `Env |> Env * Env` can be constructed as `(Env |> Env) * Env` or as `Env |> (Env * Env)`. *Associativity* and *priority* declarations are a way to resolve most ambiguities of this type. In the example, the ambiguity in the expressions above is resolved by means of the priority declaration

```
module Functional-Programs
exports
  sorts Var Fun Term FunDef Program
  aliases
    {Term ","}* -> Terms
    {Var  ","}* -> Vars
  lexical syntax
    [\ \t\n]            -> LAYOUT
    "%%" ~[\n]*         -> LAYOUT
    [A-Z][a-zA-Z0-9\-]* -> Var
    [a-z][a-zA-Z0-9\-]* -> Fun
  context-free syntax
    Var               -> Term
    Fun "(" Terms ")" -> Term

    "function" Fun "(" Vars ")" "is" Term -> FunDef
    "program" FunDef*                     -> Program

module Environments
exports
  sorts Key Value Env
  context-free syntax
    "[" (Key "|->" Value)* "]" -> Env
    Env "(" Key ")"            -> Value
    Env "|>" Env               -> Env {left}
    Env "*" Env                -> Env {left}
    "(" Env ")"                -> Env {bracket}
  context-free priorities
    Env "|>" Env -> Env > Env "*" Env  -> Env

module Function-Eval
imports Functional-Programs
        Environments [Key => Var Value => Term]
exports
  context-free syntax
    eval "[[" Program "]]" "(" Terms ")" "_" Env -> Terms
```

Figure 6.1: SDF2 definition of the *syntax* of a small functional programming language and its main evaluation function.

```
  Env "|>" Env -> Env > Env "*" Env  -> Env
```

that declares `"|>"` to have higher priority than `"*"`, which entails the `(Env |> Env) * Env` interpretation. The `left` attribute of a production declares that the operator is left-associative.

6.2.8 Lexical and Context-free Syntax

The phrases making up a string over a language are usually divided into lexical tokens—the words of a sentence—and context-free phrases. The distinction between tokens and phrases is that the tokens making up a phrase can be separated by layout (whitespace and comments) while the characters comprising a token cannot. In definitions this distinction is indicated by means of *lexical* and *context-free* productions. For example, the lexical definition

```
[a-z][a-zA-Z0-9\-]* -> Fun
```

indicates that function symbols consist of a number of adjacent characters starting with a lowercase letter, followed by zero or more letters, digits or hyphens. While the tokens in the term `succ ( zero() )` can be separated by spaces, the characters in the token `succ` cannot. The layout that can occur between tokens should also be specified. The symbol `LAYOUT` is reserved for this purpose. In the example, layout is declared as

```
[\ \t\n]     -> LAYOUT
"%%" ~[\n]* -> LAYOUT
```

meaning that spaces, tabs and newlines (also called whitespace) are layout and that any suffix of a line starting with two percent signs is comment.

6.2.9 Modules

Grammars can be divided in a number of *modules* such that parts of a grammar can be reused in various language definitions. Modules consist of a list of exports and hiddens sections. An import of a module M into a module N denotes the inclusion of the exported grammar of M into N. Thus the import of module `Terms` in module `Functions` means that the syntax of terms is included in the syntax of programs. To prevent name clashes or to instantiate generic modules, *renamings* of symbols and productions can be applied to imported modules. For example, the module `Function-Eval` specifying the evaluation function, imports the generic module defining environments by means of

```
imports Environments [Key => Var Value => Term]
```

renaming the sort `Key` to `Var` and the sort `Value` to `Term`, thus instantiating it for use with the terms of the functional programming language.

Modules can also be parameterized with a list of parameter symbols that can be instantiated on import. For instance, module `Environments` might also have been declared as

```
module Environments[Key Value]
```

declaring `Key` and `Value` as parameters. The import

```
imports Environments[Var Term]
```

would then perform the instantiation.

A complete syntax definition consists of a list of modules and a designated top module. The language defined by such a definition is the one defined by the grammar associated to the top module. Of course, in a programming environment for SDF2 this list does not have to reside in a single file. More likely, each module will be defined in one file with the module name as the file name.

6.3 Design

The next chapters give a formal algebraic specification of the syntax and semantics of SDF2. The semantics of a syntax definition is characterized by the well-formed trees it generates. A tree is associated with a sentence—its *yield*. The language associated to a definition is the set of sentences that are yields of trees generated by the definition. A parser is a function that given a sentence, produces the tree (or set of trees) that have that sentence as yield. We do not describe parsing as part of the specification of SDF2, but specify the output required of a parser and allow any implementation that does so. Parsing for SDF2 grammars is described in Chapter 3.

6.3.1 Modularization

The formalism SDF2 is not designed monolithically, but modularized, as a family of formalisms. The kernel of this family is formed by context-free grammars. All features are defined as independent extensions of the kernel. The combination of the features forms SDF2. This setup makes it easier to construct a variant of the formalism by adding, removing or modifying features. Figure 6.2 depicts the structure of the family by means of (an abstraction of) the import graph of the specification.

Furthermore, the specification of SDF2 covers several aspects. The *syntax* of the formalism consists of the definition of the form of all its constructs. *Projection* functions on these constructs are defined in order to extract information from them. *Normalization* functions transform a syntax definition in order to simplify it. The specification of *parse trees* consists of several parts. A generic format for the representation of structured data called ATerms (Van den Brand *et al.*, 1997a) is used to represent parse trees. In order to use this format for a specific purpose, *constructor* names have to be defined. To represent grammar information in parse trees, several constructs of the formalism have to be *encoded as ATerms*. Given this framework, the well-formedness of a tree with respect to a grammar can be defined. Furthermore, the *yield* of trees and the *equality* of trees are defined.

For each feature a number of modules are defined that each define an aspect of the formalism for that feature. The result is the matrix of modules listed in Table 6.1. The rows of the matrix contain the modules for one feature. The columns of the matrix contain all modules for one aspect. Each module in the matrix has a name consisting of the name of the feature and the name of the aspect separated by -Sdf-. For instance module Kernel-Sdf-Syntax specifies the

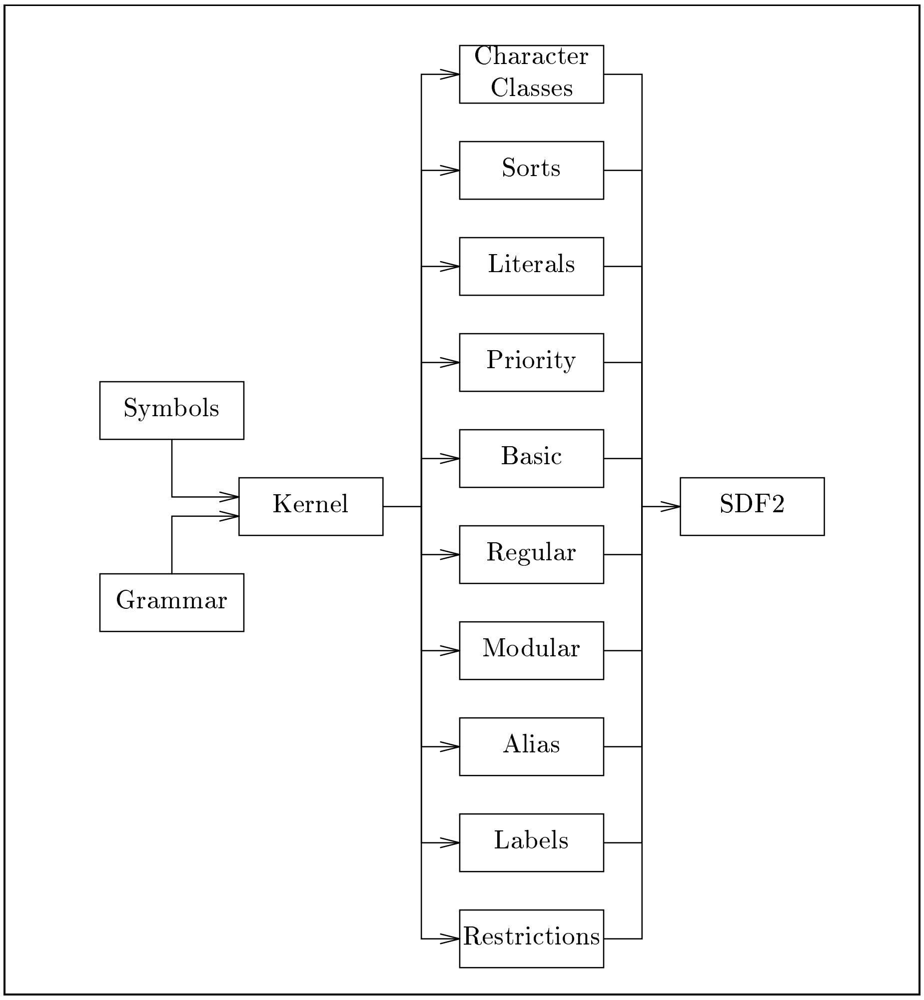


Figure 6.2: Import graph for the definition of SDF2.

syntax of the constructs in the kernel. So for each feature X we have modules X-Sdf-Syntax, X-Sdf-Projection, X-Sdf-Normalization, X-Sdf-Renamings, X-Sdf-Constructors, X-Sdf-ATerms, X-Sdf-Trees and X-Sdf-Equality. With the exception that if some feature does not change some aspect, the module is omitted.

6.3.2 Normalization

An important role in the design of SDF2 is played by the normalization function. In general, a normalization function defines a transformation on an expression that yields an expression in the same language, which uses less features. The

	Syntax	*Projection*	*Normalization*	*Renaming*	*Constructors*	*ATerms*	*Trees*	*Equality*
Symbols	7.1					7.5.3		
Grammar	7.2		7.3.3		A.2.1	A.2.2		
Kernel	7.3.1	7.3.2	7.3.3	9.1.3	7.5.2	7.5.4	7.5.5	7.5.8
Sorts	7.4.1	7.4.1	7.4.1	A.3	A.2.1	A.2.2		
CC	7.4.2		7.4.2	A.3	A.2.1	A.2.2	7.5.6	
Literals	7.4.3		7.4.3	A.3	A.2.1	A.2.2		
Priority	8.1.1	8.1.2	8.1.3	A.3		A.2.2	8.1.4	
Regular	8.2.1		8.2.2	A.3	A.2.1	A.2.2		
Basic	8.3.1		8.3.2	A.3	A.2.1	A.2.2		8.3.3
Restrictions	8.4.1	8.4.2	8.4.3	A.3				
Renaming	9.1.1	9.1.2		9.1.3			9.1.4	
Alias	9.2.1	9.2.2	9.2.3	9.2.3				
Modular	9.3.1	9.3.2	9.3.3	9.3.4				
Label								
Sdf2	10.1.1	A.4	10.1.2	A.4	A.4	A.4	A.4	A.4

Table 6.1: Modules of the family of syntax definition formalisms. The last row contains the collecting modules for SDF2. There are no collection modules for the rows. The numbers refer to the sections presenting the modules.

normalized expression has the same meaning as the original one. Thus, a normalization is a mapping from the language onto (a subset of) the same language. Ideally, a normalization function should be idempotent, i.e., yield the same result when applied twice. An implementation for such a language only has to consider the simplified expressions, while users have a more expressive language at their disposal.

The requirement that normalization produces an expression in the language itself entails that all constructs used for encodings should also be present in the original language, i.e., the language should be closed under normalization. For example, one of the normalizations in this definition renames a symbol $\mathcal{A}$ into $\langle \mathcal{A}$`-LEX`$\rangle$ if it occurs in the lexical syntax. Therefore, the constructor $\langle$`_-LEX`$\rangle$ introduced for the purpose of normalization also becomes a construct of the language before normalization.

A consideration in the definition of a normalization is whether two different expressions that are equivalent with respect to the semantics have the same normal form. This can be useful when expressions have to be compared. This is for example the case in the normalization of character classes. In Visser (1997b) a normalization of character classes to a unique normal forms is defined such that two character classes that represent the same set of characters are normalized to the same character class expression. In general, however, the normalizations in this chapter will not have this property. For instance, all permutations of a list of productions are equivalent. Although such lists could be ordered by imposing an ordering on productions, this is not done here, since comparisons of lists of productions are not needed. In such cases definitions can not use syntactic equality to determine equivalence.

Using this approach SDF2 is an expressive formalism that depends on a small set of features, i.e., we have:

$$
\begin{array}{rcl}
\text{SDF2} & = & \text{context-free grammars} \\
 & + & \text{priorities} \\
 & + & \text{character-classes} \\
 & + & \text{reject productions} \\
 & + & \text{follow restrictions}
\end{array}
$$

Features that are provided in the formalism, but that are eliminated, i.e., expressed using the features above are: literals, regular expressions, lexical and context-free syntax, variables, modules, renamings, and aliases. Furthermore, character classes, priority declarations and grammar composition are simplified considerably.

The normalization of SDF2 is defined as a pipeline of normalizations, as is illustrated in Figure 6.3. This modularization of the definition of normalization makes it easy to define an extension and express it in existing features using a new normalization function. The overall normalization is extended by adding the new function to the normalization pipeline.

6.4 Organization

The next chapters discuss the specification of the family of syntax definition formalisms that is the basis of SDF2. Chapter 7 defines context-free grammars, the basic symbols sorts, character classes and literals and defines the well-formed parse trees characterized by a grammar. Chapter 8 defines disambiguation by means of priorities, regular expressions, lexical and context-free syntax and restrictions for lexical disambiguation. Chapter 9 introduces renamings, aliases and modules. Chapter 10 these extensions are combined in the formalism SDF2. The formalism is compared to SDF and a discussion of possible improvements and extensions is given. Appendix A gives some auxiliary modules for the specification of SDF2.

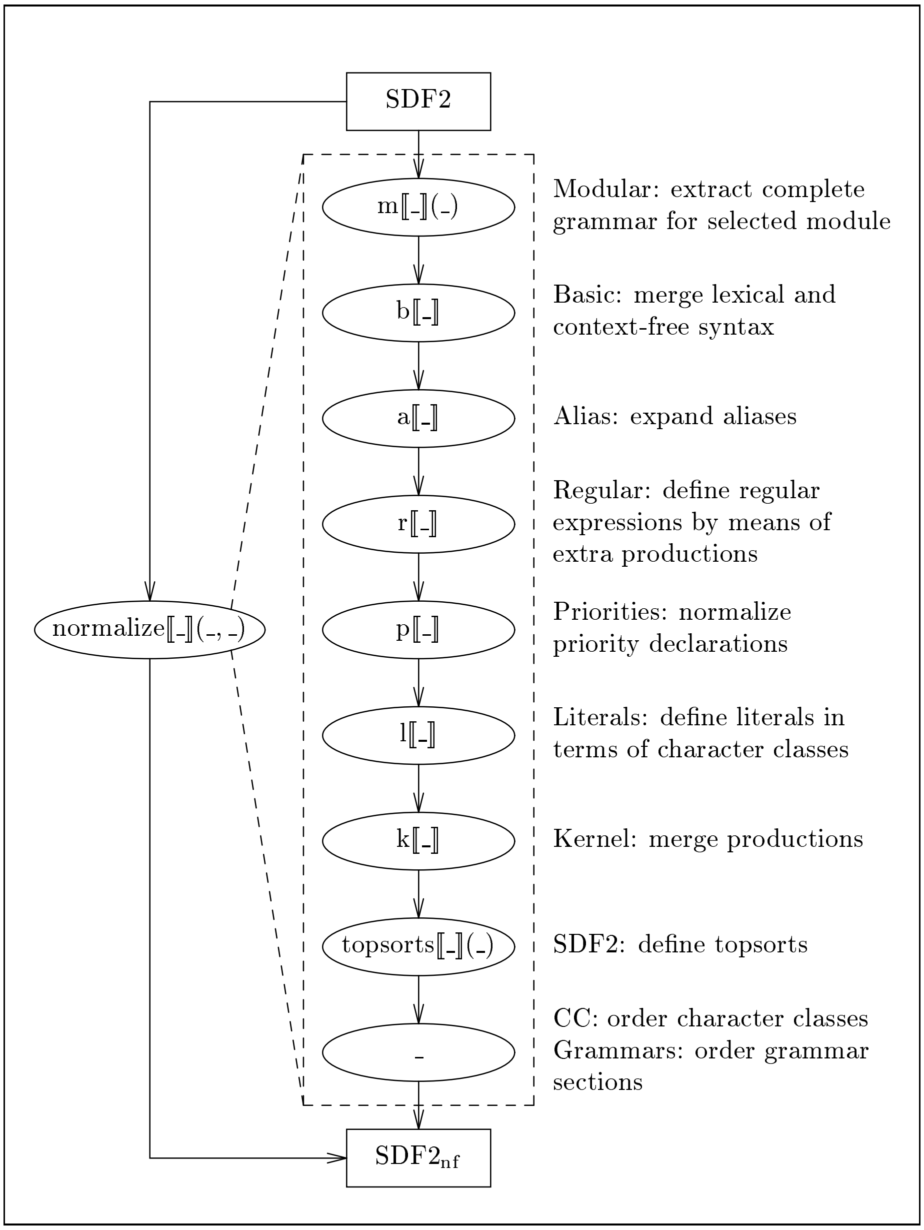


Figure 6.3: The normalization of SDF2 definitions consists of a series of independent transformations. The last step is not performed by a transformation function, but by rewrite rules acting on the constructors themselves.

7

Context-Free Grammars

In this chapter a context-free grammar formalism is defined. First an abstract framework of symbols and grammars is introduced. In this framework a grammar is interpreted by means of several predicates and functions that characterize the trees and strings of symbols generated by a grammar. A well-formedness predicate on parse trees characterizes the trees over a grammar. From the parse trees over a grammar the strings of the language defined by the grammar are derived. Parse trees are represented in the annotated term format ATerms. An instance of this framework is set up with as kernel context-free productions. We introduce three kinds of basic symbols to be used in productions: sorts, character classes and literals.

7.1 Symbols

Syntax definitions define languages, i.e., sets of strings of *symbols*. A string of symbols is a list of zero or more symbols. The sort Symbol is declared without actually specifying any constructors for it. This entails that the sort is empty at this point, but can be extended later on with constructors. We do not make a distinction between terminal and nonterminal symbols. Whether a symbol is a terminal or nonterminal symbol is determined by the interpretation and is not fixed syntactically. A symbol that plays the role of a terminal in one view can be a nonterminal in another view. An example is a literal that can be considered as a terminal token or as a nonterminal that is defined in terms of characters.

module Symbols
imports Layout
exports
  sorts Symbol Symbols
  context-free syntax
    "(" Symbol ")" → Symbol {**bracket**}
    Symbol∗ → Symbols
  variables
    $[\mathcal{ABC}][\mathit{0}\text{-}\mathit{9}']*$ → Symbol
    $[\alpha\beta\gamma][\mathit{0}\text{-}\mathit{9}']*$ → Symbol∗
    $[\alpha\beta\gamma]$"+"$[\mathit{0}\text{-}\mathit{9}']*$ → Symbol+

7.1.1 Projection

The function $+\!\!+$ concatenates strings and $|\_|$ gives the length of a string. The predicate $\in$ decides list membership.

module Symbols-Projection
imports Symbols$^{7.1}$ Booleans Integers
exports
  context-free syntax
    Symbols "$+\!\!+$" Symbols → Symbols {**assoc**}
    "|" Symbols "|" → Int
    Symbol "$\in$" Symbols → Bool
equations
Concatenation, length and membership of symbol lists.

[1] $\alpha +\!\!+ \beta = \alpha\ \beta$
[2] $|\ | = 0$
[3] $|\ \mathcal{A}\ \alpha\ | = |\ \alpha\ | + 1$
[4] $\mathcal{A} \in\ = \bot$
[5] $\mathcal{A} \in \mathcal{A}\ \alpha = \top$
[6] $\mathcal{A} \in \mathcal{B}\ \alpha = \mathcal{A} \in \alpha$ **when** $\mathcal{A} \neq \mathcal{B}$

The concatenation function on sorts such as Symbols is needed because the concatenation of the built-in associative lists (e.g., Symbol$*$) of ASF+SDF is not inherited through the injection Symbol$*$ → Symbol. The injection is needed because list sorts cannot be output sorts of functions in ASF+SDF.

7.1.2 Sets

From lists of symbols we construct sets of symbols by means of the operation $\{\_\}$. Although this constructor does not remove double elements from the list, it signifies that the number of occurrences in the list does not matter. Operations on sets are union ($\cup$), difference ($/$) and membership ($\in$). The union $\upsilon_1 \cup \upsilon_2$ adds only those elements of υ_1 to υ_2 that do not already occur in υ_2. If a set is constructed by means of union from singleton sets, the resulting set does not contain double elements. Although this is not strictly necessary it is useful when later on something has to be done once for each symbol in some set.

module Symbol-Sets
imports Symbols-Projection$^{7.1.1}$ Booleans
exports
  sorts SymbolSet
  context-free syntax
    "{" Symbols "}" → SymbolSet
    SymbolSet "$\cup$" SymbolSet → SymbolSet {**right**}
    SymbolSet "/" SymbolSet → SymbolSet {**left**}
    "(" SymbolSet ")" → SymbolSet {**bracket**}
    Symbol "$\in$" SymbolSet → Bool

priorities
SymbolSet "/"SymbolSet → SymbolSet >
SymbolSet "∪"SymbolSet → SymbolSet
variables
"v"[*0-9′*]∗ → SymbolSet
equations
Membership

[1] $$\mathcal{A} \in \{\alpha\} = \mathcal{A} \in \alpha$$

Union

[2] $$\{\} \cup v = v$$
[3] $$v \cup \{\} = v$$
[4] $$\{\alpha^+\ \beta^+\} \cup v = \{\alpha^+\} \cup \{\beta^+\} \cup v$$
[5] $$\{\mathcal{A}\} \cup v = v \quad \textbf{when}\ \ \mathcal{A} \in v = \top$$
[6] $$\{\mathcal{A}\} \cup \{\alpha\} = \{\mathcal{A}\ \alpha\} \quad \textbf{otherwise}$$

Difference

[7] $$\{\} \mathbin{/} v = \{\}$$
[8] $$v \mathbin{/} \{\} = v$$
[9] $$\{\alpha^+\ \beta^+\} \mathbin{/} v = \{\alpha^+\} \mathbin{/} v \cup \{\beta^+\} \mathbin{/} v$$
[10] $$\{\mathcal{A}\} \mathbin{/} v = \{\} \quad \textbf{when}\ \ \mathcal{A} \in v = \top$$
[11] $$\{\mathcal{A}\} \mathbin{/} v = \{\mathcal{A}\} \quad \textbf{otherwise}$$

7.2 Grammars

A syntax definition consists of a grammar. The only generic operations on grammars that we define at this point are an associative composition operation that is used to combine grammars and the constant ∅ representing the empty grammar.

module Grammar-Syntax
imports Layout
exports
sorts Grammar
context-free syntax
"∅" → Grammar
Grammar Grammar → Grammar {**assoc**}
"(" Grammar ")" → Grammar {**bracket**}
variables
"$\mathcal{G}$"[*0-9′*]∗ → Grammar

7.2.1 Interpretation

A grammar defines a set of strings of symbols, a language. We specify the language derived by a grammar indirectly, via the trees it generates. Parse trees will be represented by means of ATerms, a term format for the representation and exchange of structured data (Van den Brand *et al.*, 1997a). The format will be introduced in §7.5.1. In fact we will not just define parse trees, but parse forests. A parse forest is a compact encoding of a collection of parse trees in which contexts are shared. A parse forest is used to represent all parse trees for an ambiguous sentence.

The interpretation of a grammar is now given by two predicates and a function. The predicate $\mathcal{G} \vdash T$ characterizes the terms T that are *well-formed parse forests* over grammar $\mathcal{G}$. The function yield$[\![\mathcal{G}]\!](T)$ maps a parse tree T to a string of symbols. The predicate $T \mathrel{\dot{\in}} T'$ determines membership of a tree T in a forest T'. The set of trees generated by a grammar is denoted by $\mathcal{T}[\![\mathcal{G}]\!]$.

Given these functions we can derive the notion of the language generated by a grammar and the notion of a parser for a grammar. The *language* generated by a grammar corresponds to the set of yields of the parse trees it generates. In other words, a string of symbols α is an element of the language defined by a grammar $\mathcal{G}$, if there exists a well-formed tree T over the grammar with α as its yield. A *parser* $\Pi[\![\mathcal{G}]\!](\alpha)$ is a function that maps a string α in $\mathcal{L}[\![\mathcal{G}]\!]$ to a parse forest T containing all well-formed trees such that their yield is α. The module below summarizes these definitions. The equations define the set of parse trees $\mathcal{T}[\![\mathcal{G}]\!]$, the language $\mathcal{L}[\![\mathcal{G}]\!]$ and the parser $\Pi[\![\mathcal{G}]\!](\_)$ in terms of well-formedness, yield and parse forest membership.

module Grammar-Interpretation
imports Grammar-Syntax[7.2] Symbols[7.1] Booleans ATerms[7.5.1] Symbols-Sets ATerm-Sets
exports
context-free syntax

Grammar "⊢" ATerm	→ Bool
ATerm "$\dot{\in}$" ATerm	→ Bool
yield "⟦" Grammar "⟧" "(" ATerm ")"	→ Symbols
"T" "⟦" Grammar "⟧"	→ ATermList
"$\mathcal{L}$" "⟦" Grammar "⟧"	→ SymbolsSet
"Π" "⟦" Grammar "⟧" "(" Symbols ")"	→ ATerm

equations

[1]
$$\frac{\mathcal{G} \vdash T = \top}{T \in \mathrm{T}[\![\mathcal{G}]\!] \;=\; \top}$$

[2]
$$\frac{\mathcal{G} \vdash T = \top,\ \mathrm{yield}[\![\mathcal{G}]\!](T) = \alpha}{\alpha \in \mathcal{L}[\![\mathcal{G}]\!] \;=\; \top}$$

[3]
$$\frac{\mathcal{G} \vdash T = \top,\ \mathrm{yield}[\![\mathcal{G}]\!](T) = \alpha}{T \mathrel{\dot{\in}} \Pi[\![\mathcal{G}]\!](\alpha) \;=\; \top}$$

Note that these equations are non-constructive, i.e., do not provide decision procedures, but are merely a specification of the required behaviour. (Module Symbols-Sets defines sets of strings of symbols in a similar way as sets of symbols are defined in module Symbol-sets.)

7.2.2 Overview

In the rest of this chapter we will provide the specifications of well-formedness, yield and forest membership for a context-free grammar formalism. In §7.3 we define the syntax and normalization of context-free productions. In §7.4 we define basic symbols to be used in grammars: sorts, character classes and literals. In §7.5 we define the well-formed parse trees generated by a context-free grammar. In the next chapters this formalism is extended with a number of features.

An implementation of a parser is not specified, but can be chosen such as to comply with this specification. One possible implementation is discussed in Chapter 3.

7.3 Context-Free Grammars (Kernel)

7.3.1 Syntax

The kernel of SDF2 is formed by context-free grammars. A context-free production is a structure $\alpha \to \mathcal{A}$, where α is a list of symbols and $\mathcal{A}$ a symbol. A context-free grammar is formed by a list of productions preceded by the keyword `syntax`.

Conventionally, since their introduction by Chomsky (1956), context-free productions are written as $\mathcal{A} \to \alpha$ to emphasize the generative view of grammars. A grammar *generates* a string from a symbol, by repeatedly replacing some symbol in a string by the symbols on the right-hand side of a production. There exist many variants of this 'standard' notation, e.g., $\mathcal{A} ::= \alpha$ in BNF (Backus, 1959) and $\mathcal{A} : \alpha_1 | \ldots | \alpha_n;$ in YACC (Johnson, 1975).

The unconventional $\alpha \to \mathcal{A}$ notation for productions introduced by Heering *et al.* (1989) emphasizes the *functional* view of productions when used in the context of algebraic specification. A production coincides with the declaration of the name and type of a function. This notation is a unification of the definition of context-free productions with the declaration of mixfix functions in algebraic specification formalisms. For example, the declaration of the infix addition operator on natural numbers that is declared as `Nat ::= Nat "+" Nat` in BNF, is declared as `op _ + _ : nat nat -> nat .` in OBJ (Futatsugi *et al.*, 1985) and as `@ + @ : (nat nat) nat` in Elan (Vittek, 1994). In SDF this becomes `Nat "+" Nat -> Nat`.

All these notations are equivalent in expressive power and could be used instead of the current one. We could effortlessly define a version of SDF that uses the $\mathcal{A} ::= \alpha$ notation of BNF and define its meaning by translation to the notation used here. Note, however, that this does not mean that other aspects of these formalisms have the same expressive power nor that the parsing techniques coupled to these formalisms all have the same power.

Optionally, productions can have a list of attributes. An *attribute* is an annotation of a production that gives some extra syntactic or semantic information about the production. An example of an attribute that will be introduced in §8.1 is `left` that indicates left associativity of the production. Productions can have any number of attributes. The kernel does not provide any attributes, but to be able to introduce attributes later on without having to introduce an extra constructor for productions, the attribution of a production is defined here.

module Kernel-Sdf-Syntax
imports Symbols$^{7.1}$ Grammar-Syntax$^{7.2}$
exports
  sorts Attribute Attributes Production Productions
  context-free syntax

"{" {Attribute ","}∗ "}"	→ Attributes
	→ Attributes
Symbols "→" Symbol Attributes	→ Production
Production∗	→ Productions
"syntax" Productions	→ Grammar

  variables

"*attr*"[*0-9′*]∗	→ Attribute
"*attr*""∗"[*0-9′*]∗	→ {Attribute ","}∗
"*attr*""+"[*0-9′*]∗	→ {Attribute ","}+
"$"[*0-9′*]∗	→ Attributes
[*p*][*0-9′*]∗	→ Production
[*p*]"∗"[*0-9′*]∗	→ Production∗
[*p*]"+"[*0-9′*]∗	→ Production+

7.3.2 Projection

We define concatenation functions for lists of productions and lists of attributes. The concatenation function for attributes removes duplicates. A production with an empty list of attributes is equal to a production without attributes. The projection function 'P' gives the productions of a grammar, the function '$\overline{\mathrm{P}}$' gives the non-production parts of a grammar (to be defined later), and the function '$\mathrm{P}_{\mathcal{A}}$' gives all productions defining a symbol $\mathcal{A}$. The function 'symbols' gives the set of all symbols in a grammar. The function 'reachable' gives all productions reachable from some set of symbols, i.e., used in the definition of those symbols.

module Kernel-Sdf-Projection
imports Kernel-Sdf-Syntax$^{7.3.1}$ Symbol-Sets$^{7.1.2}$
exports
context-free syntax

Productions "++" Productions	→ Productions	{**right**}
Attributes "++" Attributes	→ Attributes	{**right**}
Production "∈" Productions	→ Bool	
Productions "⊆" Productions	→ Bool	
Production "≅" Production	→ Bool	
"P"(Grammar)	→ Productions	
"$\overline{\text{P}}$" "(" Grammar ")"	→ Grammar	
"P" "_" Symbol "(" Productions ")"	→ Productions	
symbols(Productions)	→ SymbolSet	
symbols(Grammar)	→ SymbolSet	
reachable(SymbolSet, SymbolSet, Productions)	→ Productions	
reachable(SymbolSet, Grammar)	→ Grammar	

equations

Concatenation of lists of productions, membership and subset of a list of productions.

[1] $$p_1^* \mathbin{+\!\!+} p_2^* = p_1^*\ p_2^*$$
[2] $$p \in p_1^*\ p\ p_2^* = \top$$
[3] $$p \in p^* = \bot \quad \textbf{otherwise}$$
[4] $$\subseteq p^* = \top$$
[5] $$p\ p_1^* \subseteq p_2^* = p \in p_2^* \wedge p_1^* \subseteq p_2^*$$

Concatenation of attribute lists. Attributes ocurring in both lists are added only once.

[6] $$\{attr_1^+,\ attr_2^+\} \mathbin{+\!\!+} \{attr^*\} = \{attr_1^+\} \mathbin{+\!\!+} \{attr_2^+\} \mathbin{+\!\!+} \{attr^*\}$$
[7] $$\{attr\} \mathbin{+\!\!+} \{attr_1^*,\ attr,\ attr_2^*\} = \{attr_1^*,\ attr,\ attr_2^*\}$$
[8] $$\{attr\} \mathbin{+\!\!+} \{attr^*\} = \{attr,\ attr^*\} \quad \textbf{otherwise}$$
[9] $$\$ \mathbin{+\!\!+} \ = \$$$
[10] $$\mathbin{+\!\!+} \$ = \$$$
[11] $$\{\} =$$

The last equation states that an empty list of attributes {} is equal to no attributes.

Two productions are similar if they are the same up to their attributes

[12] $$\alpha \to \mathcal{A}\ \$_1 \cong \alpha \to \mathcal{A}\ \$_2 = \top$$
[13] $$p_1 \cong p_2 = \bot \quad \textbf{otherwise}$$

Function 'P' gives all productions of a grammar and function '$\overline{\mathrm{P}}$' gives all non-syntax parts of a grammar. The function $\mathrm{P}_{\mathcal{A}}$ gives all productions defining the symbol $\mathcal{A}$.

[14] $\mathrm{P}(\text{syntax } p^*) = p^*$

[15] $\mathrm{P}(\mathcal{G}_1\ \mathcal{G}_2) = \mathrm{P}(\mathcal{G}_1) +\!\!+\ \mathrm{P}(\mathcal{G}_2)$

[16] $\mathrm{P}(\mathcal{G}) = \quad$ **otherwise**

[17] $\overline{\mathrm{P}}(\text{syntax } p^*) = \emptyset$

[18] $\overline{\mathrm{P}}(\mathcal{G}_1\ \mathcal{G}_2) = \overline{\mathrm{P}}(\mathcal{G}_1)\ \overline{\mathrm{P}}(\mathcal{G}_2)$

[19] $\overline{\mathrm{P}}(\mathcal{G}) = \mathcal{G}$ **otherwise**

[20] $\mathrm{P}_{\mathcal{A}}() =$

[21] $\mathrm{P}_{\mathcal{A}}(\alpha \to \mathcal{A}\ \$\ p^*) = \alpha \to \mathcal{A}\ \$\ +\!\!+\ \mathrm{P}_{\mathcal{A}}(p^*)$

[22] $\mathrm{P}_{\mathcal{A}}(p\ p^*) = \mathrm{P}_{\mathcal{A}}(p^*)$ **otherwise**

The function 'symbols' gives the set of symbols of a list of productions or a grammar.

[23] $\text{symbols}() = \{\}$

[24] $\text{symbols}(\alpha \to \mathcal{A}\ \$\ p^*) = \{\alpha\} \cup \{\mathcal{A}\} \cup \text{symbols}(p^*)$

[25] $\text{symbols}(\text{syntax } p^*) = \text{symbols}(p^*)$

[26] $\text{symbols}(\mathcal{G}_1\ \mathcal{G}_2) = \text{symbols}(\mathcal{G}_1) \cup \text{symbols}(\mathcal{G}_2)$

[27] $\text{symbols}(\mathcal{G}) = \{\}$ **otherwise**

The function 'reachable' gives the subgrammar with those productions reachable from some set of symbols υ. It is defined by applying the auxiliary 'reachable' function to the productions of the grammar. Observe how the resulting grammar is a composition of the reachable productions and the non-production parts of a grammar. The auxiliary function selects for each symbol in the original set the productions for that symbol from the original grammar. This is applied recursively to the symbols used in the left-hand sides of those productions. The first set argument of the auxiliary function represents the symbols already handled. The second set contains the symbols for which the productions still have to be looked up.

[28] $\text{reachable}(\upsilon, \mathcal{G}) \ = \ \text{syntax reachable}(\{\}, \upsilon, \mathrm{P}(\mathcal{G}))\ \overline{\mathrm{P}}(\mathcal{G})$

[29] $\text{reachable}(\upsilon, \{\}, p^*) \ =$

[30] $$\frac{\mathcal{A} \in \upsilon_1 = \bot,\ \ \mathrm{P}_{\mathcal{A}}(p_1^*) = p_2^*,\ \ \text{symbols}(p_2^*)\ /\ \upsilon_1 = \upsilon_2}{\text{reachable}(\upsilon_1, \{\mathcal{A}\ \alpha\}, p_1^*) \ = \ p_2^* +\!\!+ \text{reachable}(\upsilon_1 \cup \{\mathcal{A}\}, \upsilon_2 \cup \{\alpha\}, p_1^*)}$$

[31] $\text{reachable}(\upsilon_1, \{\mathcal{A}\ \alpha\}, p^*) \ = \ \text{reachable}(\upsilon_1, \{\alpha\}, p^*)$
otherwise

7.3.3 Normalization

Grammar Normalization Composition of grammars is commutative and the empty grammar is a unit for grammar composition. Since commutativity cannot be expressed by means of a terminating rewrite system, the following module normalizes grammar compositions as a right associative list, where the grammars are ordered or merged as specified by the operation $\diamond$. If $\mathcal{G}_1 \diamond \mathcal{G}_2$ yields a pair $\langle \mathcal{G}_3, \mathcal{G}_4 \rangle$ this means that the composition $\mathcal{G}_1 \mathcal{G}_2$ should be replaced by $\mathcal{G}_3 \mathcal{G}_4$. The definition of $\diamond$ can either merge the two grammars into one, yielding the pair $\langle \mathcal{G}_3, \emptyset \rangle$, or exchange the grammars yielding $\langle \mathcal{G}_2, \mathcal{G}_1 \rangle$. The termination of this normalization depends on the property of $\diamond$ that a swap cannot be undone. The definition of $\diamond$ should be extended for each new grammar constructor. For example, the merging of the productions of two adjacent syntax sections is expressed in the next module. The ordering could also be defined directly on the grammar composition operator, but that would entail that two equations would have to be written for each pair of constructors that have to be merged or swapped, corresponding to the last two equations below.

module Grammar-Normalization
imports Grammar-Syntax$^{7.2}$
exports
  sorts Grammar-Grammar
  context-free syntax
    Grammar "◇" Grammar → Grammar-Grammar
    "<" Grammar "," Grammar ">" → Grammar-Grammar
equations
The empty grammar $\emptyset$ is a unit for composition and composition is associative.

[1] $$\emptyset\ \mathcal{G} = \mathcal{G}$$
[2] $$\mathcal{G}\ \emptyset = \mathcal{G}$$
[3] $$\mathcal{G}_1\ (\mathcal{G}_2\ \mathcal{G}_3) = \mathcal{G}_1\ \mathcal{G}_2\ \mathcal{G}_3$$

Subgrammars can be swapped as specified by the function $\diamond$.

[4] $$\mathcal{G}_1\ \mathcal{G}_2 = \mathcal{G}_1'\ \mathcal{G}_2' \quad \textbf{when} \ \mathcal{G}_1 \diamond \mathcal{G}_2 = \langle \mathcal{G}_1', \mathcal{G}_2' \rangle$$
[5] $$\mathcal{G}_1\ \mathcal{G}_2\ \mathcal{G}_3 = \mathcal{G}_1\ \mathcal{G}_2'\ \mathcal{G}_3' \quad \textbf{when} \ \mathcal{G}_2 \diamond \mathcal{G}_3 = \langle \mathcal{G}_2', \mathcal{G}_3' \rangle$$

Context-free Grammar Normalization The normalization function $\mathrm{k}[\![\_]\!]$ for the kernel, merges productions with the same arguments and result symbols. If such productions have different attributes, these are joined. This normalization entails that two occurrences of the same production are identified and do therefore not cause an ambiguity. Consequently, other normalization functions can generate a production more than once, without changing the meaning of the grammar. This strategy will be relevant later on when we introduce modularization of grammars. The identification of productions means that a production that is declared in two or more different modules is identified when these modules are imported in the same module.

module Kernel-Sdf-Normalization
imports Kernel-Sdf-Projection[7.3.2] Grammar-Normalization[7.3.3]
exports
  context-free syntax
    "k" "⟦" Grammar "⟧" → Grammar
    merge(Productions)   → Productions
equations

An empty list of productions is equivalent to an empty grammar and multiple syntax sections are merged into one.

$$[1] \qquad \text{syntax} = \emptyset$$

$$[2] \qquad \text{syntax } p_1^* \diamond \text{syntax } p_2^* = \langle \text{syntax } p_1^* \ p_2^*, \emptyset \rangle$$

The normalization function 'k' merges productions with the same arguments and result, using the auxiliary function 'merge'.

$$[3] \qquad \mathrm{k}[\![\mathcal{G}]\!] \ = \ \text{syntax merge}(\mathrm{P}(\mathcal{G})) \ \overline{\mathrm{P}}(\mathcal{G})$$

$$[4] \qquad \frac{p_1 = \alpha \to \mathcal{A}\ \$_1, \ \ p_2 = \alpha \to \mathcal{A}\ \$_2, \ \ \$_1 +\!\!+ \$_2 = \$_3, \ \ p_3 = \alpha \to \mathcal{A}\ \$_3}{\text{merge}(p_1^* \ p_1 \ p_2^* \ p_2 \ p_3^*) \ = \ \text{merge}(p_1^* \ p_3 \ p_2^* \ p_3^*)}$$

$$[5] \qquad \text{merge}(p^*) \ = \ p^*$$
otherwise

7.4 Basic Symbols

The kernel formalism presented in the previous section is a complete definition of context-free grammars, except for the notation of symbols. In this section we present three extensions of the kernel that provide notation for basic symbols needed in syntax definition. Sorts represent the non-terminals of grammars, the categories or domains that the grammar introduces. Character classes are used to represent the terminals of grammars, the characters from which strings are built. Literals are convenient abbreviations for fixed strings of characters. With these extensions we will have a complete notation for context-free grammars. The extensions in later sections will provide features to make this formalism more expressive.

7.4.1 Sorts

Syntax Sorts are the symbols that represent the basic domains or categories of a syntax definition. A sort identifier is a word starting with an uppercase letter followed by zero or more letters or digits. Hyphens can be used between the first and last character. Sorts used in the productions of a grammar should be declared in a separate sorts section that consists of the keyword 'sort' and a list of symbols.

module Sorts-Sdf-Syntax
imports Kernel-Sdf-Syntax[7.3.1]
exports
  sorts Sort
  lexical syntax
    `[A-Z]` $\rightarrow$ Sort
    `[A-Z][A-Za-z0-9\-]*[A-Za-z0-9]` $\rightarrow$ Sort
  context-free syntax
    Sort $\rightarrow$ Symbol
    "sorts" Symbols $\rightarrow$ Grammar
  variables
    "$\mathcal{S}$"[*0-9'*]$*$ $\rightarrow$ Sort

Normalization Ordering of sorts and syntax sections (sorts are placed before syntax sections) and merging of sorts sections.

module Sorts-Sdf-Normalization
imports Sorts-Sdf-Syntax[7.4.1] Kernel-Sdf-Normalization[7.3.3]
equations

[1] $$\text{sorts} = \emptyset$$
[2] $$\text{sorts } \alpha \diamond \text{sorts } \beta = \langle \text{sorts } \alpha\ \beta,\ \emptyset \rangle$$
[3] $$\text{syntax } p^* \diamond \text{sorts } \alpha = \langle \text{sorts } \alpha,\ \text{syntax } p^* \rangle$$

Projection The projection function 'S' gives the list of sorts of a grammar.

module Sorts-Sdf-Projection
imports Kernel-Sdf-Projection[7.3.2] Sorts-Sdf-Syntax[7.4.1]
exports
  context-free syntax
    "S"(Grammar) $\rightarrow$ Symbols
equations
The declared sorts of a grammar.

[1] $$\text{S}(\text{sorts } \alpha) = \alpha$$
[2] $$\text{S}(\mathcal{G}_1\ \mathcal{G}_2) = \text{S}(\mathcal{G}_1) +\!\!+ \text{S}(\mathcal{G}_2)$$
[3] $$\text{S}(\mathcal{G}) = \quad \textbf{otherwise}$$

7.4.2 Character Classes

A character class is an expression such as, for example, `[a-z\']` that denotes a set of characters, in this case the set of all lower case letters and a prime. For example, the following definition defines identifiers as lists of characters starting with a lowercase letter followed by zero or more lowercase letters or digits.

```
sorts Id
syntax
  [a-z]        -> Id
  Id [a-z0-9] -> Id
```

The meaning of character classes could be defined in terms of productions and characters, effectively eliminating them from the formalism. For instance, the character class `[a-z]` is completely defined by 26 productions of the form:

```
[a] -> [a-z]   [b] -> [a-z]      ...       [z] -> [a-z]
```

However, this would cause an enormous increase in the number of productions. Therefore, the interpretation of character classes is not defined by translating character classes out of the language. This means that interpretation functions should be extended to character classes.

We do not give the complete specification of character classes and character class arithmethic. A full specification of character classes can be found in Visser (1997b). The normalization defined there ensures that two classes that contain the same elements have the same normal form.

Characters A character is a constant of the form $\backslash d_1 \dots d_n$, where the d_i are decimal digits, denoting the $d_1 \dots d_n$-th member of some finite, linearly ordered universe of characters. Since specifying characters by their index in some encoding scheme is difficult, we provide easier syntax for specification of characters. Alphanumeric characters (letters and digits) can be specified as themselves. Other visible characters in the ASCII set can be specified by escaping them using a backslash, e.g., `\(` for left parenthesis, `\-` for a hyphen and `\ ` (a backslash followed by a space) for space. The characters `\t` and `\n` represent tabs and newlines. Finally, there are two special characters, `\EOF` and `\TOP`. `\EOF` is the character used to indicate represent the end of a file. `\TOP` is used to represent the largest character in the character universe.

module Character-Syntax
imports Layout
exports
  sorts Character NumChar ShortChar
  lexical syntax
    [\][0-9]+ → NumChar
    [a-zA-Z0-9] → ShortChar
    [\]~[\000-\037A-Za-mo-su-z0-9] → ShortChar
  context-free syntax
    NumChar → Character
    ShortChar → Character
    "\TOP" → Character
    "\EOF" → Character
  variables
    "c"[*0-9′*]∗ → Character

Character Classes A set of characters—a character class—is represented by a list of characters and character ranges between square brackets [and]. A list is constructed by an injection of characters into lists and by a right associative binary concatenation operator on lists. Operations on character classes are difference (/), intersection ($\wedge$), union ($\vee$) and complement with respect to the complete character set, i.e., the characters in the range `\0-\TOP`, ($\sim$).

module Character-Class-Syntax
imports Character-Syntax[7.4.2]
exports
  sorts CharRange CharRanges OptCharRanges CharClass
  context-free syntax

Character	$\rightarrow$ CharRange	
Character "$-$" Character	$\rightarrow$ CharRange	
CharRange	$\rightarrow$ CharRanges	
CharRanges CharRanges	$\rightarrow$ CharRanges	{**right**}
"(" CharRanges ")"	$\rightarrow$ CharRanges	{**bracket**}
	$\rightarrow$ OptCharRanges	
CharRanges	$\rightarrow$ OptCharRanges	
"[" OptCharRanges "]"	$\rightarrow$ CharClass	
"$\sim$" CharClass	$\rightarrow$ CharClass	
CharClass "/" CharClass	$\rightarrow$ CharClass	{**left**}
CharClass "$\wedge$" CharClass	$\rightarrow$ CharClass	{**left**}
CharClass "$\vee$" CharClass	$\rightarrow$ CharClass	{**left**}
"(" CharClass ")"	$\rightarrow$ CharClass	{**bracket**}

  priorities
    "$\sim$"CharClass $\rightarrow$ CharClass $>$ CharClass "/"CharClass $\rightarrow$ CharClass $>$ CharClass "$\wedge$"CharClass $\rightarrow$ CharClass $>$ CharClass "$\vee$"CharClass $\rightarrow$ CharClass
  variables

"*cr*"[*0-9'*]$*$	$\rightarrow$ CharRange
"*cr*""$*$"[*0-9'*]$*$	$\rightarrow$ OptCharRanges
"*cr*""+"[*0-9'*]$*$	$\rightarrow$ CharRanges
"*cc*"[*0-9'*]$*$	$\rightarrow$ CharClass

Syntax The kernel formalism is extended by adding character classes as symbols.

module CC-Sdf-Syntax
imports Character-Class-Syntax[7.4.2] Kernel-Sdf-Syntax[7.3.1]
exports
  context-free syntax
    CharClass $\rightarrow$ Symbol

Normalization Character classes can be normalized to a unique normal form by ordering the ranges such that all characters are translated to their numeric equivalent and such that smaller characters are before larger characters and by fusing adjacent or overlapping ranges. For example, the class `[A-Z0-9\%]` has normal form `[\37\48-\57\65-\90]`, because `\37` is the numerical representation of `\%`, `\48-\57` of `0-9`, `\65-\90` of `A-Z` and these do not overlap and are ordered. This normalization is specified in module Character-Class-Normalization that can be found in Visser (1997b).

module CC-Sdf-Normalization
imports CC-Sdf-Syntax[7.4.2] Character-Class-Normalization
Kernel-Sdf-Normalization[7.3.3]

7.4.3 Literals

Literals are abbreviations for fixed lists of characters. For example, the following production uses literals to define the keywords of a conditional statement.

```
"if" Exp "then" Stat "else" Stat -> Stat
```

The meaning of literals is expressed by means of a production that specifies the sequence of characters that makes up the literal. For instance, the meaning of the literals above is expressed by the productions

```
[\105] [\102]               -> "if"
[\116] [\104] [\101] [\110] -> "then"
[\101] [\108] [\115] [\101] -> "else"
```

Literals that are identifiers starting with a lowercase letter can be specified without the double quotes.

Another useful abbreviation in this category is the definition of the syntax of prefix functions in the form

```
add(Nat, Nat) -> Nat
```

as an abbreviation of

```
"add" "(" Nat "," Nat ")" -> Nat
```

Syntax Literals consist of a list of characters between double quotes. For the complete syntax of literals see §A.1. Literals that start with a lowercase letter can be written without quotes, hence the name 'unquoted literals'. Prefix functions can be declared by means of a special form of productions, where the double quotes for the parentheses and commas can be omitted.

module Literals-Sdf-Syntax
imports Kernel-Sdf-Syntax[7.3.1] Literals[A.1]
exports
  sorts UQLiteral
  lexical syntax
    [a-z] → UQLiteral
    [a-z][A-Za-z0-9\-]*[A-Za-z0-9] → UQLiteral

context-free syntax

UQLiteral	$\rightarrow$ Literal
Literal	$\rightarrow$ Symbol
Literal "(" {Symbol ","}* ")" "$\rightarrow$" Symbol Attributes	$\rightarrow$ Production

Normalization The normalization function 'l' generates a defining production for each literal that is used as a symbol in one of the productions of the grammar.

module Literals-Sdf-Normalization
imports Literals-Sdf-Syntax[7.4.3] CC-Sdf-Normalization[7.4.2]
exports
context-free syntax

"l⟦" Grammar "⟧"	$\rightarrow$ Grammar
literals(SymbolSet)	$\rightarrow$ Productions
chars(Literal)	$\rightarrow$ Symbols
symbols({Symbol ","}*)	$\rightarrow$ Symbols

variables

"c"[*0-9'*]*	$\rightarrow$ CHAR
"c" "+"[*0-9'*]*	$\rightarrow$ CHAR+
"c" "*"[*0-9'*]*	$\rightarrow$ CHAR*
"L"	$\rightarrow$ Literal
"γ"[*0-9'*]*	$\rightarrow$ {Symbol ","}*
"γ" "+"[*0-9'*]*	$\rightarrow$ {Symbol ","}+

equations
Unquoted literals are translated to quoted literals.

[1] $$\text{uqliteral}(c^+) = \text{literal}(\texttt{"\textbackslash""}\ c^+\ \texttt{"\textbackslash""})$$

The function l⟦_⟧ generates a production for each literal symbol in the grammar. The production generated for a literal L has the form $\alpha \rightarrow L$, where α is a list of singleton character classes representing the characters of L. This list is produced by the function 'chars'.

[2] $$\mathrm{l}[\![\mathcal{G}]\!] = \mathcal{G}\ \text{syntax literals}(\text{symbols}(\mathcal{G}))$$
[3] $$\text{literals}(\{L\}) = \text{chars}(L) \rightarrow L$$
[4] $$\text{literals}(\{\mathcal{A}\}) = \quad \textbf{otherwise}$$
[5] $$\text{literals}(\{\}) =$$
[6] $$\text{literals}(\{\alpha^+\ \beta^+\}) = \text{literals}(\{\alpha^+\}) + \!\!\! + \text{literals}(\{\beta^+\})$$

The function 'chars' scans the characters in the literal string, translating them to short characters. These are then normalized to numeric character codes by character normalization. The third equation tries if the first character of the string is a short-character by normalizing it and then testing whether it has reduced to a numeric character. This works for letters and digits. If this fails, the fourth equation translates the character to an escaped short-character, which succeeds for all other characters. Characters that are already escaped are handled by the second equation.

[7] $$\text{chars}(\texttt{""}) \ =$$

[8] $$\frac{[\text{shortchar}(\texttt{"\textbackslash"}\ c)] = cc}{\text{chars}(\text{literal}(\texttt{"""}\ \texttt{"\textbackslash"}\ c\ c^*\ \texttt{"""})) \ = \ cc \mathbin{+\!\!+} \text{chars}(\text{literal}(\texttt{"""}\ c^*\ \texttt{"""}))}$$

[9] $$\frac{[\text{shortchar}(c)] = cc,\ \ cc = [\text{numchar}(c^+)]}{\text{chars}(\text{literal}(\texttt{"""}\ c\ c^*\ \texttt{"""})) \ = \ cc \mathbin{+\!\!+} \text{chars}(\text{literal}(\texttt{"""}\ c^*\ \texttt{"""}))}$$

[10] $$\frac{\begin{array}{c}[\text{shortchar}(\texttt{"\textbackslash"}\ c)] = cc, \\ \alpha = \text{chars}(\text{literal}(\texttt{"""}\ c^*\ \texttt{"""}))\end{array}}{\text{chars}(\text{literal}(\texttt{"""}\ c\ c^*\ \texttt{"""})) \ = \ cc \mathbin{+\!\!+} \alpha} \quad \textbf{otherwise}$$

Prefix function productions are translated to normal productions by enclosing the parentheses and commas in double quotes.

[11] $$L(\gamma) \to \mathcal{A}\ \$ = L\ \texttt{"("} \mathbin{+\!\!+} \text{symbols}(\gamma) \mathbin{+\!\!+} \texttt{")"} \to \mathcal{A}\ \$$$
[12] $$\text{symbols}() =$$
[13] $$\text{symbols}(\mathcal{A}) = \mathcal{A}$$
[14] $$\text{symbols}(\gamma_1^+,\ \gamma_2^+) = \text{symbols}(\gamma_1^+) \mathbin{+\!\!+} \texttt{","} \mathbin{+\!\!+} \text{symbols}(\gamma_2^+)$$

7.5 Parse Trees

Now we can define the interpretation of grammars, that is, the well-formed trees characterized by a grammar and the yield of those trees. The general idea is that a context-free production $p = \mathcal{A}_1 \ldots \mathcal{A}_n \to \mathcal{A}_0$ constructs trees of type $\mathcal{A}_0$ labeled with the production p and with a list of direct descendants of type $\mathcal{A}_1 \ldots \mathcal{A}_n$. Such trees are represented by means of terms. The constructor 'appl' builds an application of a production p to a list of trees, i.e., if $T_1 \ldots T_n$ are trees of type $\mathcal{A}_1 \ldots \mathcal{A}_n$ then $\text{appl}(p, [T_1, \ldots, T_n])$ is a tree of type $\mathcal{A}_0$. Parse forests are constructed by representing choice nodes or *ambiguity nodes* by means of the constructor 'amb'. If $T_1 \ldots T_n$ are all trees of the same type $\mathcal{A}$, then $\text{amb}([T_1, \ldots, T_n])$ is an ambiguity node of type $\mathcal{A}$.

To formally define this notion of trees we introduce the notion of terms. We first present the generic term format that is used to encode parse trees and the encoding of symbols and grammars in that format. With those tools in place, we define the well-formedness rules in §7.5.5.

7.5.1 Term Format

Van den Brand *et al.* (1997a) introduce the generic, annotated term format ATerms for the representation and exchange of structured data. The format is designed such that all kinds of data can be represented in a single, fixed format, with the purpose of exchanging such data between tools and providing generic operations on these data. The definition of the format comes with an extensive library of (higher-order) functions. We will use the ATerm format to represent parse trees.

The syntax of *ATerms* is defined in module ATerms below. Terms are constructed by means of four constructors, i.e., an ATerm is one of the following:

— A *constant* (ACon), which is either an integer constant or a real number constant.

— A *list* of terms (ATermList), which is either empty [], or a list of one or more terms separated by commas between square brackets $[T_1, \ldots, T_n]$. The sort ATerms represents lists of one or more terms separated by commas.

— A *function symbol* (AFun).

— An *application* of a function symbol to a list of one or more terms separated by commas.

Furthermore, each of these constructors can be *annotated* by a list of one or more terms between { and } (Ann). Literals are strings of characters between double quotes. Integer constants are lists of digits and real constants are floating point numbers with an optional exponent. For the syntax of literals, integers and reals see Van den Brand *et al.* (1997a).

module ATerms
imports Literals$^{A.1}$ IntCon RealCon
exports
  sorts ATerms ATermList ACon AFun ATerm Ann
  context-free syntax

ATerm	→ ATerms
ATerm "," ATerms	→ ATerms
"[" "]"	→ ATermList
"[" ATerms "]"	→ ATermList
IntCon	→ ACon
RealCon	→ ACon
Literal	→ AFun
ACon	→ ATerm
ATermList	→ ATerm
AFun	→ ATerm
AFun "(" ATerms ")"	→ ATerm
"{" ATerms "}"	→ Ann
ACon Ann	→ ATerm
ATermList Ann	→ ATerm
AFun Ann	→ ATerm
AFun "(" ATerms ")" Ann	→ ATerm

  variables

"*Ts*"[*0-9'*]*	→ ATerms
"*Tl*"[*0-9'*]*	→ ATermList
"*ACon*"[*0-9'*]*	→ ACon
"*AFun*"[*0-9'*]*	→ AFun

“*T*”[*0-9′*]* → ATerm
“*Ann*”[*0-9′*]* → Ann

7.5.2 Constructors for Parse Trees

Function symbols can be literals—strings of characters between double quotes—or identifiers. Specification of the identifiers is not included in the ATerm format. For each application of ATerms, an appropriate set of AFuns should be declared, with the requirement that they are restricted to names of the form `[a-z][a-zA-Z0-9\-]*`. For the representation of grammars, symbols, productions and trees we define the following function symbols.

module Kernel-Sdf-Tree-Constructors
imports Grammar-Tree-Constructors$^{A.2.1}$
exports
  context-free syntax
    “prod” → AFun
    “no-attrs” → AFun
    “attrs” → AFun
    “atr” → AFun
    “syntax” → AFun

    “appl” → AFun
    “amb” → AFun

The function symbols ‘appl’ and ‘amb’ will be used to represent parse trees. The others will be used in the encoding of grammar structures. Each extension of the kernel that adds new constructors for symbols or grammars should also add the corresponding ATerm function symbols. These are included in §A.2.

7.5.3 ATerm Encoding

Now we can encode symbols, grammars and productions as terms. For each sort $\mathcal{S}$ a function aterm($\mathcal{S}$) → ATerm is defined that encodes $\mathcal{S}$-expressions as ATerms. The encoding is injective. For each sort $\mathcal{S}$ a decoding function s(ATerm) → $\mathcal{S}$ is defined such that s(aterm(s)) = s. Figure 7.1 illustrates the encoding of symbols and productions as ATerms.

The following module defines the encoding of lists of symbols. The encoding of constructors for symbols is defined in a module for each extension of the kernel; see §A.2.

module Symbols-ATerms
imports Symbols-Projection$^{7.1.1}$ ATerm-Lists
exports
  context-free syntax
    aterm(Symbol) → ATerm
    atermlist(Symbols) → ATermList
    symbol(ATerm) → Symbol
    symbols(ATermList) → Symbols

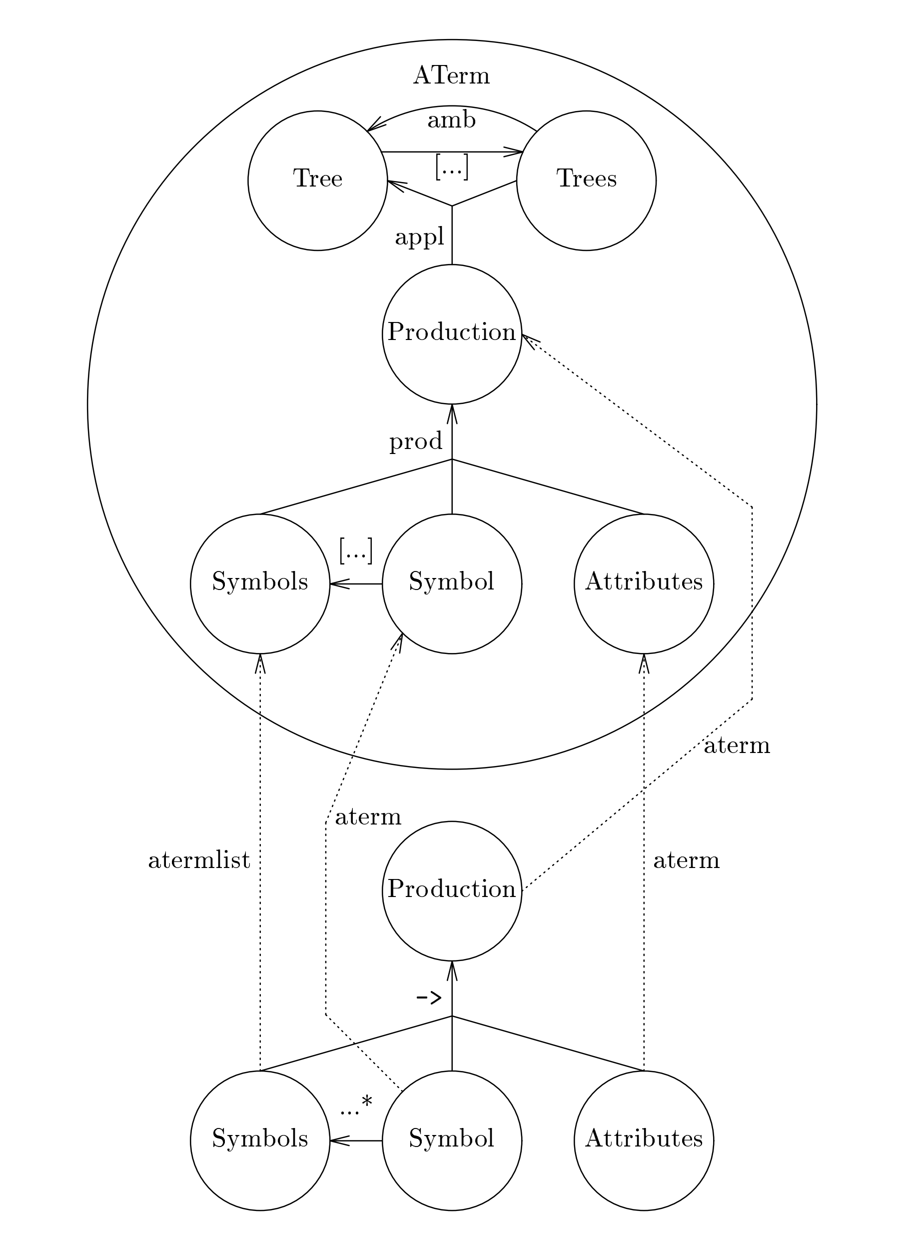


Figure 7.1: Encoding symbols, productions and parse trees in a fixed term format. Grammar domains such as Symbol and Production are mapped (the dotted arrows) onto subsets of the set of ATerms. Parse Trees are another subset of ATerms formed by the constructor 'appl' from a production and a list of trees, or by the constructor 'amb' from a list of trees.

equations
Encoding lists of symbols.

$$[1] \qquad \text{atermlist}(\mathcal{A}) = [\text{aterm}(\mathcal{A})]$$
$$[2] \qquad \text{atermlist}(\alpha) = [\,] \quad \textbf{when} \;\; \alpha =$$
$$[3] \qquad \text{atermlist}(\alpha^+ \; \beta^+) = \text{atermlist}(\alpha^+) +\!\!+ \; \text{atermlist}(\beta^+)$$

Decoding lists of symbols.

$$[4] \qquad \text{symbols}([\,]) =$$
$$[5] \qquad \text{symbols}([T]) = \text{symbol}(T)$$
$$[6] \qquad \text{symbols}([T,\ Ts]) = \text{symbol}(T) +\!\!+ \; \text{symbols}([Ts])$$

where we have the requirement that

$$[7] \qquad \text{symbol}(\text{aterm}(\mathcal{A})) = \mathcal{A}$$

The last equation requires of each future definition that it should be such that the decoding of an encoded symbol gives the original symbol.

7.5.4 Encoding Productions

The following module defines the encoding of productions. As an example, consider the production

```
Exp "+" Exp -> Exp {left}
```

which is encoded as

```
prod([sort("Exp"),lit("+"),sort("Exp")],
     sort("Exp"),attrs([atr("left")]))
```

A production is represented by the function symbol ‘prod’ and the attributes of the production are represented by ‘attrs’. Note that this makes use of the encoding of sort and literal symbols that is defined in §A.2.

module Kernel-Sdf-ATerms
imports Kernel-Sdf-Projection$^{7.3.2}$ Kernel-Sdf-Tree-Constructors$^{7.5.2}$
Symbols-ATerms$^{7.5.3}$ Grammar-ATerms$^{A.2.2}$ ATerm-Lists
exports
context-free syntax

aterm(Production)	→ ATerm
aterm(Attributes)	→ ATerm
atermlist({Attribute “,”}*)	→ ATermList
aterm(Attribute)	→ ATerm
atermlist(Productions)	→ ATermList
production(ATerm)	→ Production
attributes(ATerm)	→ Attributes

$$
\begin{array}{ll}
\text{attribute(ATerm)} & \to \text{Attribute} \\
\text{productions(ATermList)} & \to \text{Productions}
\end{array}
$$

equations

Encoding productions and attributes.

$$
\begin{array}{lrcl}
[1] & \text{aterm}(\alpha \to \mathcal{A}\ \$) &=& \text{prod}(\text{atermlist}(\alpha), \text{aterm}(\mathcal{A}), \text{aterm}(\$)) \\
[2] & \text{aterm}() &=& \text{no-attrs} \\
[3] & \text{aterm}(\{\mathit{attr}^*\}) &=& \text{attrs}(\text{atermlist}(\mathit{attr}^*)) \\
[4] & \text{atermlist}(\mathit{attr}^*) &=& [\,] \quad \textbf{when} \ \ \{\mathit{attr}^*\} = \{\} \\
[5] & \text{atermlist}(\mathit{attr}) &=& [\text{aterm}(\mathit{attr})] \\
[6] & \text{atermlist}(\mathit{attr}_1^+, \mathit{attr}_2^+) &=& \text{atermlist}(\mathit{attr}_1^+) +\!\!+ \text{atermlist}(\mathit{attr}_2^+)
\end{array}
$$

Decoding productions and attributes.

$$
\begin{array}{lrcl}
[7] & \text{production}(\text{prod}(\mathit{Tl}, T, T')) &=& \text{symbols}(\mathit{Tl}) \to \text{symbol}(T)\ \text{attributes}(T') \\
[8] & \text{attributes}(\text{no-attrs}) &=& \\
[9] & \text{attributes}(\text{attrs}(\mathit{Tl})) &=& \text{attributes}(\mathit{Tl}) \\
[10] & \text{attributes}([\,]) &=& \{\} \\
[11] & \text{attributes}([T]) &=& \{\text{attribute}(T)\} \\
[12] & \text{attributes}([T, \mathit{Ts}]) &=& \{\text{attribute}(T)\} +\!\!+ \text{attributes}([\mathit{Ts}])
\end{array}
$$

where we have the requirement that

$$
[13] \qquad \text{attribute}(\text{aterm}(\mathit{attr})) = \mathit{attr}
$$

Encoding grammars and lists of productions.

$$
\begin{array}{lrcl}
[14] & \text{aterm}(\text{syntax}\ p^*) &=& \text{syntax}(\text{atermlist}(p^*)) \\
[15] & \text{atermlist}(p) &=& [\text{aterm}(p)] \\
[16] & \text{atermlist}(p^*) &=& [\,] \quad \textbf{when} \ \ p^* = \\
[17] & \text{atermlist}(p_1^+\ p_2^+) &=& \text{atermlist}(p_1^+) +\!\!+ \text{atermlist}(p_2^+)
\end{array}
$$

Decoding of grammars and lists of productions.

$$
\begin{array}{lrcl}
[18] & \text{grammar}(\text{syntax}(\mathit{Tl})) &=& \text{syntax productions}(\mathit{Tl}) \\
[19] & \text{productions}([\,]) &=& \\
[20] & \text{productions}([T]) &=& \text{production}(T) \\
[21] & \text{productions}([T, \mathit{Ts}]) &=& \text{production}(T) +\!\!+ \text{productions}([\mathit{Ts}])
\end{array}
$$

7.5.5 Well-formed Parse Trees

Now we have prepared all equipment for the characterization of the terms that represent the well-formed parse trees over a grammar. The predicate $\mathcal{G} \vdash T$ determines whether a tree T is well-formed with respect to grammar $\mathcal{G}$. This is defined in terms of $\vdash T : T'$, which checks whether T is a tree of type T'. Since a tree contains all type information explicitly in the form of productions, this can be checked without reference to the grammar. The well-formedness with respect to the grammar is then defined by checking that all productions in a tree are actually productions of the grammar.

The main constructor for trees is the function 'appl' that creates an *application* of a context-free production to a list of trees such that the types of the argument trees correspond to the symbols in the left-hand side of the production. As an example, consider the grammar

```
sorts E
syntax
  [a-z]    -> E
  [\+]     -> "+"
  E "+" E -> E {left}
```

The following ATerm is a well-formed parse tree over this grammar for the sentence `a+b`.

```
appl(prod([sort("E"),lit("+"),sort("E")],
          sort("E"), attrs([atr("left")])),
     [appl(prod([char-class([range(97,122)])],
                sort("E"),no-attrs),
           [97]),
      appl(prod([char-class([43])],
                lit("+"),no-attrs),
           [43]),
      appl(prod([char-class([range(97,122)])],
                sort("E"),no-attrs),
           [98])])
```

Observe that the main `appl` has the aterm encoding of the production `E "+" E -> E {left}` as first argument and as second argument a list of three trees with types that correspond to the arguments of that production. The numbers at the leafs of the trees denote the ASCII values of the characters, i.e., 97 denotes `a`, 43 denotes `+` and 98 denotes `b`.

Context-free grammars can be ambiguous, i.e., generate more than one tree for a single sentence. The constructor 'amb' is introduced to represent parse forests, i.e., compact representations of sets of parse trees. A term amb($[T_1, \ldots, T_n]$) represents the set of parse trees containing the terms $T_1, \ldots, T_n$ (which can again contain 'amb' nodes). For example, the string `E "*" E "+" E` is ambiguous with respect to the grammar

```
sorts E
syntax
  E "+" E  -> E
  E "*" E  -> E
```

and therefore has the following parse forest that represents the two possible parses, left-associative `(E "*" E) "+" E` and right associative `E "*" (E "+" E)`.

```
amb([appl(prod([sort("E"),lit("+"),sort("E")],
               sort("E"),no-attrs),
          [appl(prod([sort("E"),lit("*"),sort("E")],
                     sort("E"),no-attrs),
                [sort("E"),
                 lit("*"),
                 sort("E")]),
           lit("+"),
           sort("E")]),
     appl(prod([sort("E"),lit("*"),sort("E")],
               sort("E"),no-attrs),
          [sort("E"),
           lit("*"),
           appl(prod([sort("E"),lit("+"),sort("E")],
                     sort("E"),no-attrs),
                [sort("E"),
                 lit("+"),
                 sort("E")])])])
```

Note that in order to reduce the size of the term, the subtrees for `E` and `"*"` and `"+"` are symbols. The tree represents the parse tree for a sentential form.

The yield of a tree is the concatenation of the characters at the leafs of the tree. For instance, the yield of the first tree above is `[97] [43] [98]`, i.e., the list of characters `a+b`.

module Kernel-Sdf-Trees
imports Kernel-Sdf-ATerms[7.5.4] Kernel-Sdf-Projection[7.3.2]
exports
context-free syntax

Grammar "⊢" ATerm	→ Bool
"⊢" ATerm ":" ATerm	→ Bool
yield "⟦" Grammar "⟧" "(" ATerm ")"	→ Symbols
args(ATerm)	→ ATermList
type(ATerm)	→ ATerm
prods(ATerm)	→ Productions

variables

"*Prod*"[*0-9′*]∗	→ ATerm
"*Res*"[*0-9′*]∗	→ ATerm
"*Attrs*"[*0-9′*]∗	→ ATerm
"*Args*"[*0-9′*]∗	→ ATermList

equations

A term T is a well-formed parse tree *over grammar* $\mathcal{G}$, if it is a well-formed tree and if all its productions are productions of $\mathcal{G}$.

[1] $$\frac{\vdash\ T : \mathrm{type}(T) \wedge \mathrm{prods}(T) \subseteq \mathrm{P}(\mathcal{G}) = \top}{\mathcal{G} \vdash T\ =\ \top}$$

Otherwise, the tree is not well-formed.

[2] $$\mathcal{G} \vdash T = \bot \quad \textbf{otherwise}$$

The function 'prods' gives the productions of a term, i.e., the list of all productions used in its applications.

[3] $$\text{prods}(\text{appl}(\mathit{Prod}, \mathit{Args})) = \text{production}(\mathit{Prod}) +\!\!+ \text{prods}(\mathit{Args})$$
[4] $$\text{prods}(\text{amb}(\mathit{Args})) = \text{prods}(\mathit{Args})$$
[5] $$\text{prods}([\,]) =$$
[6] $$\text{prods}([T]) = \text{prods}(T)$$
[7] $$\text{prods}([T, \mathit{Ts}]) = \text{prods}(T) +\!\!+ \text{prods}([\mathit{Ts}])$$
[8] $$\text{prods}(T) = \quad \textbf{otherwise}$$

For the definition of $\mathcal{G} \vdash T$ we need several auxiliary functions on terms. The function 'args' gives the arguments of a production. The 'type' of a production is its result type. The 'type' of an application is the result type of its production.

[9] $$\text{args}(\text{prod}(\mathit{Tl}, \mathit{Res}, \mathit{Attrs})) = \mathit{Tl}$$
[10] $$\text{args}(\text{appl}(\mathit{Prod}, \mathit{Args})) = \mathit{Args}$$
[11] $$\text{type}(\text{prod}(\mathit{Tl}, \mathit{Res}, \mathit{Attrs})) = \mathit{Res}$$
[12] $$\text{type}(\text{appl}(\mathit{Prod}, \mathit{Args})) = \text{type}(\mathit{Prod})$$
[13] $$\text{type}(\text{amb}([\mathit{Ts}])) = \text{type}(\text{first}([\mathit{Ts}]))$$

An application is a well-formed term of type T if the type of its production is T and if its arguments are well-formed terms with types that correspond to the argument types of the production of the application.

[14] $$\frac{\text{type}(\mathit{Prod}) = T, \ \vdash \mathit{Args} : \text{args}(\mathit{Prod}) = \top}{\vdash \text{appl}(\mathit{Prod}, \mathit{Args}) : T \ = \ \top}$$

A list of trees or symbols is well-formed if each element is well-formed.

[15] $$\vdash [\,] : [\,] = \top$$
[16] $$\vdash [T] : [T'] = \vdash T : T'$$
[17] $$\vdash [T, \mathit{Ts}] : [T', \mathit{Ts}'] = \vdash T : T' \wedge \vdash [\mathit{Ts}] : [\mathit{Ts}']$$

An ambiguous node is well-formed if all possibilities have the same type.

[18] $$\vdash \text{amb}([T]) : T' = \vdash T : T'$$
[19] $$\vdash \text{amb}([T, \mathit{Ts}]) : T' = \vdash T : T' \wedge \vdash \text{amb}([\mathit{Ts}]) : T'$$

A symbol term T is a well-formed tree of type T. This is used to represent trees for sentential forms.

[20] $$\frac{\text{symbol}(T) = \text{symbol}(T')}{\vdash T : T' \ = \ \top}$$

The yield of a term is the concatenation of all its leaf symbols.

[21] $$\text{yield}[\![\mathcal{G}]\!](\text{appl}(\mathit{Prod}, \mathit{Tl})) = \text{yield}[\![\mathcal{G}]\!](\mathit{Tl})$$

[22] $\text{yield}[\![\mathcal{G}]\!](\text{amb}(Tl)) = \text{yield}[\![\mathcal{G}]\!](\text{first}(Tl))$
[23] $\text{yield}[\![\mathcal{G}]\!]([\,]) =$
[24] $\text{yield}[\![\mathcal{G}]\!]([T]) = \text{yield}[\![\mathcal{G}]\!](T)$
[25] $\text{yield}[\![\mathcal{G}]\!]([T,\ Ts]) = \text{yield}[\![\mathcal{G}]\!](T) +\!\!+ \text{yield}[\![\mathcal{G}]\!]([Ts])$
[26] $\text{yield}[\![\mathcal{G}]\!](T) = \text{symbol}(T)$ **otherwise**

7.5.6 Trees with Characters and Literals

In the previous section we defined character classes and literals as symbols in grammars. Since the meaning of literals is defined in terms of character classes by means of context-free productions, the definition of well-formedness of parse trees does not need to be extended for literals. However, for character classes we have to extend the definition such that a character is a tree with as type any character class that contains it, i.e., a parse tree is a well-formed tree of a character class type if it is a character (represented by a natural number) that is an element of the character class. For example, the term

```
appl(prod([sort("Id"),char-class([range(97,122)])],
          sort("Id"),no-attrs),
     [appl(prod([char-class([range(97,122)])],
                sort("Id"),no-attrs),
           [97]),
      98])
```

is a well-formed parse tree for the identifier `ab`. The definition of well-formedness and yield is extended as follows:

module CC-Sdf-Trees
imports CC-Sdf-ATerms$^{A.2.2}$ Kernel-Sdf-Trees$^{7.5.5}$
Character-Class-Normalization
equations

A character is represented by an integer. The characteristic functions for trees are extended for this new tree constructor. The type of a character is the character code itself. The yield of a character is a character class containing the single character. A character does not contain any productions.

[1] $\text{type}(n) = n$
[2] $\text{yield}[\![\mathcal{G}]\!](n) = [\text{character}(n)]$
[3] $\text{prods}(n) =$

A character code n is a well-formed tree of type T if T represents a character class that contains the character corresponding to n.

[4] $$\frac{\text{symbol}(T) = cc,\ \text{character}(n) \in cc = \top}{\vdash n : T\ =\ \top}$$

7.5.7 Cyclic Parse Forests

Some grammars generate infinitely many parse trees for a single string. For instance, the grammar

```
syntax
       -> S
  [a] -> S
  S S -> S
```

generates infinitely many trees for the string `a`. The collection of parse trees for strings over such grammars can be finitely represented by means of a cyclic parse forest (Billot and Lang, 1989, Rekers, 1992). However, in the term format defined here we have no provisions for cyclic forests. The parser for SDF2 in Chapter 3 *does* generate a cyclic parse forest.

In ASF+SDF cyclic structures cannot be expressed in a natural way. This can be simulated by explicitly representing the pointer structure by means of a table of references and tags to represent these references. But this would complicate the entire specification. Since the application of SDF2 will be mainly to non-cyclic grammars we have not gone into the trouble of defining cyclic forests in ASF+SDF.

7.5.8 Equality of Trees

We define an equality predicate $\doteq$ on parse trees. At this point this comes down to syntactic equality. Later on we will extend the predicate such that trees that are not syntactically equal can be equal. This will be useful to abstract from certain details in parse trees. For instance, in §8.3 we will introduce parse trees containing layout. This is useful for applications that are aware of layout. But most applications will want to abstract from the specific layout in a tree and consider two trees equal up to layout. Another application is the equality of trees with associative operators such as list concatenation. The definition of the equality predicate below is intended to specify the details of such equality considerations. Furthermore, we define membership of a tree in a parse forest.

module Kernel-Sdf-Equality
imports Kernel-Sdf-Trees$^{7.5.5}$
exports
  context-free syntax
    ATerm "$\doteq$" ATerm $\rightarrow$ Bool
    ATerm "$\dot{\in}$" ATerm $\rightarrow$ Bool
equations
Equality of applications. If the productions are the same two applications are equal if the argument lists are.

[1] $$\frac{Args \doteq Args' = \top}{\text{appl}(Prod,\ Args) \doteq \text{appl}(Prod,\ Args')\ =\ \top}$$

Argument lists are equal if the elements are pairwise equal.

[2] $$[\,] \doteq [\,] = \top$$
[3] $$[T] \doteq [T'] = T \doteq T'$$
[4] $$[T,\ Ts] \doteq [T',\ Ts'] = T \doteq T' \wedge [Ts] \doteq [Ts']$$

An ambiguity node is equal to a tree if it all its possibilities are contained in the tree and vice versa.

[5] $$\frac{\text{amb}([Ts]) \mathrel{\dot\in} T \wedge T \mathrel{\dot\in} \text{amb}([Ts]) = \top}{\text{amb}([Ts]) \doteq T \;=\; \top}$$

[6] $$\frac{\text{amb}([Ts]) \mathrel{\dot\in} T \wedge T \mathrel{\dot\in} \text{amb}([Ts]) = \top}{T \doteq \text{amb}([Ts]) \;=\; \top}$$

If none of the cases above apply the terms are not equal.

[7] $$T_1 \doteq T_2 = \bot \quad \textbf{otherwise}$$

A tree is member of a parse forest (tree containing ambiguities) if it is contained in one of the possibilities of an ambiguity.

[8] $$\frac{T_1 \mathrel{\dot\in} T_2 = \top}{T_1 \mathrel{\dot\in} \text{amb}([T_2]) \;=\; \top}$$

[9] $$\frac{T_1 \mathrel{\dot\in} T_2 \vee T_1 \mathrel{\dot\in} \text{amb}([Ts]) = \top}{T_1 \mathrel{\dot\in} \text{amb}([T_2,\ Ts]) \;=\; \top}$$

An ambiguity is contained in a forest if *all* its possibilities are contained in the forest.

[10] $$\frac{T_1 \mathrel{\dot\in} T_2 = \top}{\text{amb}([T_1]) \mathrel{\dot\in} T_2 \;=\; \top}$$

[11] $$\frac{T_1 \mathrel{\dot\in} T_2 \wedge \text{amb}([Ts]) \mathrel{\dot\in} T_2 = \top}{\text{amb}([T_1,\ Ts]) \mathrel{\dot\in} T_2 \;=\; \top}$$

An application is contained in an application, if the arguments of the first are contained in the arguments of the second.

[12] $$\frac{Args_1 \mathrel{\dot\in} Args_2 = \top}{\text{appl}(Prod,\ Args_1) \mathrel{\dot\in} \text{appl}(Prod,\ Args_2) \;=\; \top}$$

Lists

[13] $$[\,] \mathrel{\dot\in} [\,] = \top$$
[14] $$[T] \mathrel{\dot\in} [T'] = T \mathrel{\dot\in} T'$$
[15] $$[T,\ Ts] \mathrel{\dot\in} [T',\ Ts'] = T \mathrel{\dot\in} T' \wedge [Ts] \mathrel{\dot\in} [Ts']$$

If none of the cases above apply membership does not hold

[16] $$T_1 \mathrel{\dot\in} T_2 = \bot \quad \textbf{otherwise}$$

8

Disambiguation and Abbreviation

In this chapter we present features for disambiguation of ambiguous grammars and abbreviation of common patterns. Priorities are used to disambiguate ambiguous expression syntax, providing support for compact abstract syntax. Priorities are defined by means of an extension of the well-formedness predicate on parse trees. Regular expressions abbreviate common patterns of productions such as lists, optional constructs, alternatives, etc. Regular expressions are defined by generating the defining productions for each expression in the grammar. Lexical and context-free syntax sections separate the definition of tokens and phrases. These are integrated into a single context-free grammar by normalization such that no interference between the two levels is created. Furthermore, the definition of the placement of layout between tokens is handled by this normalization. Follow restrictions and reject productions are provided to express lexical disambiguation rules such as 'prefer longest match' and 'prefer literals'.

8.1 Priorities

Context-free grammars can be ambiguous. There are many methods for the disambiguation of context-free grammars. Most programming language oriented formalisms provide some kind of precedence based method. Here we adopt the method of disambiguation by associativity and priority as used in SDF. New with respect to the design of priorities in Heering *et al.* (1989) is (a) disambiguation of lexical syntax by lexical priorities, (b) a more uniform notation for priority declarations, and (c) derivation of productions from priority declarations, which provides a more compact notation by avoiding multiple declarations of productions. A feature not present in SDF2 is the abbreviation of productions in priority declarations by the list of literals of the left-hand side. (For example, `"*" > "+"` as an abbreviation of `E "*" E -> E > E "+" E -> E`.) The reason for this omission is the unclear semantics in combination with modularization. Also $<$ priority-chains are not included in SDF2 because these can also be expressed using $>$ chains.

We first define syntax, projection functions and normalization of priority declarations. In §8.1.4 we describe an extension of the well-formedness predicate on parse trees that characterizes parse trees without priority conflicts.

Example 8.1.1 The following grammar defines priority and associativity relations over the syntax of expressions with unary negation and binary operators for exponentiation, multiplication, addition and subtraction. Note that, unlike in SDF, the syntax for arithmetic operators can be defined *inside* the priorities section.

```
syntax
  "(" E ")" -> E {bracket}
priorities
   "-" E   -> E
>  E "^" E -> E {right}
>  E "*" E -> E {left}
> {left:
   E "+" E -> E {assoc}
   E "-" E -> E {left}}
```

This grammar declares that unary $-$ has *higher priority* than ^, which has higher priority than binary $*$, which has higher priority than $+$ and binary $-$. The latter two are mutually left associative as declared by the *group associativity*. The bracket production declares that parentheses can also be used to disambiguate expressions. With this grammar the expression `E--E*E+E-E^E` should be interpreted as `((E-((-E)*E))+E)-(E^E)`. □

8.1.1 Syntax

The priorities section of a grammar defines the priority relation $>$ on productions and the associativity relations 'left', 'right', 'assoc', and 'non-assoc'. A priority declaration is either a `>` chain or an associativity declaration. The objects of these declarations are single productions or groups of productions. A group can have an X-associativity, which declares the productions in the group to be mutually X-associative.

The 'bracket' attribute declares a production of the form $l\ \mathcal{A}\ r \rightarrow \mathcal{A}$ {bracket}, with 'l' ('r') denoting the syntax for the left- (right-) bracket, to be the identity function on $\mathcal{A}$. Such productions can be used to explicitly disambiguate some text or to indicate a different disambiguation than the one given by priority rules.

module Priority-Sdf-Syntax
imports Kernel-Sdf-Syntax[7.3.1]
exports
  sorts Associativity Group Priority Priorities
  context-free syntax

"left"	→ Associativity
"right"	→ Associativity
"non-assoc"	→ Associativity
"assoc"	→ Associativity

"bracket"	→ Attribute
Associativity	→ Attribute
Production	→ Group
"{" Productions "}"	→ Group
"{" Associativity ":" Productions "}"	→ Group
{Group ">"}+	→ Priority
Group Associativity Group	→ Priority
{Priority ","}*	→ Priorities
"priorities" Priorities	→ Grammar

variables

"*g*"[*0-9'*]*	→ Group
"*gg**"[*0-9'*]*	→ {Group ">"}*
"*gg*+"[*0-9'*]*	→ {Group ">"}+
"*pr*"[*0-9'*]*	→ Priority
"*pr*""*"[*0-9'*]*	→ {Priority ","}*
"*pr*""+"[*0-9'*]*	→ {Priority ","}+
"*as*"[*0-9'*]*	→ Associativity

8.1.2 Projection

The projection function 'Pr' yields the list of all priority declarations of a grammar. The projection function '$\overline{\text{Pr}}$' yields the grammar without its priority declarations.

module Priority-Sdf-Projection
imports Priority-Sdf-Syntax[8.1.1] Kernel-Sdf-Projection[7.3.2]
exports
context-free syntax

Priorities "++" Priorities	→ Priorities	{**assoc**}
"Pr"(Grammar)	→ Priorities	
"$\overline{\text{Pr}}$"(Grammar)	→ Grammar	
Priority "∈" Priorities	→ Bool	

equations
Concatenation of priority declarations.

[1] $$pr_1^* +\!\!+\; pr_2^* = pr_1^*,\, pr_2^*$$

The priorities and non-priorities of a grammar.

[2] $$\text{Pr}(\text{priorities } pr^*) = pr^*$$
[3] $$\text{Pr}(\mathcal{G}_1\ \mathcal{G}_2) = \text{Pr}(\mathcal{G}_1) +\!\!+\; \text{Pr}(\mathcal{G}_2)$$
[4] $$\text{Pr}(\mathcal{G}) = \quad \textbf{otherwise}$$
[5] $$\overline{\text{Pr}}(\text{priorities } pr^*) = \emptyset$$
[6] $$\overline{\text{Pr}}(\mathcal{G}_1\ \mathcal{G}_2) = \overline{\text{Pr}}(\mathcal{G}_1)\ \overline{\text{Pr}}(\mathcal{G}_2)$$
[7] $$\overline{\text{Pr}}(\mathcal{G}) = \mathcal{G} \quad \textbf{otherwise}$$

Membership of a priority declaration. A pair is member of a declaration if the declaration contains a pair with similar productions. Recall from §7.3.2 that two productions are similar if they are the same except for their attributes, which may be different.

[8] $$\frac{p_1 \cong p_3 = \top,\;\; p_2 \cong p_4 = \top}{p_1 > p_2 \in pr_1^*,\, p_3 > p_4,\, pr_2^* \;=\; \top}$$

[9] $$\frac{p_1 \cong p_3 \wedge p_2 \cong p_4 \vee p_1 \cong p_4 \wedge p_2 \cong p_3 = \top}{p_1 \;as\; p_2 \in pr_1^*,\, p_3 \;as\; p_4,\, pr_2^* \;=\; \top}$$

[10] $$pr \in pr^* \;=\; \bot$$
otherwise

8.1.3 Normalization

The complex syntax for priority declarations can be expressed by means of only binary declarations for the $>$ relation and the associativity relations as follows: (1) Priority chains of the form $p_1 > \ldots > p_n$ are normalized to lists of simple priorities of the form $p_i > p_{i+1}$. This relation is closed transitively. (2) Associativity declarations in attributes and group associativities are expressed as binary associativity declarations of the form p_1 *as* p_2. (3) The productions that are mentioned in priorities sections are added to the syntax section of the grammar.

Example 8.1.2 The normalization of the grammar in Example 8.1.1 on page 158 is:

```
syntax
  "-" E   -> E
  E "^" E -> E {right}
  E "*" E -> E {left}
  E "+" E -> E {assoc}
  E "-" E -> E {left}
priorities
  "-" E -> E          >     E "^" E -> E {right},
  E "^" E -> E {right} right E "^" E -> E {right},
  E "^" E -> E {right} >     E "*" E -> E {left},
  E "*" E -> E {left}  >     E "+" E -> E {assoc},
  E "*" E -> E {left}  >     E "-" E -> E {left},
  E "*" E -> E {left}  left  E "*" E -> E {left},
  E "+" E -> E {assoc} left  E "-" E -> E {left},
  E "+" E -> E {assoc} assoc E "+" E -> E {assoc},
  E "-" E -> E {left}  left  E "-" E -> E {left}
```

Observe that all productions mentioned in the priorities declaration are also declared as productions in the 'syntax' part. Furthermore, the $>$ chain is made into a binary relation, which is transitively closed. All associativity attributes are expressed by means of binary declarations. □

module Priority-Sdf-Normalization
imports Priority-Sdf-Syntax[8.1.1] Booleans Kernel-Sdf-Normalization[7.3.3]
Priority-Sdf-Projection[8.1.2]
exports
context-free syntax

"p⟦" Grammar "⟧"	→ Grammar
"assoc" "⟦" Grammar "⟧"	→ Priorities
"assoc" "⟦" Productions "⟧"	→ Priorities
"syntax" "⟦" Priorities "⟧"	→ Grammar
"norm" "⟦" Priorities "⟧"	→ Priorities
"trans" "⟦" Priorities "⟧"	→ Priorities

equations

The normalization function p⟦_⟧ extracts syntax information from priorities and priority information from syntax, normalizes the priorities declarations and takes the transitive closure.

$$[1] \quad \frac{\mathrm{Pr}(\mathcal{G}) = pr_1^*,\ \mathrm{norm}[\![pr_1^*]\!] = pr_2^*,\ \overline{\mathrm{Pr}}(\mathcal{G})\ \mathrm{syntax}[\![pr_2^*]\!] = \mathcal{G}\,'}{\mathrm{p}[\![\mathcal{G}]\!]\ =\ \mathcal{G}\,'\ \mathrm{priorities\ trans}[\![pr_2^* +\!\!+\ \mathrm{assoc}[\![\mathcal{G}\,']\!]]\!]}$$

The function norm⟦_⟧ normalizes a priority declaration to a list of pairs of the form $p > p'$ or $p\ as\ p'$ by eliminating >-chains and {_} groups.

$$[2] \quad \mathrm{norm}[\![p\!> p\,']\!] = p\!> p\,'$$
$$[3] \quad \mathrm{norm}[\![p\ as\ p\,']\!] = p\ as\ p\,',\ p\,'\ as\ p$$

Each of the priority declarations in the list is normalized.

$$[4] \quad \mathrm{norm}[\![\,]\!] =$$
$$[5] \quad \mathrm{norm}[\![p]\!] =$$
$$[6] \quad \mathrm{norm}[\![pr_1^+,\ pr_2^+]\!] = \mathrm{norm}[\![pr_1^+]\!] +\!\!+\ \mathrm{norm}[\![pr_2^+]\!]$$

A > chain is broken into binary > declarations. The transitive closure defined below ensures that $p_1 > p_3$ if $p_1 > p_2 > p_3$ was specified.

$$[7] \quad \mathrm{norm}[\![gg_1^+\!> gg_2^+\!> gg_3^+]\!] = \mathrm{norm}[\![gg_1^+\!> gg_2^+,\ gg_2^+\!> gg_3^+]\!]$$

Groups and priority. A group is an abbreviation for a pointwise extension of the declared relation to the members of the group.

$$[8] \quad \mathrm{norm}[\![\{p\}]\!] =$$
$$[9] \quad \mathrm{norm}[\![\{\}\!> g]\!] =$$
$$[10] \quad \mathrm{norm}[\![g\!> \{\}]\!] =$$
$$[11] \quad \mathrm{norm}[\![\{p_1^+\ p_2^+\}\!> g]\!] = \mathrm{norm}[\![\{p_1^+\}\!> g,\ \{p_2^+\}\!> g]\!]$$
$$[12] \quad \mathrm{norm}[\![g\!> \{p_1^+\ p_2^+\}]\!] = \mathrm{norm}[\![g\!> \{p_1^+\},\ g\!> \{p_2^+\}]\!]$$
$$[13] \quad \mathrm{norm}[\![\{p\}\!> g]\!] = \mathrm{norm}[\![p\!> g]\!]$$
$$[14] \quad \mathrm{norm}[\![g\!> \{p\}]\!] = \mathrm{norm}[\![g\!> p]\!]$$
$$[15] \quad \mathrm{norm}[\![\{as : p^*\}\!> g]\!] = \mathrm{norm}[\![\{as : p^*\},\ \{p^*\}\!> g]\!]$$
$$[16] \quad \mathrm{norm}[\![g\!> \{as : p^*\}]\!] = \mathrm{norm}[\![\{as : p^*\},\ g\!> \{p^*\}]\!]$$

Groups and associativity.

[17] $\mathrm{norm}[\![\{\}\ \mathit{as}\ g]\!] = \mathrm{norm}[\![g]\!]$
[18] $\mathrm{norm}[\![\{p\ p^*\}\ \mathit{as}\ g]\!] = \mathrm{norm}[\![p\ \mathit{as}\ g,\ \{p^*\}\ \mathit{as}\ g]\!]$
[19] $\mathrm{norm}[\![g\ \mathit{as}\ \{\}]\!] = \mathrm{norm}[\![g]\!]$
[20] $\mathrm{norm}[\![g\ \mathit{as}\ \{p\ p^*\}]\!] = \mathrm{norm}[\![g\ \mathit{as}\ p,\ g\ \mathit{as}\ \{p^*\}]\!]$
[21] $\mathrm{norm}[\![\{\mathit{as}_1 : p^*\}\ \mathit{as}_2\ g]\!] = \mathrm{norm}[\![\{\mathit{as}_1 : p^*\},\ \{p^*\}\ \mathit{as}_2\ g]\!]$
[22] $\mathrm{norm}[\![g\ \mathit{as}_2\ \{\mathit{as}_1 : p^*\}]\!] = \mathrm{norm}[\![\{\mathit{as}_1 : p^*\},\ g\ \mathit{as}_2\ \{p^*\}]\!]$

Associativity groups are abbreviations. The members of an associativity group are mutually associative with respect to the declared relation. If the group contains a single production, it is taken to define the associativity for that production. Otherwise, the associativities are defined only *between* the members of the group and are not defined reflexively. This makes it possible, for instance, to have a production that is left-associative with respect to itself, but right-associative with respect to a group of other productions.

[23] $\mathrm{norm}[\![\{\mathit{as} : p\}]\!] = p\ \mathit{as}\ p$
[24] $\mathrm{norm}[\![\{\mathit{as} : p_1\ p_2\}]\!] = p_1\ \mathit{as}\ p_2$
[25] $\mathrm{norm}[\![\{\mathit{as} : p_1\ p_2\ p^+\}]\!] = \mathrm{norm}[\![p_1\ \mathit{as}\ p_2,\ \{\mathit{as} : p_1\ p^+\},\ \{\mathit{as} : p_2\ p^+\}]\!]$

The function $\mathrm{trans}[\![\_]\!]$ takes the transitive closure of the $>$ relation.

[26] $$\frac{pr_1^*,\ p_1 > p_2,\ pr_2^* = pr^*,\ \ pr^* = pr_3^*,\ p_2 > p_3,\ pr_4^*,\ \ p_1 > p_3 \in pr^* \neq \top}{\mathrm{trans}[\![pr^*]\!] \ = \ \mathrm{trans}[\![p_1 > p_3,\ pr^*]\!]}$$

[27] $\mathrm{trans}[\![pr_1^*,\ pr,\ pr_2^*,\ pr,\ pr_3^*]\!] \ = \ \mathrm{trans}[\![pr_1^*,\ pr,\ pr_2^*,\ pr_3^*]\!]$

[28] $\mathrm{trans}[\![pr^*]\!] \ = \ pr^*$
otherwise

The function $\mathrm{assoc}[\![\_]\!]$ derives associativity declarations from the productions of a grammar. Productions that have an attribute declaring them as left, right, or non-associative produce a declaration of that associativity in the priorities declaration.

[29] $\mathrm{assoc}[\![\mathcal{G}]\!] = \mathrm{assoc}[\![\mathrm{P}(\mathcal{G})]\!]$
[30] $\mathrm{assoc}[\![]\!] =$
[31] $\mathrm{assoc}[\![p_1^+\ p_2^+]\!] = \mathrm{assoc}[\![p_1^+]\!] \mathbin{+\!\!+} \mathrm{assoc}[\![p_2^+]\!]$
[32] $\mathrm{assoc}[\![p]\!] = p\ \mathit{as}\ p \quad \textbf{when} \ \ p = \alpha \to \mathcal{A}\ \{\mathit{attr}_1^*,\ \mathit{as},\ \mathit{attr}_2^*\}$
[33] $\mathrm{assoc}[\![p]\!] = \quad$ **otherwise**

The function 'syntax' derives from a priorities declaration the list of all productions referred to in that declaration.

[34] $\mathrm{syntax}[\![]\!] = \emptyset$
[35] $\mathrm{syntax}[\![pr_1^+,\ pr_2^+]\!] = \mathrm{syntax}[\![pr_1^+]\!]\ \mathrm{syntax}[\![pr_2^+]\!]$
[36] $\mathrm{syntax}[\![p_1 > p_2]\!] = \mathrm{syntax}\ p_1\ p_2$

[37] $\text{syntax}[\![p_1 \ \mathit{as}\ p_2]\!] = \text{syntax}\ p_1\ p_2$

Merging and ordering of grammars.

[38] $\text{priorities} = \emptyset$

[39] $\text{priorities}\ pr_1^* \diamond \text{priorities}\ pr_2^* = \langle \text{priorities}\ pr_1^*,\ pr_2^*,\ \emptyset \rangle$

[40] $\text{priorities}\ pr^* \diamond \text{syntax}\ p^* = \langle \text{syntax}\ p^*,\ \text{priorities}\ pr^* \rangle$

8.1.4 Parse Trees with Priority Conflicts

We extend the notion of well-formedness of parse trees to well-formedness over a grammar with priorities. A tree is well-formed if it is a well-formed context-free tree and if, moreover, it does not contain priority conflicts.

module Priority-Sdf-Trees
imports Kernel-Sdf-Trees[7.5.5] Priority-Sdf-Projection[8.1.2]
exports
  context-free syntax

Grammar "$\vdash_{\text{prio}}$" ATerm	$\rightarrow$ Bool
conf "$[\![$" Priorities "$]\!]$" "(" ATerm ")"	$\rightarrow$ Bool
rootconf "$[\![$" Priorities "$]\!]$" "(" Production "," ATermList ")"	$\rightarrow$ Bool
left "$[\![$" Priorities "$]\!]$" "(" Production "," ATerm ")"	$\rightarrow$ Bool
middle "$[\![$" Priorities "$]\!]$" "(" Production "," ATermList ")"	$\rightarrow$ Bool
right "$[\![$" Priorities "$]\!]$" "(" Production "," ATerm ")"	$\rightarrow$ Bool

equations

We introduce an extension of the notion of well-formedness. A tree is well-formed with respect to the priorities in a grammar, if it is well-formed with respect to the grammar and does not contain a priority conflict.

[1] $$\frac{\mathcal{G} \vdash T = \top,\ \ \text{conf}[\![\text{Pr}(\mathcal{G})]\!](T) = \bot}{\mathcal{G} \vdash_{\text{prio}} T\ =\ \top}$$

An application has a conflict if it has a root conflict or if any of its descendants has a conflict.

[2] $$\frac{\text{rootconf}[\![pr^*]\!](\text{production}(\mathit{Prod}),\ \mathit{Args}) \vee \text{conf}[\![pr^*]\!](\mathit{Args}) = \mathit{Bool}}{\text{conf}[\![pr^*]\!](\text{appl}(\mathit{Prod},\ \mathit{Args}))\ =\ \mathit{Bool}}$$

For the other constructors, a tree has a conflict if any descendant has.

[3] $\text{conf}[\![pr^*]\!](\text{amb}([T])) = \text{conf}[\![pr^*]\!](T)$

[4] $\text{conf}[\![pr^*]\!](\text{amb}([T,\ \mathit{Ts}])) = \text{conf}[\![pr^*]\!](T) \vee \text{conf}[\![pr^*]\!](\text{amb}([\mathit{Ts}]))$

[5] $\text{conf}[\![pr^*]\!]([\ ]) = \bot$

[6] $\text{conf}[\![pr^*]\!]([T]) = \text{conf}[\![pr^*]\!](T)$

[7] $\text{conf}[\![pr^*]\!]([T,\ \mathit{Ts}]) = \text{conf}[\![pr^*]\!](T) \vee \text{conf}[\![pr^*]\!]([\mathit{Ts}])$

An application with no descendants does not have a conflict.

[8] $$\text{rootconf}[\![pr^*]\!](p, [\,]) = \bot$$

An application with more than one descendant has a root conflict if it has a left conflict, a middle conflict or a right conflict.

[9] $$\text{rootconf}[\![pr^*]\!](p, [T,\ Ts]) = \text{left}[\![pr^*]\!](p,\ T) \vee \text{middle}[\![pr^*]\!](p, [Ts])$$

An injection, i.e., an application with only one child has a root conflict if its production has higher priority than its child's production.

[10] $$\frac{\text{production}(\mathit{Prod}) = p_2,\ \ p_1 > p_2 \in pr^* = \mathit{Bool}}{\text{rootconf}[\![pr^*]\!](p_1, [\text{appl}(\mathit{Prod},\ \mathit{Args})]) \ = \ \mathit{Bool}}$$

A tree has a left conflict if the productions of root and left-most child are mutually right-associative or non-associative, or if the root production has higher priority than the child production.

[11] $$\frac{\begin{array}{c}\text{production}(\mathit{Prod}) = p_2,\\ p_1 \text{ right } p_2 \in pr^* \vee p_1 \text{ non-assoc } p_2 \in pr^* \vee p_1 > p_2 \in pr^* = \mathit{Bool}\end{array}}{\text{left}[\![pr^*]\!](p_1, \text{appl}(\mathit{Prod},\ \mathit{Args})) \ = \ \mathit{Bool}}$$

If the left-most child is an ambiguitity node, the tree has a conflict if there is a left conflict with any of the possibilities of the ambiguity.

[12] $$\text{left}[\![pr^*]\!](p_1, \text{amb}([T])) = \text{left}[\![pr^*]\!](p_1,\ T)$$

[13] $$\text{left}[\![pr^*]\!](p_1, \text{amb}([T,\ Ts])) = \text{left}[\![pr^*]\!](p_1,\ T) \vee \text{left}[\![pr^*]\!](p_1, \text{amb}([Ts]))$$

A tree has a middle conflict if the root production has higher priority than any of the middle child productions.

[14] $$\frac{\text{production}(\mathit{Prod}) = p_2,\ \ p_1 > p_2 \in pr^* \vee \text{middle}[\![pr^*]\!](p_1, [Ts]) = \mathit{Bool}}{\text{middle}[\![pr^*]\!](p_1, [\text{appl}(\mathit{Prod},\ \mathit{Args}),\ Ts]) \ = \ \mathit{Bool}}$$

[15] $$\text{middle}[\![pr^*]\!](p, [T]) \ = \ \text{right}[\![pr^*]\!](p,\ T)$$

[16] $$\text{middle}[\![pr^*]\!](p_1, [\text{amb}([T]),\ Ts]) \ = \ \text{middle}[\![pr^*]\!](p_1, [T,\ Ts])$$

[17] $$\frac{\text{middle}[\![pr^*]\!](p_1, [T,\ Ts]) \vee \text{middle}[\![pr^*]\!](p_1, [\text{amb}([Ts]),\ Ts]) = \mathit{Bool}}{\text{middle}[\![pr^*]\!](p_1, [\text{amb}([T,\ Ts]),\ Ts]) \ = \ \mathit{Bool}}$$

A tree has a right conflict if the productions of root and right-most child are mutually left-associative, non-associative or associative (a synonym for left), or if the root production has higher priority than the child production.

[18] $$\frac{\begin{array}{l}\quad\text{production}(\mathit{Prod}) = p_2,\\ p_1 \text{ left } p_2 \in pr^* \vee p_1 \text{ assoc } p_2 \in pr^*\\ \vee\, p_1 \text{ non-assoc } p_2 \in pr^* \vee p_1 > p_2 \in pr^*\end{array} = \mathit{Bool}}{\text{right}[\![pr^*]\!](p_1, \text{appl}(\mathit{Prod},\ \mathit{Args})) \ = \ \mathit{Bool}}$$

The case of an ambiguity as right-most child.

[19] $$\text{right}[\![pr^*]\!](p_1, \text{amb}([T])) \;=\; \text{right}[\![pr^*]\!](p_1,\ T)$$

[20] $$\frac{\text{right}[\![pr^*]\!](p_1,\ T) \vee \text{right}[\![pr^*]\!](p_1, \text{amb}([Ts])) = \mathit{Bool}}{\text{right}[\![pr^*]\!](p_1, \text{amb}([T,\ Ts])) \;=\; \mathit{Bool}}$$

8.1.5 Discussion

Here we have described the requirements on parse trees that a parser should produce, i.e., not containing priority conflicts. There are various ways to implement this requirement. One possible scheme that is further discussed in Chapter 4 is to interpret the priority rules as a filter on parse forests that prunes the subtrees with conflicts. This scheme is used in the parser in the current ASF+SDF Meta-Environment (Heering *et al.*, 1989, Klint, 1993). An advantage of this approach is that disambiguation is decoupled from parsing and that other disambiguation filters could be added. The drawback of the approach is that the parse forest can become very large, which hampers efficiency. Therefore, applying the priority rules as early as possible in the parsing process will increase efficiency. A parser-generation time interpretation of priorities is described in Chapter 5. There the priorities are completely expressed in the parse tables produced by the parser generator. An implementation of this method is discussed in Chapter 3.

Other Disambiguation Methods Disambiguation by priority conflicts is similar to the methods using precedences of Earley (1975) and Aho *et al.* (1975). The latter also describe a method for interpreting these rules in the parser generation process, which is less general than the one in Chapter 5. Disambiguation by priorities as defined in this section is based on the definition of priorities in Heering *et al.* (1989). In that definition a second interpretation of priorities is defined. Parse trees are interpreted as a multi-set of productions and the priorities are interpreted as an ordering of such multi-sets. This ordering is used to make a further selection of trees if the filtering by priority conflicts does not solve all ambiguities.

Subtree exclusion is a disambiguation method introduced by Thorup (1994a) that works by specifying a finite set of partial parse trees that are forbidden as subtrees of parse trees yielded by the parser. This method allows a more fine tuned disambiguation than is achievable by the priority scheme. Examples are disambiguation of generic operators and internal arguments. Some problems can not be solved appropriately. The if-then-else ambiguity is solved in the same way as with priorities, which is not correct. In Chapter 4 these and several other disambiguation methods are studied in the framework of filters on parse forests.

Brackets Unparsing is complicated in the presence of priorities. When a parse tree is created by a semantics processor, a rewriter for instance, it might create a well-formed tree that does not satisfy the $\vdash_{\text{prio}}$ predicate, i.e., contains a priority conflict. Such trees are semantically meaningful, but problematic when their yield is considered. Naively translating an abstract syntax tree to a string as described before might lead to a string that, when parsed, does not represent the same tree because it would contain conflicts. To force equivalence of tree and string, brackets should be introduced. In Van den Brand and Visser (1996) the rules for priority conflicts are used to place brackets when unparsing an abstract syntax tree.

8.2 Regular Expressions

Certain patterns of context-free productions occur again and again. Examples of such patterns are lists, lists with separators, optional constructs and alternative. For example, a list of one or more identifiers can be specified by the grammar

```
syntax
  Id                -> Id-List
  Id-List Id-List -> Id-List {left}
```

Here a list is defined in terms of two constructors, one for singleton lists and one for concatenation of lists.

Many formalisms provide shortcuts for such patterns by extending the language of context-free grammars with some collection of regular operators on symbols. For instance, BNF provides an alternative at the level of productions, i.e., a production has the form $\mathcal{A} := \mathcal{A}_0 | \ldots | \mathcal{A}_n$, where the symbol '|' has the meaning of *or*. Extended BNF (EBNF) is the canonical extension of BNF with regular operators. In one formulation, Wirth (1977) adds the operators $\{\mathcal{A}\}$ for iteration and $[\mathcal{A}]$ for optionality. Variations on this notation appear in Lee (1972) and Williams (1982). SDF provides iteration $\mathcal{A}*$ and $\mathcal{A}+$ and $\{\mathcal{A}\ L\}\oplus$ iteration for abbreviation of lists of $\mathcal{A}$s separated by a literal L.

In this section we give an extension of context-free productions by a set of regular operators on symbols. In all the approaches mentioned above regular operators are given a special treatment. New in our formulation is the treatment of regular operators as first class citizens. They are nothing but constructors of new symbols that spare the specifier the burden of having to invent new names. As a consequence, a regular expression can occur at all positions where a normal symbol can occur, in particular in the right-hand side of a production.

This approach is motivated by the following considerations: (1) It enables us to express the meaning of regular expressions by means of a normalization of the grammar that adds defining productions for each expression. (2) Our grammars function as signatures for algebraic specifications, where each production represents a function. If regular symbols can not be the result of functions, as is the case in SDF, we still have to define an auxiliary symbol to define a function that yields such a result. For example, suppose that we want to define a function `add` that adds an integer to each integer in a list of integers. In the syntax below we can write this as

```
add(Int, Int*) -> Int*
```

whereas in SDF we should introduce an auxiliary sort `IntList` to represent the result sort of this function.

8.2.1 Syntax

We consider the following operators:

— Empty: The symbol () represents the empty string

— Concatenation: The symbol $(\mathcal{A}_1 \dots \mathcal{A}_n)$ with $n > 2$ denotes the concatenation $a_1 \dots a_n$ of expressions a_i of type $\mathcal{A}_i$.

— Alternative: The symbol $\mathcal{A}_1 | \dots | \mathcal{A}_n$, with $n \geq 2$, denotes an expression a of one of the types $\mathcal{A}_i$.

— Optional: The symbol $\mathcal{A}?$ is an optional $\mathcal{A}$

— Iteration: The symbol $\mathcal{A}*$ $(\mathcal{A}+)$ denotes a list $a_1 \dots a_n$ with $n \geq 0$ $(n \geq 1)$ of expressions a_i of type $\mathcal{A}$.

— Iteration with separator: The symbol $\{\mathcal{A}\ \mathcal{B}\}*$ $(\{\mathcal{A}\ \mathcal{B}\}+)$ denotes a list $a_1 b_1 \dots b_{n-1} a_n$ with $n \geq 0$ $(n \geq 1)$ of expressions a_i of type $\mathcal{A}$ separated by expressions b_i of type $\mathcal{B}$. Observe that, unlike in SDF, in SDF2 there is no limitation of the symbols that can be used as separators. For example, `{Stat [\;]|[\n]}*` denotes lists of statements separated by semicolons *or* newlines.

— Constrained iteration: The symbol $\{\mathcal{A}\}n+$ with $n \geq 2$ denotes a list $a_1 \dots a_n$ of n or more expressions a_i of type $\mathcal{A}$. Similarly for $\{\mathcal{A}\ \mathcal{B}\}n+$ with separator $\mathcal{B}$.

— Set expressions: The symbol Set[$\mathcal{A}$] represents the syntax of set expressions of the form $\{a_1, \dots, a_n\}$ with the a_i expressions of type $\mathcal{A}$

— Product: The symbol $\mathcal{A}_1 \# \dots \# \mathcal{A}_n$, with $n \geq 2$, denotes tuples $\langle a_1, \dots, a_n \rangle$ of expressions a_i of type $\mathcal{A}_i$.

— Functions: The symbol $(\mathcal{A}_1 \dots \mathcal{A}_n \Rightarrow \mathcal{B})$, with $n \geq 0$, denotes function expressions f that can be used in expressions $f(a_1 \dots a_n)$ of type $\mathcal{B}$ with the a_i expressions of type $\mathcal{A}_i$.

— Permutation: The symbol $\ll \mathcal{A}_1 \dots \mathcal{A}_n \gg$ denotes expressions of the form $a_1 \dots a_n$ such that for each $\mathcal{A}_i$ exactly one of the a_j has type $\mathcal{A}_i$.

The syntax of these operators is defined in the following module. Observe that the empty symbol () and sequences $(\mathcal{A}_1 \dots \mathcal{A}_n)$ are not defined using a single production "(" Symbol $*$ ")" $\rightarrow$ Symbol because parentheses around a single symbol are already used as brackets; see §7.1.

module Regular-Sdf-Syntax
imports Kernel-Sdf-Syntax[7.3.1] IntCon
exports
  context-free syntax

"(" ")"	$\to$ Symbol	
"(" Symbol Symbol+ ")"	$\to$ Symbol	
Symbol "?"	$\to$ Symbol	
Symbol "+"	$\to$ Symbol	
Symbol "$*$"	$\to$ Symbol	
"{" Symbol Symbol "}" "+"	$\to$ Symbol	
"{" Symbol Symbol "}" "$*$"	$\to$ Symbol	
"{" Symbol "}" NatCon "+"	$\to$ Symbol	
"{" Symbol Symbol "}" NatCon "+"	$\to$ Symbol	
"Set" "[" Symbol "]"	$\to$ Symbol	
Symbol "#" Symbol	$\to$ Symbol	{**right**}
"(" Symbols "$\Rightarrow$" Symbol ")"	$\to$ Symbol	
Symbol "|" Symbol	$\to$ Symbol	{**right**}
"$\ll$" Symbols "$\gg$"	$\to$ Symbol	

  priorities

{Symbol "?" $\to$ Symbol, Symbol "$*$" $\to$ Symbol, Symbol "+" $\to$ Symbol, Symbol NatCon "+" $\to$ Symbol} $>$ Symbol "#"Symbol $\to$ Symbol $>$ Symbol "|"Symbol $\to$ Symbol

8.2.2 Normalization

We define a normalization function r⟦_⟧ that for each regular expression that is used in the grammar introduces one or more productions that define its meaning. In this interpretation regular expressions form a shorthand for defining extra symbols and productions.

Example 8.2.1 The following production defines a single production describing the structure of a block in a While program consisting of an optional declaration followed by a list of statements.

```
syntax
  "begin" (Decl ";")? {Stat ";"}+ "end" -> Stat
```

The normalization of this grammar is:

```
syntax
  "begin" (Decl ";")? {Stat ";"}+ "end" -> Stat
                                         -> (Decl ";")?
  (Decl ";")                             -> (Decl ";")?
  Decl ";"                               -> (Decl ";")
  Stat                                   -> {Stat ";"}+
  {Stat ";"}+ ";" {Stat ";"}+            -> {Stat ";"}+ {left}
```

```
    {Stat ";"}+ ";" {Stat ";"}*          -> {Stat ";"}+
    {Stat ";"}* ";" {Stat ";"}+          -> {Stat ";"}+
    {Stat ";"}* ";" {Stat ";"}*          -> {Stat ";"}* {left}
                                         -> {Stat ";"}*
    {Stat ";"}+                          -> {Stat ";"}*
  priorities
  {left :
    {Stat ";"}+ ";" {Stat ";"}+          -> {Stat ";"}+ {left}
    {Stat ";"}+ ";" {Stat ";"}*          -> {Stat ";"}+
    {Stat ";"}* ";" {Stat ";"}+          -> {Stat ";"}+
    {Stat ";"}* ";" {Stat ";"}*          -> {Stat ";"}* {left}
  }>
    {Stat ";"}+                          -> {Stat ";"}*
```

We see that the meaning of the operators is expressed by means of extra productions. Observe how regular expressions are used as target symbols of productions. □

module Regular-Sdf-Normalization
imports Regular-Sdf-Syntax$^{8.2.1}$ Priority-Sdf-Syntax$^{8.1.1}$
Literals-Sdf-Syntax$^{7.4.3}$ Kernel-Sdf-Normalization$^{7.3.3}$
exports
context-free syntax

"r⟦" Grammar "⟧"	$\rightarrow$ Grammar
"r⟦" Symbols "⟧"	$\rightarrow$ Grammar
alt(Symbol, Symbol)	$\rightarrow$ Grammar
tup(Symbol)	$\rightarrow$ Symbols
perm(Symbols)	$\rightarrow$ Productions
perm3(Symbols, Symbols)	$\rightarrow$ Productions

equations
The function r⟦_⟧ adds defining productions for each regular expresssion occurring in one of the productions of the grammar. Existing productions are not affected.

[1] $$\mathrm{r}[\![\mathcal{G}]\!] = \mathcal{G}\ \mathrm{r}[\![\alpha]\!] \quad \textbf{when}\ \ \{\alpha\}\ =\ \mathrm{symbols}(\mathcal{G})$$

Recall that the function 'symbols', defined in §7.3.2, gives the set of all symbols in a grammar. The function r⟦_⟧ generates a grammar for each of the regular expressions in the list of symbols.

[2] $$\mathrm{r}[\![\,]\!] = \emptyset$$
[3] $$\mathrm{r}[\![\alpha^+\ \beta^+]\!] = \mathrm{r}[\![\alpha^+]\!]\ \mathrm{r}[\![\beta^+]\!]$$

Concatenation The regular expression (α) is a *symbol* that abbreviates the *concatenation* of the symbols α.

[4] $$\mathrm{r}[\![(\ )]\!]\ =\ \mathrm{syntax}\ \rightarrow (\ )$$

[5]
$$\frac{p \;=\; \mathcal{A}\,\alpha^{+} \rightarrow (\mathcal{A}\,\alpha^{+})}{\mathrm{r}[\![(\mathcal{A}\,\alpha^{+})]\!] \;=\; \text{syntax}\; p\; \mathrm{r}[\![\mathcal{A}\,\alpha^{+}]\!]}$$

Note that $\mathrm{r}[\![\mathcal{A}\,\alpha^{+}]\!]$ recursively produces the productions for regular expressions in the list of symbols $\mathcal{A}\,\alpha^{+}$.

Alternative The *alternative* $\mathcal{A}|\mathcal{B}$ denotes either $\mathcal{A}$ or $\mathcal{B}$. We could thus define $\mathrm{r}[\![\mathcal{A}|\mathcal{B}]\!]$ to yield the productions $\mathcal{A} \rightarrow \mathcal{A}|\mathcal{B}$ and $\mathcal{B} \rightarrow \mathcal{A}|\mathcal{B}$. However, if one of the alternatives is again an alternative, an unnecessary chain $\mathcal{A} \rightarrow \mathcal{A}|\mathcal{B}$ and $\mathcal{A}|\mathcal{B} \rightarrow \mathcal{A}|\mathcal{B}|\mathcal{C}$ is created. We would rather have $\mathcal{A} \rightarrow \mathcal{A}|\mathcal{B}|\mathcal{C}$. Therefore, we define

[6]
$$\mathrm{r}[\![\mathcal{A} \mid \mathcal{B}]\!] = \mathrm{alt}(\mathcal{A} \mid \mathcal{B},\, \mathcal{A} \mid \mathcal{B})$$

where the function 'alt' unpacks the alternative until a symbol is reached that is not an alternative.

[7]
$$\mathrm{alt}(\mathcal{B}_1 \mid \mathcal{B}_2,\, \mathcal{A}) = \mathrm{alt}(\mathcal{B}_1,\, \mathcal{A})\; \mathrm{alt}(\mathcal{B}_2,\, \mathcal{A})$$

[8]
$$\mathrm{alt}(\mathcal{B},\, \mathcal{A}) = \text{syntax}\; \mathcal{B} \rightarrow \mathcal{A}\;\; \mathrm{r}[\![\mathcal{B}]\!] \quad \textbf{otherwise}$$

Optional The *optional construct* $\mathcal{A}?$ is either empty or $\mathcal{A}$.

[9]
$$\frac{\begin{array}{c} p_1 \;=\; \rightarrow \mathcal{A}?\;, \\ p_2 \;=\; \mathcal{A} \rightarrow \mathcal{A}? \end{array}}{\mathrm{r}[\![\mathcal{A}?]\!] \;=\; \text{syntax}\; p_1\; p_2\; \mathrm{r}[\![\mathcal{A}]\!]}$$

Iteration The *iteration* operator $\mathcal{A}+$ denotes lists of *one or more* $\mathcal{A}$'s, i.e., either $\mathcal{A}$ or $\mathcal{A}\;\mathcal{A}$ or $\mathcal{A}\;\mathcal{A}\;\mathcal{A}$ or .... The iteration $\mathcal{A}*$ denotes a list of *zero or more* $\mathcal{A}$'s, i.e., ϵ (empty) or $\mathcal{A}$ or $\mathcal{A}\;\mathcal{A}$ or $\mathcal{A}\;\mathcal{A}\;\mathcal{A}$ or ... There are several ways to define such lists with productions. It is not sufficient to define a list by means of the productions

$$\begin{array}{rcr} \rightarrow \mathcal{A}* & \qquad\qquad & \mathcal{A} \rightarrow \mathcal{A}+ \\ \mathcal{A}+ \rightarrow \mathcal{A}* & & \mathcal{A}+\mathcal{A} \rightarrow \mathcal{A}+ \end{array}$$

The symbols $\mathcal{A}*$ and $\mathcal{A}+$ can be the right-hand side of any production, i.e., lists can be the result of arbitrary functions. Therefore, an $\mathcal{A}*$ expression can also contain function calls and variables. For instance, if a grammar contains the production

```
yield(Tree) -> Symbol*
```

then `yield(T1) yield(T2)` should also be an expression of type `Symbol*` (with `T1` and `T2` expression of type `Tree`). We have the following rules for the composition of list expressions.

0. A single $\mathcal{A}$ is an $\mathcal{A}+$.
1. An $\mathcal{A}+$ followed by an $\mathcal{A}+$ is an $\mathcal{A}+$.

2. An $\mathcal{A}+$ followed by an $\mathcal{A}*$ is an $\mathcal{A}+$.

3. An $\mathcal{A}*$ followed by an $\mathcal{A}+$ is an $\mathcal{A}+$.

4. An $\mathcal{A}*$ followed by an $\mathcal{A}*$ is an $\mathcal{A}*$.

5. An $\mathcal{A}*$ can be empty.

6. An $\mathcal{A}+$ is an $\mathcal{A}*$.

Productions expressing these rules are generated by the following equation. The priorities section declares the concatenation operators to be mutually left-associative. The priority prevents that the empty production and the injection are used vacuously.

[10]
$$\frac{\begin{array}{rcl} p_0 &=& \mathcal{A} \to \mathcal{A}+\ , \\ p_1 &=& \mathcal{A}+\ \mathcal{A}+ \to \mathcal{A}+\ \{\text{left}\}, \\ p_2 &=& \mathcal{A}+\ \mathcal{A}* \to \mathcal{A}+\ , \\ p_3 &=& \mathcal{A}*\ \mathcal{A}+ \to \mathcal{A}+\ , \\ p_4 &=& \mathcal{A}*\ \mathcal{A}* \to \mathcal{A}*\ \{\text{left}\}, \\ p_5 &=& \ \to \mathcal{A}*\ , \\ p_6 &=& \mathcal{A}+ \to \mathcal{A}* \end{array}}{\begin{array}{rcl} \mathrm{r}[\![\mathcal{A}*]\!] &=& \text{syntax } p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6 \\ && \text{priorities } \{\text{left} : p_1\ p_2\ p_3\ p_4\} > \{p_5\ p_6\}\ \mathrm{r}[\![\mathcal{A}]\!] \end{array}}$$

[11]
$$\mathrm{r}[\![\mathcal{A}+]\!] = \mathrm{r}[\![\mathcal{A}*]\!]$$

Iteration with Separator The *iteration with separator* operators $\{\mathcal{A}\ \mathcal{B}\}+$ and $\{\mathcal{A}\,\mathcal{B}\}*$ denote iteration of $\mathcal{A}$'s separated by $\mathcal{B}$'s. Their meaning is defined analogously to $\mathcal{A}+$ and $\mathcal{A}*$.

[12]
$$\frac{\begin{array}{rcl} p_0 &=& \mathcal{A} \to \{\mathcal{A}\,\mathcal{B}\}+\ , \\ p_1 &=& \{\mathcal{A}\,\mathcal{B}\}+\ \mathcal{B}\ \{\mathcal{A}\,\mathcal{B}\}+ \to \{\mathcal{A}\,\mathcal{B}\}+\ \{\text{left}\}, \\ p_2 &=& \{\mathcal{A}\,\mathcal{B}\}+\ \mathcal{B}\ \{\mathcal{A}\,\mathcal{B}\}* \to \{\mathcal{A}\,\mathcal{B}\}+\ , \\ p_3 &=& \{\mathcal{A}\,\mathcal{B}\}*\ \mathcal{B}\ \{\mathcal{A}\,\mathcal{B}\}+ \to \{\mathcal{A}\,\mathcal{B}\}+\ , \\ p_4 &=& \{\mathcal{A}\,\mathcal{B}\}*\ \mathcal{B}\ \{\mathcal{A}\,\mathcal{B}\}* \to \{\mathcal{A}\,\mathcal{B}\}*\ \{\text{left}\}, \\ p_5 &=& \ \to \{\mathcal{A}\,\mathcal{B}\}*\ , \\ p_6 &=& \{\mathcal{A}\,\mathcal{B}\}+ \to \{\mathcal{A}\,\mathcal{B}\}* \end{array}}{\begin{array}{rcl} \mathrm{r}[\![\{\mathcal{A}\,\mathcal{B}\}*]\!] &=& \text{syntax } p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6 \\ && \text{priorities } \{\text{left} : p_1\ p_2\ p_3\ p_4\} > p_6\ \mathrm{r}[\![\mathcal{A}\,\mathcal{B}]\!] \end{array}}$$

[13]
$$\mathrm{r}[\![\{\mathcal{A}\,\mathcal{B}\}+]\!] = \mathrm{r}[\![\{\mathcal{A}\,\mathcal{B}\}*]\!]$$

Constrained Iteration The iteration operator $\{\mathcal{A}\}n+$ denotes the iteration of *at least* n $\mathcal{A}$s. First of all we define that zero or more $\mathcal{A}$s corresponds to $*$ iteration and that one or more $\mathcal{A}$s corresponds to $+$ iteration. For integers $n \geq 2$

we define $\{\mathcal{A}\}n+$ in terms of $\{\mathcal{A}\}(n-1)+$, and eventually $\mathcal{A}+$, by productions of the form $\mathcal{A}\ \{\mathcal{A}\}(n-1)+ \to \{\mathcal{A}\}n+$.

[14] $$\{\mathcal{A}\}\ 0+ = \mathcal{A}*$$
[15] $$\{\mathcal{A}\}\ 1+ = \mathcal{A}+$$

[16] $$\frac{n \geq 2\ =\ \top,\ \ n-1\ =\ n',\ \ p\ =\ \mathcal{A}\ \{\mathcal{A}\}\ n'+ \to \{\mathcal{A}\}\ n+}{\mathrm{r}[\![\{\mathcal{A}\}\ n+]\!]\ =\ \text{syntax}\ p\ \mathrm{r}[\![\{\mathcal{A}\}\ n'+]\!]}$$

Constrained iteration is defined similarly for lists with separators.

[17] $$\{\mathcal{A}\ \mathcal{B}\}\ 0+ = \{\mathcal{A}\ \mathcal{B}\}*$$
[18] $$\{\mathcal{A}\ \mathcal{B}\}\ 1+ = \{\mathcal{A}\ \mathcal{B}\}+$$

[19] $$\frac{n \geq 2\ =\ \top,\ \ n-1\ =\ n',\ \ p\ =\ \mathcal{A}\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}\ n'+ \to \{\mathcal{A}\ \mathcal{B}\}\ n+}{\mathrm{r}[\![\{\mathcal{A}\ \mathcal{B}\}\ n+]\!]\ =\ \text{syntax}\ p\ \mathrm{r}[\![\{\mathcal{A}\ \mathcal{B}\}\ n'+]\!]}$$

Tuples For the definition of functions that return a tuple of values, new sorts have to be invented. To give sensible types to tuples the notation $\mathcal{A}\#\mathcal{B}$ is introduced. A symbol $\mathcal{A}_1\#\ldots\#\mathcal{A}_n$ denotes a tuple of $\mathcal{A}_1\ldots\mathcal{A}_n$ expressions. A tuple is written as $\langle T_1,\ldots,T_n\rangle$, where the T_i are expresions of type $\mathcal{A}_i$.

[20] $$\frac{\texttt{"<"} + \text{tup}(\mathcal{A}\ \#\ \mathcal{B}) + \texttt{">"}\ =\ \alpha,\ \ p\ =\ \alpha \to \mathcal{A}\ \#\ \mathcal{B}}{\mathrm{r}[\![\mathcal{A}\ \#\ \mathcal{B}]\!]\ =\ \text{syntax}\ p\ \mathrm{r}[\![\alpha]\!]}$$

The auxiliary function tup$[\![_]\!]$ derives the syntax of the body of the tuple by separating the symbols by commas.

[21] $$\text{tup}(\mathcal{A}\ \#\ \mathcal{B}) = \text{tup}(\mathcal{A}) + \texttt{","} + \text{tup}(\mathcal{B})$$
[22] $$\text{tup}(\mathcal{A}) = \mathcal{A}\quad \textbf{otherwise}$$

Sets The conventional notation for sets is a list of items between { and }. The operator Set$[\mathcal{A}]$ generates this notation such that if $T_1\ldots T_n$ are expressions of type $\mathcal{A}$, then $\{T_1,\ldots,T_n\}$ is an expression of type Set$[\mathcal{A}]$.

[23] $$\frac{\alpha\ =\ \texttt{"\{"}\ \{\mathcal{A}\ \texttt{","}\}*\ \texttt{"\}"},\ \ p\ =\ \alpha \to \text{Set}[\mathcal{A}]}{\mathrm{r}[\![\text{Set}[\mathcal{A}]]\!]\ =\ \mathrm{r}[\![\alpha]\!]\ \text{syntax}\ p}$$

Functions Sometimes it is convenient to pass functions around as data. The operator $(\alpha \Rightarrow \mathcal{B})$ can be used to give a type to functions. It denotes the sort of functions from α to $\mathcal{B}$. The operator generates syntax for the prefix application of a function to an argument.

[24] $$\frac{p\ =\ (\alpha \Rightarrow \mathcal{B})\ \texttt{"("}\ \alpha\ \texttt{")"} \to \mathcal{B}}{\mathrm{r}[\![(\alpha \Rightarrow \mathcal{B})]\!]\ =\ \text{syntax}\ p\ \mathrm{r}[\![\alpha\ \mathcal{B}]\!]}$$

Permutation The permutation symbol $\ll \alpha \gg$ denotes any concatenation of the symbols in α, i.e., $\beta \rightarrow \ll \alpha \gg$ if β is a permutation of α.

[25] $$\mathrm{r}[\![\ll \alpha \gg]\!] = \text{syntax perm}(\alpha)\ \mathrm{r}[\![\alpha]\!]$$

The function 'perm' generates the productions for all permutations of a set of symbols. In case the permutation consists of two elements it generates the two productions directly. In case of more elements the function 'perm3' is used to generate permutations.

[26] $$\mathrm{perm}() = \ \rightarrow \ll\ \gg$$
[27] $$\mathrm{perm}(\mathcal{A}) = \mathcal{A} \rightarrow \ll \mathcal{A} \gg$$
[28] $$\mathrm{perm}(\mathcal{A}\,\mathcal{B}) = \mathcal{A}\,\mathcal{B} \rightarrow \ll \mathcal{A}\,\mathcal{B} \gg\ \ \mathcal{B}\,\mathcal{A} \rightarrow \ll \mathcal{A}\,\mathcal{B} \gg$$
[29] $$\mathrm{perm}(\alpha) = \mathrm{perm3}(,\alpha) \quad \textbf{when}\ \mid \alpha \mid > 2 \ = \ \top$$

For each symbol in the list a production is generated with that symbol first and a permutation of the other symbols following it.

[30] $$\mathrm{perm3}(\alpha,) \ =$$

[31] $$\frac{p \ = \ \mathcal{A} \ll \alpha\,\beta \gg \rightarrow \ll \alpha\,\mathcal{A}\,\beta \gg}{\mathrm{perm3}(\alpha, \mathcal{A}\,\beta) \ = \ p +\!\!+ \ \mathrm{perm3}(\alpha\,\mathcal{A}, \beta) +\!\!+ \ \mathrm{perm}(\alpha\,\beta)}$$

It should be observed that this is not a very efficient way to implement permutation constructs. It should be adequate for permutations of 2 or 3 elements, though. What is needed in addition to the generation of these productions, is the normalization of the parse trees over these productions to a form that lists the elements in a fixed order such that semantic functions do not also have to deal with all permutations. Cameron (1993) describes an extension of LL(1) parsing for permutation operators. An alternative approach suggested by Cameron (1993) is the introduction of an intermediate symbol representing the union of the symbols in the permutation and a check after parsing that each symbol in the permutation is represented exactly once.

Discussion We could have handled several of these regular expressions by translating them to other regular expressions. For instance, optionality can be expressed by means of empty and alternative via the equation $\mathcal{A}? = ()|\mathcal{A}$. In the specification above we have chosen not follow this route. Except for a few cases involving constrained iteration.

8.2.3 Equality of Parse Trees with Lists

Since all new constructs are expressed by means of existing constructs—all regular expressions are expressed by means of generated context-free productions—there is no need to extend the definition of well-formedness of parse trees.

We do extend the definition of the equality of trees. This definition makes lists equal modulo associativity of the concatenation operators. It is the basis for matching modulo associativity. We give the equations that should be considered in matching, where a variable a^+ (a^*) ranges over all constructs of type $\mathcal{A}+$ ($\mathcal{A}*$) and ϵ_A denotes the tree constructed with $\rightarrow \mathcal{A}*$. Empty sublists are units for concatenation and can be removed.

$$\epsilon_A\ a^+ = a^+ \qquad\qquad a^+\ \epsilon_A = a^+$$
$$\epsilon_A\ a^* = a^* \qquad\qquad a^*\ \epsilon_A = a^*$$

Injections from $\mathcal{A}+$ into $\mathcal{A}*$ can be removed or lifted over concatenations.

$$[a_1^+ \rightarrow \mathcal{A}*]\ [a_1^+ \rightarrow \mathcal{A}*] = [(a_1^+\ a_2^+) \rightarrow \mathcal{A}*]$$
$$a_1^+\ [a_1^+ \rightarrow \mathcal{A}*] = a_1^+\ a_2^+$$
$$[a_1^+ \rightarrow \mathcal{A}*]\ a_1^+ = a_1^+\ a_2^+$$

Right-associative concatenations are equal to left-associative ones. Each of these expressions involves different concatenation operators.

$$a_1^+\ (a_2^+\ a_3^+) = (a_1^+\ a_2^+)\ a_3^+ \qquad\qquad a_1^*\ (a_2^+\ a_3^+) = (a_1^*\ a_2^+)\ a_3^+$$
$$a_1^*\ (a_2^*\ a_3^+) = (a_1^*\ a_2^*)\ a_3^+ \qquad\qquad a_1^*\ (a_2^+\ a_3^*) = (a_1^*\ a_2^+)\ a_3^*$$
$$a_1^+\ (a_2^*\ a_3^+) = (a_1^+\ a_2^*)\ a_3^+ \qquad\qquad a_1^+\ (a_2^*\ a_3^*) = (a_1^+\ a_2^*)\ a_3^*$$
$$a_1^+\ (a_2^+\ a_3^*) = (a_1^+\ a_2^+)\ a_3^* \qquad\qquad a_1^*\ (a_2^*\ a_3^*) = (a_1^*\ a_2^*)\ a_3^*$$

8.3 Lexical and Context-Free Syntax

The syntax of a programming language is usually divided into two levels: lexical syntax and context-free syntax. Lexical syntax is the syntax of the tokens, the words of the language, e.g., identifiers, numbers and keywords. Context-free syntax is the syntax of the sentences of a language, e.g., expressions, statements, type declarations and function definitions. The division affects both language definition and implementation. Conventionally lexical analysis is restricted to grammars that can be recognized by finite automata, whereas context-free analysis is implemented with push-down automata. Indeed, it is sometimes not clear whether the division is motivated by the implementation or by an inherent concept of lexical syntax.

In many formalisms the separation is even physical; lexical and context-free syntax are defined with completely different formalisms that are written in separate files. For instance, YACC and METAL use LEX to define lexical syntax. This means lexical definitions in the form of a number of regular expressions are defined in a separate file. Context-free and lexical definitions share a declaration of token symbols that constitutes the interface between the lexical and context-free level. The syntax definition formalism of PCCTS uses a lexical syntax similar to LEX, but provides a mechanism to include token definitions in the same file as the context-free syntax definition. In SDF lexical and context-free syntax are integrated in one formalism, but still uses different semantics for both. All these approaches have in common that the distinction between lexical and context-free syntax is identified with the distinction between regular and context-free grammars.

In SDF2 the inherent distinction between the two categories is that context-free symbols can be *separated by layout*, while lexical symbols cannot. Beyond that difference there is none. The exact same features should be available for the definition of lexical and context-free syntax.

New in this approach is that we provide a uniform notation for the definition of lexical and context-free syntax by means of context-free productions. Grammars for lexical and context-free syntax are normalized to the context-free grammars of the kernel. The distinction between lexical and context-free syntax is completely expressed in the resulting productions.

By treating lexical and context-free syntax identically, every extension that is defined for one is also applicable to the other. For instance, in §8.1 we defined priorities for disambiguation. In Heering *et al.* (1989) these are only defined for context-free syntax. As result of our approach we can also provide lexical disambiguation through priorities. Similarly the regular operators introduced in §8.2 can be used in the definition of both lexical and context-free syntax.

In addition to lexical syntax we also define variables. Variable schemes are used in the specification of the semantics of a language. We also introduce the notion of lexical variables that range over constructs introduced in lexical syntax grammars.

The extension in this section is called *Basic Sdf* because it covers the basic idea of the original SDF: integration of lexical and context-free syntax in one formalism.

Example 8.3.1 The following definition introduces a simple expression language with variables and addition.

```
sorts Id Exp
lexical syntax
  [\ \t\n]    -> LAYOUT
  [a-z]+      -> Id
context-free syntax
  Id          -> Exp
  Exp "+" Exp -> Exp {left}
variables
  [i]   -> Id
  [xyz] -> Exp
```

The lexical syntax section defines the syntax of layout as spaces, tabs and newlines and identifiers as lists of one or more lowercase letters. The division in lexical and context-free syntax entails that whitespace can occur between expressions, but not between the letters of an identifier.

To illustrate the power of the integration of lexical and context-free syntax we can extend the layout convention above by introducing C-like comments consisting of a string of comment words between `/*` and `*/`, as follows:

```
sorts ComWord Comment
lexical syntax
  ~[\ \t\n\|\/\*]+ -> ComWord
context-free syntax
  "/*" ComWord* "*/" -> Comment
  Comment            -> LAYOUT
```

Because the definition of comments is part of the context-free syntax, comment words can be separated by layout, including layout. This means that we have specified nested comments, which is useful when commenting out pieces of code already containing comment.

We can extend the defition of comments further to include syntactically correct expressions between bars as comment words.

```
context-free syntax
  "|" Exp "|" -> ComWord
```

For instance, the text

```
 a + b /* an expression |x + y| denotes
          the addition of |x| and |y| */ + c
```

is a syntactically correct expression over the grammar above denoting the expression `a+b+c` with some comment after `b`. In the conventional setting of a separate scanner and parser this would require a call to the parser from the scanner. One application of syntactically correct program fragments in comments is in typesetting programs for documentation. The typesetting algorithms applied to the real program text can also be applied in typesetting the expressions in comments and crossreferences to program variables can be extended to variables occurring comments. □

8.3.1 Syntax

The grammar constructors 'lexical syntax' and 'context-free syntax' introduce the syntax of lexical constructs and context-free constructs, respectively. The grammar constructors 'variables' and 'lexical variables' introduce the syntax of variables over context-free symbols and variables over lexical symbols, respectively. The symbol constructors ⟨_-LEX⟩, ⟨_-CF⟩ and ⟨_-VAR⟩ are used to indicate lexical symbols, context-free symbols and variable symbols, respectively. The special symbol `LAYOUT` is used to define layout.

module Basic-Sdf-Syntax
imports Kernel-Sdf-Syntax[7.3.1]
exports
  context-free syntax
    "lexical" "syntax" Productions → Grammar
    "context-free" "syntax" Productions → Grammar
    "variables" Productions → Grammar
    "lexical" "variables" Productions → Grammar

“<” Symbol “-CF” “>”	$\to$ Symbol
“<” Symbol “-LEX” “>”	$\to$ Symbol
“<” Symbol “-VAR” “>”	$\to$ Symbol
“LAYOUT”	$\to$ Symbol

8.3.2 Normalization

The normalization function defined below expresses the meaning of lexical and context-free syntax by merging them into a single grammar. To avoid interference between the two levels, the symbols in the lexical syntax are renamed into $\langle$_-LEX$\rangle$ symbols and the symbols in the context-free syntax are renamed into $\langle$_-CF$\rangle$ symbols. These ideas are illustrated in the following example.

Example 8.3.2 The grammar in Example 8.3.1 is mapped to the following grammar in which lexical and context-free syntax have been merged.

```
sorts Id Exp ComWord Comment
syntax
  <[a-z]+-LEX>                     -> <Id-LEX>
  <Id-LEX>                         -> <Id-CF>
  <Id-CF>                          -> <Exp-CF>
  <Exp-CF> <LAYOUT?-CF> "+"
           <LAYOUT?-CF> <Exp-CF>   -> <Exp-CF> {left}
  [i]                              -> <<Id-CF>-VAR>
  <<Id-CF>-VAR>                    -> <Id-CF>
  [xyz]                            -> <<Exp-CF>-VAR>
  <<Exp-CF>-VAR>                   -> <Exp-CF>
  <~[\ \t\n\|\047\*]+-LEX>         -> <ComWord-LEX>
  <ComWord-LEX>                    -> <ComWord-CF>
  "|" <LAYOUT?-CF> <Exp-CF>
      <LAYOUT?-CF> "|"             -> <ComWord-CF>
  "/*" <LAYOUT?-CF> <ComWord*-CF>
       <LAYOUT?-CF> "*/"           -> <Comment-CF>
  [\ \t\n]                         -> <LAYOUT-LEX>
  <LAYOUT-LEX>                     -> <LAYOUT-CF>
  <Comment-CF>                     -> <LAYOUT-CF>
  <LAYOUT-CF> <LAYOUT-CF>          -> <LAYOUT-CF> {left}
                                   -> <LAYOUT?-CF>
  <LAYOUT-CF>                      -> <LAYOUT?-CF>
```

The symbols in lexical productions are renamed into $\langle$_-LEX$\rangle$ symbols. The symbols in context-free productions are renamed into $\langle$_-CF$\rangle$ symbols. The connection between lexical and context-free syntax is made by an injection from each $\langle\mathcal{A}$-LEX$\rangle$ symbol into the corresponding $\langle\mathcal{A}$-CF$\rangle$ symbol. □

The following module makes these ideas formal by introducing the normalization function $b[\![\_]\!]$.

module Basic-Sdf-Normalization
imports Basic-Sdf-Syntax[8.3.1] Regular-Sdf-Syntax[8.2.1] Priority-Sdf-Syntax[8.1.1] Kernel-Sdf-Normalization[7.3.3]
exports
context-free syntax

"b⟦" Grammar "⟧"	→ Grammar
"b$_{\mathrm{aux}}$⟦" Grammar "⟧"	→ Grammar
"<" Symbols "`-LEXs`" ">"	→ Symbols
"<" Production "`-LEX`" ">"	→ Production
"<" Productions "`-LEXs`" ">"	→ Productions
"<" Grammar "`-LEX`" ">"	→ Grammar
"<" Symbols "`-CFs`" ">"	→ Symbols
"<" Production "`-CF`" ">"	→ Production
"<" Productions "`-CFs`" ">"	→ Productions
"<" Grammar "`-CF`" ">"	→ Grammar
"<" Productions "`-VARs`" ">"	→ Productions
"<" Productions "`-LEXVARs`" ">"	→ Productions

equations

The normalization function $\mathrm{b}[\![\_]\!]$ integrates lexical and context-free syntax. It applies the auxiliary function $\mathrm{b}_{\mathrm{aux}}$ to each subgrammar of a grammar to transform lexical and context-free sections into normal production sections by renaming symbols and separating context-free symbols by $\langle\texttt{LAYOUT?-CF}\rangle$, which entails that two tokens can optionally be separated by $\langle\texttt{LAYOUT?-CF}\rangle$. Context-free layout is a list of lexical layout. Concatenation of layout is defined by the production added by the function $\mathrm{b}[\![\_]\!]$.

[1] $$\frac{p = \langle\texttt{LAYOUT-CF}\rangle\ \langle\texttt{LAYOUT-CF}\rangle \to \langle\texttt{LAYOUT-CF}\rangle\ \{\mathrm{left}\}}{\mathrm{b}[\![\mathcal{G}]\!]\ =\ \mathrm{syntax}\ p\ \mathrm{b}_{\mathrm{aux}}[\![\mathcal{G}]\!]}$$

The default rule declares that unless otherwise stated $\mathrm{b}_{\mathrm{aux}}$ does not affect a grammar. Below we deal with the exceptions.

[2] $$\mathrm{b}_{\mathrm{aux}}[\![\emptyset]\!] = \emptyset$$
[3] $$\mathrm{b}_{\mathrm{aux}}[\![\mathcal{G}_1\ \mathcal{G}_2]\!] = \mathrm{b}_{\mathrm{aux}}[\![\mathcal{G}_1]\!]\ \mathrm{b}_{\mathrm{aux}}[\![\mathcal{G}_2]\!]$$
[4] $$\mathrm{b}_{\mathrm{aux}}[\![\mathcal{G}]\!] = \mathcal{G} \quad \textbf{otherwise}$$

Lexical Syntax Lexical syntax grammars are translated to normal syntax grammars by encoding the symbols of the grammar to $\langle\mathcal{A}\texttt{-LEX}\rangle$ symbols. Furthermore, for each symbol appearing in a lexical syntax section an injection from the lexical into the context-free symbol is added.

[5] $$\mathrm{b}_{\mathrm{aux}}[\![\mathrm{lexical\ syntax}\ p^*]\!] = \langle\mathrm{syntax}\ p^*\texttt{-LEX}\rangle$$

[6] $$\langle\alpha\texttt{-LEXs}\rangle = \quad \textbf{when}\ \ \alpha =$$

[7] $\langle \mathcal{A}\texttt{-LEXs} \rangle = \langle \mathcal{A}\texttt{-LEX} \rangle$
[8] $\langle \alpha^+ \ \beta^+\texttt{-LEXs} \rangle = \langle \alpha^+\texttt{-LEXs} \rangle + \!\!+ \ \langle \beta^+\texttt{-LEXs} \rangle$
[9] $\langle \alpha \rightarrow \mathcal{A}\ \$\texttt{-LEX} \rangle = \langle \alpha\texttt{-LEXs} \rangle \rightarrow \langle \mathcal{A}\texttt{-LEX} \rangle\ \$$
[10] $\langle \alpha \rightarrow \mathcal{A}\ \$\texttt{-LEXs} \rangle = \langle \alpha \rightarrow \mathcal{A}\ \$\texttt{-LEX} \rangle\ \langle \mathcal{A}\texttt{-LEXs} \rangle \rightarrow \langle \mathcal{A}\texttt{-CF} \rangle$
[11] $\langle p^*\texttt{-LEXs} \rangle = \quad \textbf{when} \ \ p^* =$
[12] $\langle p_1^+ \ p_2^+\texttt{-LEXs} \rangle = \langle p_1^+\texttt{-LEXs} \rangle + \!\!+ \ \langle p_2^+\texttt{-LEXs} \rangle$
[13] $\langle \text{syntax}\ p^*\texttt{-LEX} \rangle = \text{syntax}\ \langle p^*\texttt{-LEXs} \rangle$
[14] $\langle \mathcal{G}_1\ \mathcal{G}_2\texttt{-LEX} \rangle = \langle \mathcal{G}_1\texttt{-LEX} \rangle\ \langle \mathcal{G}_2\texttt{-LEX} \rangle$
[15] $\langle \emptyset\texttt{-LEX} \rangle = \emptyset$

Context-free Syntax Context-free syntax is treated similarly to lexical syntax. All symbols in the production are mapped to $\langle \mathcal{A}\texttt{-CF} \rangle$ symbols. The important difference is that each adjacent pair of symbols in the left-hand side of a production is separated by the symbol $\langle \texttt{LAYOUT?-CF} \rangle$.

[16] $\text{b}_{\text{aux}}[\![\text{context-free syntax}\ p^*]\!] = \langle \text{syntax}\ p^*\texttt{-CF} \rangle$

[17] $\langle \alpha\texttt{-CFs} \rangle = \quad \textbf{when} \ \ \alpha =$
[18] $\langle \mathcal{A}\texttt{-CFs} \rangle = \langle \mathcal{A}\texttt{-CF} \rangle$
[19] $\langle \alpha^+ \ \beta^+\texttt{-CFs} \rangle = \langle \alpha^+\texttt{-CFs} \rangle + \!\!+ \ \langle \texttt{LAYOUT?-CF} \rangle + \!\!+ \ \langle \beta^+\texttt{-CFs} \rangle$
[20] $\langle \alpha \rightarrow \mathcal{A}\ \$\texttt{-CF} \rangle = \langle \alpha\texttt{-CFs} \rangle \rightarrow \langle \mathcal{A}\texttt{-CF} \rangle\ \$$
[21] $\langle p\texttt{-CFs} \rangle = \langle p\texttt{-CF} \rangle$
[22] $\langle p^*\texttt{-CFs} \rangle = \quad \textbf{when} \ \ p^* =$
[23] $\langle p_1^+ \ p_2^+\texttt{-CFs} \rangle = \langle p_1^+\texttt{-CFs} \rangle + \!\!+ \ \langle p_2^+\texttt{-CFs} \rangle$
[24] $\langle \text{syntax}\ p^*\texttt{-CF} \rangle = \text{syntax}\ \langle p^*\texttt{-CFs} \rangle$
[25] $\langle \mathcal{G}_1\ \mathcal{G}_2\texttt{-CF} \rangle = \langle \mathcal{G}_1\texttt{-CF} \rangle\ \langle \mathcal{G}_2\texttt{-CF} \rangle$
[26] $\langle \emptyset\texttt{-CF} \rangle = \emptyset$

Variables Variables and lexical variables grammars introduce tokens that have the status of variables. The symbol constructor $\langle \mathcal{A}\texttt{-VAR} \rangle$ is used to denote variables over the symbol $\mathcal{A}$. The left-hand sides of variable productions are interpreted as lexical syntax. The lexical value produced by such a left-hand side is given the type of a variable over the symbol in the right-hand side of the production. For each production in a variables grammar, two productions are generated. The first interprets the left-hand side of the production as a lexical pattern, i.e., the symbols on the left-hand side are lexical symbols and no layout between symbols can occur. The right-hand side is $\langle \langle \mathcal{A}\texttt{-CF} \rangle\texttt{-VAR} \rangle$ indicating that the pattern is a variable over the context-free symbols $\mathcal{A}$. The second production injects $\langle \mathcal{A}\texttt{-CF} \rangle$ variables into $\langle \mathcal{A}\texttt{-CF} \rangle$ such that a variable can occur whereever an $\langle \mathcal{A}\texttt{-CF} \rangle$ can occur.

[27] $\text{b}_{\text{aux}}[\![\text{variables}\ p^*]\!] = \text{syntax}\ \langle p^*\texttt{-VARs} \rangle$
[28] $\langle \texttt{-VARs} \rangle =$
[29] $\langle p_1^+ \ p_2^+\texttt{-VARs} \rangle = \langle p_1^+\texttt{-VARs} \rangle + \!\!+ \ \langle p_2^+\texttt{-VARs} \rangle$

[30] $$\langle \alpha \rightarrow \mathcal{A}\ \$\texttt{-VARs} \rangle \ = \ \langle \alpha\texttt{-LEXs} \rangle \rightarrow \langle \langle \mathcal{A}\texttt{-CF} \rangle\texttt{-VAR} \rangle\ \$ \quad \langle \langle \mathcal{A}\texttt{-CF} \rangle\texttt{-VAR} \rangle \rightarrow \langle \mathcal{A}\texttt{-CF} \rangle$$

Lexical variables are treated similarly, but their result sort is the corresponding lexical sort.

[31] $\text{baux}[\![\text{lexical variables } p^*]\!] = \text{syntax } \langle p^*\texttt{-LEXVARs}\rangle$

[32] $\langle\texttt{-LEXVARs}\rangle =$

[33] $\langle p_1^+ \; p_2^+\texttt{-LEXVARs}\rangle = \langle p_1^+\texttt{-LEXVARs}\rangle +\!\!+ \langle p_2^+\texttt{-LEXVARs}\rangle$

[34] $$\langle \alpha \to \mathcal{A} \;\texttt{\$-LEXVARs}\rangle \;=\; \langle\alpha\texttt{-LEXs}\rangle \to \langle\langle\mathcal{A}\texttt{-LEX}\rangle\texttt{-VAR}\rangle \;\texttt{\$} \\ \langle\langle\mathcal{A}\texttt{-LEX}\rangle\texttt{-VAR}\rangle \to \langle\mathcal{A}\texttt{-LEX}\rangle$$

Ordering Grammars The following equations specify the ordering of grammars, where the following order is obtained: lexical syntax, context-free syntax, lexical variables, and variables. We only show two of the equations, the other cases are similar.

[35] $$\frac{\mathcal{G}_1 = \text{context-free syntax } p_1^*,\;\; \mathcal{G}_2 = \text{context-free syntax } p_2^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \;=\; \langle \text{context-free syntax } p_1^* \; p_2^*, \emptyset\rangle}$$

[36] $$\frac{\mathcal{G}_1 = \text{context-free syntax } p_1^*,\;\; \mathcal{G}_2 = \text{lexical syntax } p_2^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \;=\; \langle \text{lexical syntax } p_2^*, \text{context-free syntax } p_1^*\rangle}$$

8.3.3 Parse Trees

Since we have expressed the meaning of lexical syntax and context-free syntax in terms of normal syntax productions, we do not have to extend the definition of parse trees, except for the encoding of symbols and grammars in the ATerm format. See §A.2 for the encoding and decoding of the newly introduced constructs. This entails that trees for lexical and context-free syntax have the same form. In particular, the structure assigned to lexical tokens by the grammar is retained in parse trees for tokens.

We will refine the equality predicate on trees such that layout is ignored. In considering whether two trees are equivalent it is likely that we do not want to consider layout. For this purpose it is not required to first translate a parse tree to an abstract syntax tree. It suffices to define two arbitrary layout trees as equivalent, as is done in the following extension of the equality predicate on trees.

module Basic-Sdf-Equality
imports Kernel-Sdf-Equality$^{7.5.8}$ Basic-Sdf-ATerms$^{A.2.2}$
Regular-Sdf-ATerms$^{A.2.2}$
equations

[1] $$\frac{\text{symbol}(\text{type}(T_1)) = \langle\texttt{LAYOUT-CF}\rangle,\;\; \text{symbol}(\text{type}(T_2)) = \langle\texttt{LAYOUT-CF}\rangle}{T_1 \doteq T_2 \;=\; \top}$$

[2] $$\frac{\text{symbol}(\text{type}(T_1)) = \langle\texttt{LAYOUT?-CF}\rangle,\;\; \text{symbol}(\text{type}(T_2)) = \langle\texttt{LAYOUT?-CF}\rangle}{T_1 \doteq T_2 \;=\; \top}$$

8.3.4 Discussion

Lexical Layout In some languages, such as `FORTRAN`, tokens can contain some kind of layout. In Heering *et al.* (1989) the symbol `IGNORE` is introduced for this purpose. This can be dealt with by separating the symbols in a lexical production by a *lexical layout* symbol just as this is done with context-free productions. This is not done in the current version because for most languages this is not necessary, but it is straightforward to add this feature to the normalization above.

Implementation A conventional implementation of parsers for lexical and context-free syntax is based on a separate scanner and parser. Such an implementation can be achieved for grammars as introduced here by separating productions for ⟨_-LEX⟩ and ⟨_-VAR⟩ symbols from productions for ⟨_-CF⟩ symbols and generating a scanner based on finite automata for the first set of productions and by generating a parser for the second set of productions based on push-down automata. Scanner and parser communicate through some shared buffer-like data-structure. A requirement for this approach is that the lexical productions form a regular grammar. This can be enforced by specifying static constraints on lexical productions.

The parser generator for SDF2 described in Chapter 3 does not depend on a separate scanner. Instead 'lexical analysis', i.e., parsing according to the productions for ⟨_-LEX⟩ symbols, is incorporated in the parser. To cope with ambiguities and lookahead, generalized LR parsing is used. A similar approach is described by Salomon and Cormack (1989, 1995) under the name scannerless parsing using conventional LR techniques.

8.4 Restrictions

When a distinction is made between lexical and context-free syntax, lexical ambiguities have to be solved before tokens can be sent to the parser. This is usually done by applying rules such as 'prefer longest match', 'prefer keywords' and 'prefer variables'.

By removing this distinction, as we did in the previous section, lexical ambiguities can be dealt with in the same way as context-free ambiguities. For example, in §8.1 we defined disambiguation by priorities, which applies both to lexical and context-free syntax. Furthermore, many lexical ambiguities are solved by considering the context in which tokens occur. For instance, the well-known problem of distinguishing an occurrence of the subrange 1..10 from two consecutive occurrences of the real numbers 1. and .10 in Pascal is solved automatically, because ranges and reals do not occur in the same context in the grammar.

However, not all lexical ambiguities can be solved by context or by means of priorities. Some lexical ambiguities need to be solved by rules such as 'prefer longest match' and 'prefer literals'.

In this section we introduce two extensions of context-free grammars that are aimed at lexical disambiguation: *follow restrictions* and *reject productions*. A follow restriction $\mathcal{A}\not\!-cc$ declares that the symbol $\mathcal{A}$ can not be followed by any character in the character class cc. A reject production $\alpha \rightarrow \mathcal{A}$ {reject} declares that any tree of type $\mathcal{A}$ should be rejected if there exists a tree with the same yield that has this reject production as root production. These constructs suffice for expressing most lexical disambiguation rules.

Example 8.4.1 The definition of a simple expression language with nested comments in Example 8.3.1 contains two lexical ambiguities. First, the definition of lists of comment-words `ComWord*` is ambiguous. The string `abc` can be one comment word (a list of characters from the class `~[\ \t\n\|\/]`), but it can also be considered as a list of two comment-words `ab` and `c` or as `a` and `bc` or as `a` and `b` and `c`. We want to express that the longest possible comment-word should be selected. Second, the definition of identifiers and variables for identifiers and expressions overlap, i.e., `x` can be either an identifier or an expression of sort `Exp`. Here we want to express the rule 'prefer variables' that selects a variable over a lexical. These ambiguities are solved by the following rules:

```
lexical restrictions
  ComWord -/- ~[\ \t\n\|\/]
syntax
  <<Id-CF>-VAR>  -> <Id-CF> {reject}
  <<Exp-CF>-VAR> -> <Id-CF> {reject}
```

The first rule states that a comment-word should not be followed by any of the characters in `~[\ \t\n\|\/]`. This solves the problem because it rules out all parses, except the one in which `abc` is one word. The last two rules state that variables should be preferred over identifiers. □

8.4.1 Syntax

A follow restriction has the form $\mathcal{A}\not\!-cc$. Follow restrictions are declared in a grammar starting with the keyword 'restrictions' followed by a list of restrictions. A reject production is a normal production attributed with the attribute 'reject'.

module Restrictions-Sdf-Syntax
imports CC-Sdf-Syntax$^{7.4.2}$
exports
  sorts Restriction Restrictions
  context-free syntax
    Symbols "$\not\!-$" CharClass → Restriction
    Restriction∗ → Restrictions
    "restrictions" Restrictions → Grammar
    "reject" → Attribute
  variables
    "*restr*"[*0-9′*]∗ → Restriction
    "*restr*∗"[*0-9′*]∗ → Restriction∗

“*restr*+”[*0-9′*]∗ → Restriction+

8.4.2 Projection

The function ‘R’ gives the restrictions of a grammar. The function $\pi_{\mathcal{A}}$ looks up the restrictions for some symbol.

module Restrictions-Sdf-Projection
imports Restrictions-Sdf-Syntax[8.4.1]
exports
context-free syntax

Restrictions “++” Restrictions	→ Restrictions	{**right**}
“R”(Grammar)	→ Restrictions	
π “_” Symbol “(” Restrictions “)”	→ CharClass	

equations
Concatenation of restrictions.

$$[1] \quad restr_1^* \mathbin{+\!\!+} restr_2^* = restr_1^* \ restr_2^*$$

The restrictions of a grammar.

$$[2] \quad \mathrm{R}(\text{restrictions } restr^*) = restr^*$$
$$[3] \quad \mathrm{R}(\mathcal{G}_1 \ \mathcal{G}_2) = \mathrm{R}(\mathcal{G}_1) \mathbin{+\!\!+} \mathrm{R}(\mathcal{G}_2)$$
$$[4] \quad \mathrm{R}(\mathcal{G}) = \quad \textbf{otherwise}$$

The restrictions for a symbol.

$$[5] \quad \pi_{\mathcal{A}}() = []$$
$$[6] \quad \pi_{\mathcal{A}}(\not- cc \ restr^*) = \pi_{\mathcal{A}}(restr^*)$$
$$[7] \quad \pi_{\mathcal{A}}(\mathcal{A} \ \alpha \not- cc \ restr^*) = cc \vee \pi_{\mathcal{A}}(\alpha \not- cc \ restr^*)$$
$$[8] \quad \pi_{\mathcal{A}}(\mathcal{B} \ \alpha \not- cc \ restr^*) = \pi_{\mathcal{A}}(\alpha \not- cc \ restr^*) \quad \textbf{otherwise}$$

8.4.3 Normalization

No special normalization is needed for restrictions except the normal ordering and merging of grammars.

module Restrictions-Sdf-Normalization
imports Restrictions-Sdf-Syntax[8.4.1] CC-Sdf-Normalization[7.4.2]
equations
Merging and ordering of grammars.

$$[1] \quad \text{restrictions } restr_1^* \diamond \text{restrictions } restr_2^* = \langle \text{restrictions } restr_1^* \ restr_2^*, \emptyset \rangle$$
$$[2] \quad \text{restrictions } restr^* \diamond \text{syntax } p^* = \langle \text{syntax } p^*, \text{restrictions } restr^* \rangle$$

8.4.4 Discussion

The disambiguation rules presented above are derived from similar rules introduced by Salomon and Cormack (1989). The adjacency restriction of Salomon and Cormack (1989) is more general. It has the form $\mathcal{A} \not- \mathcal{B}$ and declares that symbols $\mathcal{A}$ and $\mathcal{B}$ should not be adjacent. Since this may require arbitrary long lookahead, we have chosen for the simpler follow restrictions, which can be implemented by restricting the lookahead of productions. The implementation of reject productions in SGLR parsing described in Chapter 3 is more general than the implementation based on noncanonical SLR(1) parsing of Salomon and Cormack (1989).

We have not presented the interpretation of follow restrictions and reject productions as disambiguation devices. Follow restrictions can be interpreted as an extension of the well-formedness predicate on parse trees. If a follow restriction applies to a symbol, for any tree with that symbol as type, the character immediately next to the right-most character of its yield should not be contained in the restriction. For a discussion of the semantics of reject productions see Chapter 3.

In the current situation lexical disambiguation rules have to be invented by the user. In SDF lexical disambiguation is completely taken care of in the scanner by means of a number of heuristics. These heuristics do cause problems in a number of cases. Therefore, it is attractive to have complete control over lexical disambiguation as is provided by restrictions and reject productions introduced here. However, it would be desirable if for most cases the necessary restrictions could be derived automatically from the grammar. Although some schemes have been considered, it is not yet clear how the derivation rules should be defined.

9

Renaming and Modularization

In this chapter we introduce a module mechanism for reusing parts of syntax definitions. In order to adapt imported modules to specific applications and to avoid name clashes, a renaming mechanism is provided that can be used to rename symbols and productions. The renaming mechanism is also used in the definition of symbol aliases that can be used to define abbreviated names for large regular expressions. Renamings are also used to define symbol parameterization of modules.

9.1 Renamings

In the previous sections we have presented a number of features that enable more concise definition of syntax than plain context-free grammars. The grammars that can be defined are long monolithic lists of productions. To promote reuse of grammars we will introduce in §9.3 a module layer on top of grammars, such that parts of a language definition can be reused in various other definitions. To make the opportunities for reuse even greater we introduce here a renaming operator on grammars. Renamings enable the adaptation of a generic grammar to specific needs by renaming sorts and productions. A renaming is either a symbol renaming $\mathcal{A} \Rightarrow \mathcal{B}$ that renames $\mathcal{A}$ to $\mathcal{B}$ or a production renaming $p_1 \Rightarrow p_2$ that renames p_1 to p_2. For example, the renaming

```
[Key => Var   Value => Term   Table => Subst
 lookup(Table, Key) -> Value
     => Subst "[" Var "]" -> Term ]
```

specifies the renaming of symbols `Key` and `Value` to `Var` and `Term`, respectively, and the renaming of the production `lookup(Table, Key) -> Value` to `Subst "[" Var "]" -> Term`.

Once we have defined renamings on grammars we can apply them in several situations: renaming of imported modules, symbol parameters of modules and symbol aliases. These will be the subject of the next sections.

9.1.1 Syntax

A renaming is a list of symbol renamings of the form $\mathcal{A} \Rightarrow \mathcal{B}$ and production renamings of the form $p_1 \Rightarrow p_2$.

module Renaming-Sdf-Syntax
imports Kernel-Sdf-Syntax[7.3.1]
exports
  sorts Renaming Renamings
  context-free syntax
    "[" Renaming∗ "]" → Renamings
    Symbol "⇒" Symbol → Renaming
    Production "⇒" Production → Renaming
  variables
    "ρ"[*0-9′*]∗ → Renamings
    "ρ" "∗"[*0-9′*]∗ → Renaming∗
    "ρ" "+"[*0-9′*]∗ → Renaming+

The only requirement on production renamings is that if $\alpha \to \mathcal{A} \Rightarrow \beta \to \mathcal{B}$ is a renaming, then α and β should be similar, i.e., the non-terminal parts should correspond. This entails that production renamings can only be used to rename literals between the arguments—the 'syntax'—and not the order of the arguments.

9.1.2 Projection

We define two projection functions for looking up the value of a symbol or a production in a list of renamings.

module Renaming-Sdf-Projection
imports Renaming-Sdf-Syntax[9.1.1]
exports
  context-free syntax
    Renamings "++" Renamings → Renamings {**right**}
    "π" "_" Symbol "(" Renamings ")" → Symbol
    "π" "_" Production "(" Renamings ")" → Production
    "(" Symbols "⇒" Symbols ")" → Renamings
equations
Concatenation of renamings.

[1] $$[\rho_1^*] +\!\!+ [\rho_2^*] = [\rho_1^* \ \rho_2^*]$$

Looking up the renaming of a symbol in a list of renamings.

[2] $$\pi_{\mathcal{A}}([\mathcal{A} \Rightarrow \mathcal{B}\ \rho^*]) = \mathcal{B}$$

[3] $$\pi_{\mathcal{A}}([\mathcal{A}' \Rightarrow \mathcal{B}\ \rho^*]) = \pi_{\mathcal{A}}([\rho^*]) \quad \textbf{otherwise}$$

[4] $$\pi_{\mathcal{A}}([p \Rightarrow p'\ \rho^*]) = \pi_{\mathcal{A}}([\rho^*])$$

[5] $$\pi_{\mathcal{A}}([]) = \mathcal{A}$$

Looking up the renaming of a production in a list of renamings.

[6] $$\pi_p([]) = p$$
[7] $$\pi_p([p \Rightarrow p'\ \rho^*]) = p'$$
[8] $$\pi_{\alpha \,\rightarrow\, \mathcal{A}\ \$}([\alpha \rightarrow \mathcal{A}\ \Rightarrow \beta \rightarrow \mathcal{B}\ \ \rho^*]) = \beta \rightarrow \mathcal{B}\ \$$$
[9] $$\pi_p([p' \Rightarrow p''\ \rho^*]) = \pi_p([\rho^*]) \quad \textbf{otherwise}$$
[10] $$\pi_p([\mathcal{A} \Rightarrow \mathcal{B}\ \rho^*]) = \pi_p([\rho^*])$$

Abbreviation of a renaming of a list of symbols into another list of symbols.

[11] $$(\ \Rightarrow\ ) = []$$
[12] $$(\alpha \Rightarrow\ ) = []$$
[13] $$(\ \Rightarrow \beta) = []$$
[14] $$(\mathcal{A}\ \alpha \Rightarrow \mathcal{B}\ \beta) = [\mathcal{A} \Rightarrow \mathcal{B}] +\!\!+ (\alpha \Rightarrow \beta)$$

This will be used for the instantiation of a list of formal parameters with a list of actual parameters.

9.1.3 Normalization

Now we can define the application of a renaming to a grammar. For each sort $\mathcal{S}$ we define an application function $(\mathcal{S})$ Renamings $\rightarrow \mathcal{S}$ that applies a renaming to constructs of sort $\mathcal{S}$. We start by defining the renaming of symbols and productions. The rest is mainly a distribution of the renamings function over the constructs building a grammar.

module Kernel-Sdf-Renaming
imports Renaming-Sdf-Projection$^{9.1.2}$ Kernel-Sdf-Projection$^{7.3.2}$
exports
  context-free syntax
    "(" Symbol ")" Renamings $\rightarrow$ Symbol
    "[" Symbol "]" Renamings $\rightarrow$ Symbol
    "(" Symbols ")*" Renamings $\rightarrow$ Symbols
    "(" Production ")" Renamings $\rightarrow$ Production
    "(" Productions ")*" Renamings $\rightarrow$ Productions
    "(" Grammar ")" Renamings $\rightarrow$ Grammar
equations
Renaming a symbol. If the symbol is defined in the renaming it is replaced by its value in the renaming. Otherwise, the renaming is applied recursively to the subsymbols of the symbol, which is done by the function $[\_]\rho$.

[1] $$(\mathcal{A})\ \rho = \mathcal{B} \quad \textbf{when}\ \ \pi_{\mathcal{A}}(\rho) = \mathcal{B},\ \ \mathcal{A} \neq \mathcal{B}$$
[2] $$(\mathcal{A})\ \rho = [\mathcal{A}]\ \rho \quad \textbf{otherwise}$$

Renaming a production works similarly. If the production is defined in the renaming it is replaced by its value. Otherwise, its symbols are renamed.

[3] $$(p)\ \rho = p' \quad \textbf{when}\ \ \pi_p(\rho) = p',\ \ p' \neq p$$
[4] $$(\alpha \rightarrow \mathcal{A}\ \$)\ \rho = (\alpha\ )*\ \rho \rightarrow (\mathcal{A})\ \rho\ \$ \quad \textbf{otherwise}$$

For all other grammar constructs, renaming is a homomorphism that applies the renamings to symbols and productions contained in the structure.
Renaming lists of symbols.

[5] $(\alpha\ )*\ \rho = \quad \textbf{when}\ \ \alpha =$

[6] $(\alpha^+\ \beta^+\ )*\ \rho = (\alpha^+\ )*\ \rho +\!\!+ (\beta^+\ )*\ \rho$

[7] $(\mathcal{A}\ )*\ \rho = (\mathcal{A})\ \rho$

Renaming lists of productions.

[8] $(p\ )*\ \rho = (p)\ \rho$

[9] $(p_1^*\ )*\ \rho = \quad \textbf{when}\ \ p_1^* =$

[10] $(p_1^+\ p_2^+\ )*\ \rho = (p_1^+\ )*\ \rho +\!\!+ (p_2^+\ )*\ \rho$

Renaming grammars.

[11] $(\emptyset)\ \rho = \emptyset$

[12] $(\mathcal{G}_1\ \mathcal{G}_2)\ \rho = (\mathcal{G}_1)\ \rho\ (\mathcal{G}_2)\ \rho$

[13] $(\text{syntax}\ p^*)\ \rho = \text{syntax}\ (p^*\ )*\ \rho$

The application of a renaming to a renaming denotes the composition of the renamings.

module Renaming-Sdf-Renaming
imports Kernel-Sdf-Renaming$^{9.1.3}$
exports
  context-free syntax
    "(" Renamings ")" Renamings → Renamings
equations

A renaming ρ_2 applied to a renaming ρ_1 $(\rho_1)\rho_2$ denotes the composition of the renamings, i.e., $(x)(\rho_1)\rho_2 = ((x)\rho_1)\rho_2$. This can be expressed by means of a single renaming by renaming the targets of ρ_1 with ρ_2 and adding ρ_2 at the end of the list of renamings.

[1] $([])\ \rho = \rho$

[2] $([\mathcal{A} \Rightarrow \mathcal{B}\ \rho^*])\ \rho = [\mathcal{A} \Rightarrow (\mathcal{B})\ \rho] +\!\!+ ([\rho^*])\ \rho$

[3] $([p \Rightarrow p'\ \rho^*])\ \rho = [p \Rightarrow (p')\ \rho] +\!\!+ ([\rho^*])\ \rho$

For each of the extensions of the kernel we have to extend the renaming functions to the new constructors. See §A.3 for the specification of these extensions.

9.1.4 Renaming Trees

If well-formed trees exist over a grammar that is renamed, the trees have to be renamed as well, if they have to be reused in the same context as the renamed grammar. For example, if equations over a grammar are defined, the equations must be renamed as well. Therefore, we extend the definition of renaming to parse trees.

module Renaming-Sdf-Trees
imports Kernel-Sdf-Trees[7.5.5] CC-Sdf-Trees[7.5.6] Basic-Sdf-Trees
Regular-Sdf-Trees Kernel-Sdf-Renaming[9.1.3]
Literals-Sdf-Normalization[7.4.3]
exports
context-free syntax

"(" ATerm ")" Renamings	→	ATerm
rnargs(ATermList, ATermList)	→	ATermList
mktree(Literal)	→	ATerm
chartrees(Symbols)	→	ATermList

equations

Renaming an application. If the production is defined in the renaming, rename the arguments and then rename the literals in the argument terms according to the new production.

[1] $$\frac{\text{aterm}((\text{production}(\mathit{Prod}))\ \rho) = \mathit{Prod}',\ \ \mathit{Prod}' \neq \mathit{Prod},\\ (\mathit{Args})\ \rho = \mathit{Args}',\ \text{rnargs}(\text{args}(\mathit{Prod}'),\ \mathit{Args}') = \mathit{Args}''}{(\text{appl}(\mathit{Prod},\ \mathit{Args}))\ \rho\ =\ \text{appl}(\mathit{Prod}',\ \mathit{Args}'')}$$

If the production is not defined in the renaming, then only rename the arguments.

[2] $$(\text{appl}(\mathit{Prod},\ \mathit{Args}))\ \rho = \text{appl}(\mathit{Prod},\ (\mathit{Args})\ \rho)\quad \textbf{otherwise}$$

Renaming is a homomorphism over the other tree constructors.

[3] $$(n)\ \rho = n$$
[4] $$(\text{amb}(\mathit{Args}))\ \rho = \text{amb}((\mathit{Args})\ \rho)$$
[5] $$(\mathit{Tl})\ \rho = [\ ]\quad \textbf{when}\ \ \mathit{Tl} = [\ ]$$
[6] $$([T])\ \rho = [(T)\ \rho]$$
[7] $$([T,\ \mathit{Ts}])\ \rho = [(T)\ \rho,\ \mathit{Ts}']\quad \textbf{when}\ \ [\mathit{Ts}'] = ([\mathit{Ts}])\ \rho$$

Renaming the arguments.

[8] $$\text{rnargs}([\ ],\ [\ ]) = [\ ]$$

Insert literals of the new pattern.

[9] $$\frac{\text{symbol}(\text{first}([\mathit{Ts}_1])) = L}{\text{rnargs}([\mathit{Ts}_1],\ \mathit{Tl}_2)\ =\ \text{mktree}(L) : \text{rnargs}(\text{rest}([\mathit{Ts}_1]),\ \mathit{Tl}_2)}$$

Skip literals of the old tree.

[10] $$\frac{\text{symbol}(\text{type}(\text{first}([\mathit{Ts}_2]))) = L}{\text{rnargs}(\mathit{Tl}_1,\ [\mathit{Ts}_2])\ =\ \text{rnargs}(\mathit{Tl}_1,\ \text{rest}([\mathit{Ts}_2]))}$$

Copy layout from old tree to new tree if layout is requested in new pattern.

$$[11]\quad \frac{\begin{array}{c}\text{symbol}(\text{first}([\,Ts_1])) = \langle\texttt{LAYOUT?-CF}\rangle,\\ \text{symbol}(\text{type}(\text{first}([\,Ts_2]))) = \langle\texttt{LAYOUT?-CF}\rangle\end{array}}{\text{rnargs}([\,Ts_1],[\,Ts_2]) \;=\; \text{first}([\,Ts_2]) : \text{rnargs}(\text{rest}([\,Ts_1]), \text{rest}([\,Ts_2]))}$$

Insert empty layout in the new tree.

$$[12]\quad \frac{\begin{array}{c}\text{symbol}(\text{first}([\,Ts_1])) = \langle\texttt{LAYOUT?-CF}\rangle,\\ \text{symbol}(\text{type}(\text{first}([\,Tl_2]))) \neq \langle\texttt{LAYOUT?-CF}\rangle\end{array}}{\begin{array}{rcl}\text{rnargs}([\,Ts_1],\ Tl_2) &=& \text{appl}(\text{aterm}(\ \to \langle\texttt{LAYOUT?-CF}\rangle\ ), [\,])\\ && : \text{rnargs}(\text{rest}([\,Ts_1]),\ Tl_2)\end{array}}$$

Skip layout of the old tree.

$$[13]\quad \frac{\begin{array}{c}\text{symbol}(\text{first}(\,Tl_1)) \neq \langle\texttt{LAYOUT?-CF}\rangle,\\ \text{symbol}(\text{type}(\text{first}([\,Ts_2]))) = \langle\texttt{LAYOUT?-CF}\rangle\end{array}}{\text{rnargs}(\,Tl_1, [\,Ts_2]) \;=\; \text{rnargs}(\,Tl_1, \text{rest}([\,Ts_2]))}$$

In the other cases there is no layout or literal in either list. This means that it concerns an argument tree that should be copied from the old tree to the renamed tree.

$$\begin{array}{lrcl}[14] & \text{rnargs}([\,Ts_1],[\,Ts_2]) &=& \text{first}([\,Ts_2]) : \text{rnargs}(\text{rest}([\,Ts_1]), \text{rest}([\,Ts_2]))\\ &&& \textbf{otherwise}\end{array}$$

The function ‘mktree’ constructs a tree for a literal L, by constructing the production according to the definition in §7.4.3 and by generating the list of character codes.

$$[15]\quad \frac{\text{chars}(L) = \alpha,\ \ \text{aterm}(\alpha \to L\ ) = \mathit{Prod}}{\text{mktree}(L) \;=\; \text{appl}(\mathit{Prod}, \text{chartrees}(\alpha))}$$

From a list of singleton character classes generate a term list of integers representing the character codes.

$$\begin{array}{lrcl}[16] & \text{chartrees}() &=& [\,]\\ [17] & \text{chartrees}([\texttt{c}]\ \alpha) &=& \text{int}(\texttt{c}) : \text{chartrees}(\alpha)\end{array}$$

9.1.5 Discussion

It would be desirable that renaming preserves well-formedness, i.e., if a tree T is well-formed under some grammer $\mathcal{G}$, it should also be well-formed when renamed with some renaming ρ. That is, we want that

$$\mathcal{G} \vdash T \Rightarrow (\mathcal{G})\rho \vdash (T)\rho$$

In fact we would like that renaming preserves the structure defined by a grammar, i.e., renaming the trees generated by a grammar gives the same trees as those generated by the renamed grammar:

$$(\mathcal{T}[\![\mathcal{G}]\!])\rho = \mathcal{T}[\![(\mathcal{G})\rho]\!]$$

Unfortunately, this is not the case for all renamings. If the argument sorts of a production are renamed using a production renaming, but the sorts are not renamed independently, the arguments of an application with that production have the wrong type after renaming. For instance, the renaming

```
[E "+" E -> E => Set "&" Set -> Set]
```

will change the notation of addition on the sort `E` into a binary operator `&` on the sort `Set`. Other constructs for sort `E` will still have type `E` after renaming, including the arguments of the `&` operator, which will hence not be well-formed. It is sufficient to require that in such cases the corresponding symbol renamings are present as well, i.e., the renaming

```
[E => Set E "+" E -> E => Set "&" Set -> Set]
```

does preserve well-formedness.

Also the interaction between regular expressions and renamings spoils the preservation property. For instance, consider the renaming

```
[{Int ","}* => {Int ";"}*]
```

that is intended to rename lists of integers separated by commas into lists separated by semicolons. This will rename all symbols `{Int ","}*`, but it will not rename the concatenation operators for this sort. The renaming

```
[{Int ","}* => {Int ";"}*
   {Int ","}* "," {Int ","}* -> {Int ","}*
=> {Int ";"}* ";" {Int ";"}* -> {Int ";"}*
   {Int ","}+ "," {Int ","}* -> {Int ","}+
=> {Int ";"}+ ";" {Int ";"}* -> {Int ";"}+
   {Int ","}* "," {Int ","}+ -> {Int ","}+
=> {Int ";"}* ";" {Int ";"}+ -> {Int ";"}+
   {Int ","}+ "," {Int ","}+ -> {Int ","}+
=> {Int ";"}+ ";" {Int ";"}+ -> {Int ";"}+
]
```

is a well-formedness preserving renaming that does have the intended effect.

In all these cases correct renamings can be given that will preserve well-formedness and achieve the intended renaming, but these examples show that care has to be taken when writing down renamings. Ideally we would like to restrict the renamings such that the preservation property holds. It might also be possible to complete a renaming to guarantee well-formedness preservation as in the examples above. This is a matter for further study.

9.2 Aliases

The regular expressions introduced in §8.2 provide a way to concisely declare a number of productions without actually having to write them down. A problem with these regular expressions is that they can become rather large. This is a property that might make their use unattractive. Therefore, we introduce symbol aliases. An alias declaration introduces a short name for a complicated regular expression. All occurences of the alias are replaced by their meaning. For example, the declarations

```
aliases
  {Term ","}*            -> Terms
  {Var  ","}*            -> Vars
  Set[(Var "|->" Term)] -> Subst
```

introduce `Terms` and `Vars` as aliases for lists of `Term` and `Var`, respectively, and `Subst` as an alias for sets of pairs of variables and terms. This entails that all operations generated for list constructs also apply to `Terms` and `Vars` and all operations generated for sets apply to `Subst`.

Aliases are defined using the renamings of the previous section. An alias $\mathcal{A} \to \mathcal{B}$ induces a grammar renaming $[\mathcal{B} \Rightarrow \mathcal{A}]$, which is applied to the entire grammar. Why then introduce this extra feature if we already have renamings? Renamings apply to a fixed grammar. Only the grammar to which the renaming is applied, including all imported grammars, is affected. An alias is a renaming of a symbol that also affects all modules that import the alias.

9.2.1 Syntax

An *alias* grammar consists of a list of aliases of the form $\mathcal{A} \to \mathcal{B}$ that define the symbol $\mathcal{B}$ to be an alias of symbol $\mathcal{A}$.

module Alias-Sdf-Syntax
imports Kernel-Sdf-Syntax[7.3.1]
exports
  sorts Alias Aliases
  context-free syntax
    "aliases" Aliases → Grammar
    Symbol "→" Symbol → Alias
    Alias∗ → Aliases
  variables
    "*al*"[*0-9′*]∗ → Alias
    "*al∗*"[*0-9′*]∗ → Alias∗
    "*al+*"[*0-9′*]∗ → Alias+

9.2.2 Projection

Concatenation of alias lists. Projection of the aliases and non-alias parts of a grammar.

module Alias-Sdf-Projection
imports Alias-Sdf-Syntax[9.2.1]
exports
  context-free syntax
    Aliases "++" Aliases → Aliases   {**right**}
    "Al"(Grammar)   → Aliases
    "$\overline{\text{Al}}$"(Grammar)   → Grammar
equations
The function 'Al' gives all alias declarations of a grammar, '$\overline{\text{Al}}$' the grammar without alias declarations.

$$
\begin{array}{lrl}
[1] & al_1^* \mathbin{+\!\!+} al_2^* & = al_1^* \ al_2^* \\
[2] & \text{Al}(\text{aliases } al^*) & = al^* \\
[3] & \text{Al}(\mathcal{G}_1 \ \mathcal{G}_2) & = \text{Al}(\mathcal{G}_1) \mathbin{+\!\!+} \text{Al}(\mathcal{G}_2) \\
[4] & \text{Al}(\mathcal{G}) & = \quad \textbf{otherwise} \\
[5] & \overline{\text{Al}}(\text{aliases } al^*) & = \emptyset \\
[6] & \overline{\text{Al}}(\mathcal{G}_1 \ \mathcal{G}_2) & = \overline{\text{Al}}(\mathcal{G}_1) \ \overline{\text{Al}}(\mathcal{G}_2) \\
[7] & \overline{\text{Al}}(\mathcal{G}) & = \mathcal{G} \quad \textbf{otherwise}
\end{array}
$$

9.2.3 Normalization

Aliases are defined by renaming all alias symbols to their defined meaning. The function a⟦_⟧ produces a renaming from the alias declarations in the grammar and applies it to the non-alias parts of the grammar. The alias declarations are then attached to the renamed grammar. This is done in order to keep the following modular property:

$$\text{a}[\![\mathcal{G}_1 \ \mathcal{G}_2]\!] = \text{a}[\![\text{a}[\![\mathcal{G}_1]\!] \ \text{a}[\![\mathcal{G}_2]\!]]\!]$$

This entails that aliases can be replaced before flattening a module, after which the aliases are still part of the grammar and keep their forward renaming property.

module Alias-Sdf-Normalization
imports Alias-Sdf-Projection[9.2.2] Kernel-Sdf-Normalization[7.3.3]
    Kernel-Sdf-Renaming[9.1.3]
exports
  context-free syntax
    "a⟦" Grammar "⟧" → Grammar
    rn(Aliases)   → Renamings
    symbols(Aliases)   → SymbolSet
equations
Replace all alias symbols by their definition by applying a renaming derived from the alias declarations to the non-alias parts of the grammar.

$$[1] \qquad \text{a}[\![\mathcal{G}]\!] = \text{aliases } al^+ \ (\overline{\text{Al}}(\mathcal{G})) \ \text{rn}(al^+) \quad \textbf{when} \ \ \text{Al}(\mathcal{G}) = al^+$$

[2] $\mathrm{a}[\![\mathcal{G}]\!] = \mathcal{G}$ **otherwise**

Build a renaming from a list of aliases. The target $\mathcal{B}$ of the alias declaration $\mathcal{A} \to \mathcal{B}$ is renamed to the source $\mathcal{A}$.

[3] $\mathrm{rn}() = []$
[4] $\mathrm{rn}(\mathcal{A} \to \mathcal{B}\ al^*) = [\mathcal{B} \Rightarrow \mathcal{A}] +\!\!+ \mathrm{rn}(al^*)$

The symbols occurring in an alias declaration.

[5] $\mathrm{symbols}(\mathrm{aliases}\ al^*) = \mathrm{symbols}(al^*)$
[6] $\mathrm{symbols}(al^*) = \{\}$ **when** $al^* =$
[7] $\mathrm{symbols}(\mathcal{A} \to \mathcal{B}\ al^*) = \{\mathcal{A}\ \mathcal{B}\} \cup \mathrm{symbols}(al^*)$

Merging and ordering of grammars.

[8] $\mathrm{aliases} = \emptyset$
[9] $\mathrm{aliases}\ al_1^* \diamond \mathrm{aliases}\ al_2^* = \langle \mathrm{aliases}\ al_1^*\ al_2^*, \emptyset \rangle$
[10] $\mathrm{syntax}\ p^* \diamond \mathrm{aliases}\ al^* = \langle \mathrm{aliases}\ al^*, \mathrm{syntax}\ p^* \rangle$

Aliases themselves can also be subject to renamings.

module Alias-Sdf-Renaming
imports Kernel-Sdf-Renaming$^{9.1.3}$ Alias-Sdf-Projection$^{9.2.2}$
exports
  context-free syntax
    "(" Aliases ")a" Renamings → Aliases
equations
Renaming of aliases.

[1] $(\mathrm{aliases}\ al^*)\ \rho = \mathrm{aliases}\ (al^*\ )\mathrm{a}\ \rho$
[2] $(al^*\ )\mathrm{a}\ \rho =$ **when** $al^* =$
[3] $(\mathcal{A} \to \mathcal{B}\ )\mathrm{a}\ \rho = (\mathcal{A})\ \rho \to (\mathcal{B})\ \rho$
[4] $(al_1^+\ al_2^+\ )\mathrm{a}\ \rho = (al_1^+\ )\mathrm{a}\ \rho +\!\!+ (al_2^+\ )\mathrm{a}\ \rho$

9.3 Modules

In this section we introduce a module framework for grammars to support management and reuse of parts of the grammar of a language. A modular definition consists of a list of named modules. Modules can be reused in other modules by means of imports. The body of a module is a list of exported and hidden grammars. Export and hiding provide a means to control what is visible from a module and what is local to that module. Hidden syntax is useful when the syntax definition formalism is coupled to a semantics formalism for the specification of the semantics of languages. Hidden syntax then plays the role of auxiliary functions. Since imports are abbreviations for grammars, an import can be hidden or exported. Modules can be parameterized by a list of symbols. An import can instantiate these parameters, although this is not required. Parameterization is an abbreviation for a renaming. When a module $M[\alpha]$ is imported as $M[\beta]$, the formal parameters $[\alpha]$ are renamed into the actual parameters $[\beta]$. An import can also be subject to a renaming of symbols and productions.

Example 9.3.1 (Aliases and Renaming) The following module defines the syntax of tables. A table is defined as an alias for a set of mappings from keys to values. The value assigned to a key can be looked up in a table using the access function `lookup`.

```
module Tables
exports
  sorts Key Value Table
  aliases
    Set[(Key "|->" Value)] -> Table
  context-free syntax
    lookup(Table, Key) -> Value
```

Below we transform tables into mappings from variables to terms, thus obtaining a representation for substitutions. This is achieved by renaming the sorts in module `Tables` such that variables become the keys and terms the values in tables.

```
module Substitutions
imports Terms
        Tables[Key => Var   Value => Term   Table => Subst
               lookup(Table, Key) -> Value
                   => Subst "[" Var "]" -> Term ]
exports
  context-free syntax
    Subst "(" Term ")" -> Term
```

The additional function applies a substitution to all variables in a term. □

Example 9.3.2 (Map) Using renaming a kind of polymorphic higher-order functions can be expressed. The following module defines a function that maps a function over the elements of a list. The function is defined for a given `A` and `B` that can be instantiated as needed.

```
module Map[A B]
exports
  sorts A B
  context-free syntax
    (A => B) "*" "(" A* ")" -> B*
```

The disadvantage of this kind of polymorphism is that for each instance of a polymorphic function, an explicit module import has to be done. □

Example 9.3.3 (Parameterized Modules) The following module defines the syntax of a list of conditional equations preceded by the keyword 'equations'. This is the syntax of the equations part of an ASF+SDF module, which is parameterized by the syntax of some language. For each sort, productions defining the syntax of equations over that sort are defined. Note the use of the constrained iteration operator to define the bar (`Implies`) between conditions and conclusion as at least 3 equal signs.

```
module Equations
exports
  sorts Tag TagId CondEquation Equation Implies
        Condition Equations
  lexical syntax
    {[\=]}3+       -> Implies
    [a-z0-9A-Z\-]+ -> TagId
  aliases
    {Condition ","}+ -> Conditions
  context-free syntax
    "equations" CondEquation*          -> Equations
    Tag (Conditions Implies)? Equation -> CondEquation
    Tag Equation "when" Conditions     -> CondEquation
    "[" TagId? "]"                     -> Tag
```

Next we define generic syntax for sorts `Equation` and `Condition` as follows:

```
module X-Equations[X]
exports
  sorts X Equation Condition
  context-free syntax
    X "="  X -> Equation
    X "="  X -> Condition
    X "!=" X -> Condition
```

To define the syntax of the equations part of a module M, the ASF+SDF Meta-Environment generates a module M-`Equations` that defines the syntax of these equations (Klint, 1993). This module imports the language independent syntax of equations and defines equations for the sorts declared in the module. With the parameterized module `X-Equations` we can express this by a module that contains an import for each declared sort. For instance, for `Boolean-Equations` we get the following module:

```
module Booleans-Equations
imports Booleans
        Equations
        X-Equations[Bool]
```

Observe that the sorts `Condition` and `Equation` are declared in two different modules. This is not problematic when these modules meet, because duplicate definitions are merged. □

9.3.1 Syntax

A modular syntax definition consists of a series of named module declarations. A module declaration consists of a list of sections, which are either exports or hiddens. A module name consists of a module identifier and an optional list of parameters. Module identifiers can contain slashes to enable the use of directory names in module names, e.g., `sdf/kernel/Syntax`. A module can import any number of other modules. An import consists of a module name with optionally a renaming applied to it. An import of a module M denotes the grammar declared in module M. An import can be contained in one of the exports or hiddens sections. In the latter case all syntax imported through that module is hidden and thus not exported from the module. Imports can also occur at the start of a module, outside any exports or hiddens section. In this case the imports are exported.

module Modular-Sdf-Syntax
imports Kernel-Sdf-Syntax[7.3.1] Renaming-Sdf-Syntax[9.1.1]
exports
  sorts ModuleId ModuleName Import Imports Section Sections Module Definition ImpSection
  lexical syntax

`[A-Za-z0-9_\-]+`	→	ModuleWord
"/"ModuleWord	→	ModuleDir
ModuleWord ModuleDir+	→	ModuleId
ModuleDir+	→	ModuleId

  context-free syntax

Module∗	→	Definition
"module" ModuleName ImpSection∗ Sections	→	Module
"exports" Grammar	→	Section
"hiddens" Grammar	→	Section
Section∗	→	Sections
ModuleId	→	ModuleName
ModuleId "[" Symbols "]"	→	ModuleName
id(ModuleName)	→	Attribute

"imports" Imports	→ ImpSection	
ImpSection	→ Grammar	
Import∗	→ Imports	
ModuleName	→ Import	
ModuleName Renamings	→ Import	
"(" Import ")"	→ Import	{**bracket**}

variables

"*Mid*"[*0-9′*]∗	→ ModuleId
"*M*"[*0-9′*]∗	→ ModuleName
"*s*"[*0-9′*]∗	→ Section
"*s*" "∗"[*0-9′*]∗	→ Section∗
"*s*" "+"[*0-9′*]∗	→ Section+
"*m*"[*0-9′*]∗	→ Module
"*m*∗"[*0-9′*]∗	→ Module∗
"*m*+"[*0-9′*]∗	→ Module+
"*d*"[*0-9′*]∗	→ Definition
"i"[*0-9′*]∗	→ Import
"i" "∗"[*0-9′*]∗	→ Import∗
"i" "+"[*0-9′*]∗	→ Import+
"*is*"[*0-9′*]∗	→ ImpSection
"*is*∗"[*0-9′*]∗	→ ImpSection∗

9.3.2 Projection

Projection functions: $\pi_M(d)$ yields the body of the module named M. 'Exp' yields the exported part of a module and 'Hid' yields the hidden part of a module.

module Modular-Sdf-Projection
imports Modular-Sdf-Syntax[9.3.1] Booleans Kernel-Sdf-Projection[7.3.2] Modular-Sdf-Renaming[9.3.4]
exports
context-free syntax

Import "∈" Imports	→ Bool	
Sections "++" Sections	→ Sections	{**assoc**}
Imports "++" Imports	→ Imports	{**assoc**}
"π" "_" ModuleName "(" Definition ")"	→ Sections	
"Exp"(Sections)	→ Grammar	
"Hid"(Sections)	→ Grammar	

equations

Membership of a list of imports.

[1] $$\mathsf{i} \in \mathsf{i}_1^* \ \mathsf{i} \ \mathsf{i}_2^* = \top$$

[2] $$\mathsf{i} \in \mathsf{i}^* = \bot \quad \textbf{otherwise}$$

Concatenation of section and imports lists.

[3] $$s_1^* + s_2^* = s_1^* \ s_2^*$$

[4] $$\mathsf{i}_1^* +\!\!+ \mathsf{i}_2^* = \mathsf{i}_1^* \; \mathsf{i}_2^*$$

Lookup of a module by its name in a list of modules. If a module name matches the module name searched for, its list of sections is yielded. If a parameterized module is imported without specifying any actual parameters, the parameters are left uninstantiated. If a list of actual paramters is given, these are used to rename the formal parameters into the actual parameters. The function $(\_ \Rightarrow \_)$ constructs a renaming from the formal parameters to the actual parameters of a parameterized module. If no modules are found the empty list of sections is yielded.

[5] $$\pi_M(\text{module } M \;\; s^* \; m^*) = s^* +\!\!+ \pi_M(m^*)$$
[6] $$\pi_{Mid}(\text{module } Mid[\alpha] \;\; s^* \; m^*) = s^* +\!\!+ \pi_{Mid}(m^*)$$
[7] $$\pi_{Mid[\beta]}(\text{module } Mid[\alpha] \;\; s^* \; m^*) = (s^*)\; (\alpha \Rightarrow \beta) +\!\!+ \pi_{Mid[\beta]}(m^*)$$
[8] $$\pi_M(\text{module } M' \;\; s^* \; m^*) = \pi_M(m^*) \quad \textbf{otherwise}$$
[9] $$\pi_M() =$$

Exported grammars from a list of sections.

[10] $$\text{Exp}() = \emptyset$$
[11] $$\text{Exp}(s_1^+ \; s_2^+) = \text{Exp}(s_1^+)\; \text{Exp}(s_2^+)$$
[12] $$\text{Exp}(\text{exports } \mathcal{G}) = \mathcal{G}$$
[13] $$\text{Exp}(\text{hiddens } \mathcal{G}) = \emptyset$$

Hidden grammars from a list of sections.

[14] $$\text{Hid}() = \emptyset$$
[15] $$\text{Hid}(s_1^+ \; s_2^+) = \text{Hid}(s_1^+)\; \text{Hid}(s_2^+)$$
[16] $$\text{Hid}(\text{exports } \mathcal{G}) = \emptyset$$
[17] $$\text{Hid}(\text{hiddens } \mathcal{G}) = \mathcal{G}$$

9.3.3 Normalization

We define the semantics of the modular constructs introduced above by means of a normalization function that yields the flattening of a module in a modular syntax definition by replacing each import by the body of the module it refers to.

Hidden productions are renamed by attaching the name of the hiding module. Since all productions occurring in a hiddens do not occur in another hiddens section (they should have been exported) it can never occur that two such renamed productions are imported into the same module. A consequence of production merging in this case is that an exported function becomes hidden if it is also in the hiddens part of the module.

We define the function $\text{m}[\![d]\!](M)$ that yields the grammar corresponding to module M in the definition d.

module Modular-Sdf-Normalization
imports Modular-Sdf-Projection$^{9.3.2}$ Modular-Sdf-Renaming$^{9.3.4}$
    Kernel-Sdf-Normalization$^{7.3.3}$ Grammar-Projection
exports
  context-free syntax
    "m⟦" Definition "⟧" "(" ModuleName ")" → Grammar
    hide(ModuleName, Grammar) → Grammar
    hide(ModuleName, Productions) → Productions
hiddens
  sorts IG
  context-free syntax
    "<" Imports "," Grammar ">" → IG
    imp "⟦" Definition "⟧" "(" Imports "," Import ")" → IG
    ims "⟦" Definition "⟧" "(" Imports "," Imports ")" → IG
    gra "⟦" Definition "⟧" "(" Imports "," Grammar ")" → IG
equations

Normalization of order of grammars.

[1] $\text{imports} = \emptyset$

[2] $\text{imports } \mathsf{i}_1^* \diamond \text{imports } \mathsf{i}_2^* = \langle \text{imports } \mathsf{i}_1^*\ \mathsf{i}_2^*,\ \emptyset \rangle$

[3] $\mathcal{G} \diamond \text{imports } \mathsf{i}^* = \langle \text{imports } \mathsf{i}^*,\ \mathcal{G} \rangle \quad \textbf{otherwise}$

Normalization of module sections. Exports and hiddens sections can be merged.

[4] $\text{module } M\ is^*\ is\ s^* = \text{module } M\ is^* \text{ exports } is\ s^*$

[5] $s_1^* \text{ exports } \mathcal{G}_1 \text{ exports } \mathcal{G}_2\ s_2^* = s_1^* \text{ exports } \mathcal{G}_1\ \mathcal{G}_2\ s_2^*$

[6] $s_1^* \text{ hiddens } \mathcal{G}_1 \text{ hiddens } \mathcal{G}_2\ s_2^* = s_1^* \text{ hiddens } \mathcal{G}_1\ \mathcal{G}_2\ s_2^*$

[7] $s_1^* \text{ hiddens } \mathcal{G}_1 \text{ exports } \mathcal{G}_2\ s_2^* = s_1^* \text{ exports } \mathcal{G}_2 \text{ hiddens } \mathcal{G}_1\ s_2^*$

The semantics of a module named M in a definition d is expressed by $\text{m}[\![d]\!](M)$ and is the composition of the exported and hidden grammars of module M with all imports replaced by the exported grammars of the modules they refer to.

[8] $$\frac{\pi_M(d) = s^*,\ \ \text{gra}[\![d]\!](,\ \text{Hid}(s^*)) = \langle \mathsf{i}_1^*,\ \mathcal{G}_1 \rangle,\ \ \text{gra}[\![d]\!](\mathsf{i}_1^*,\ \text{Exp}(s^*)) = \langle \mathsf{i}_2^*,\ \mathcal{G}_2 \rangle}{\text{m}[\![d]\!](M)\ =\ \mathcal{G}_2\ \text{hide}(M,\ \mathcal{G}_1)}$$

The function 'hide' marks all productions in the hiddens part of a module with the module name by attaching the attribute $\text{id}(M)$ to it.

[9] $\text{hide}(M,\ \emptyset) = \emptyset$

[10] $\text{hide}(M,\ \mathcal{G}_1\ \mathcal{G}_2) = \text{hide}(M,\ \mathcal{G}_1)\ \text{hide}(M,\ \mathcal{G}_2)$

[11] $\text{hide}(M,\ \text{syntax } p^*) = \text{syntax hide}(M,\ p^*)$

[12] $\text{hide}(M,\ \mathcal{G}) = \mathcal{G} \quad \textbf{otherwise}$

[13] $\text{hide}(M,\ \alpha \to \mathcal{A}\ \$) = \alpha \to \mathcal{A}\ \$ +\!\!+ \{\text{id}(M)\}$

[14] $\text{hide}(M,\ ) =$

[15] $\text{hide}(M,\ p_1^+\ p_2^+) = \text{hide}(M,\ p_1^+) +\!\!+ \text{hide}(M,\ p_2^+)$

The function 'gra' expands all the imports in a grammar. It returns a structure $\langle \mathsf{i}^*, \mathcal{G} \rangle$, which denotes a flattened grammar with the list of imports i^* that were expanded to flatten the grammar. This list is passed on to the rest of the flattening process in order to prevent multiple imports of the same module. This is important in particular in the presence of cyclic imports.

[16] $$\frac{\text{gra}[\![d]\!](\mathsf{i}_1^*, \mathcal{G}_1) = \langle \mathsf{i}_2^*, \mathcal{G}_1' \rangle,\ \ \text{gra}[\![d]\!](\mathsf{i}_2^*, \mathcal{G}_2) = \langle \mathsf{i}_3^*, \mathcal{G}_2' \rangle}{\text{gra}[\![d]\!](\mathsf{i}_1^*, \mathcal{G}_1\ \mathcal{G}_2) \ = \ \langle \mathsf{i}_3^*, \mathcal{G}_1'\ \mathcal{G}_2' \rangle}$$

[17] $$\text{gra}[\![d]\!](\mathsf{i}_1^*, \text{imports } \mathsf{i}_2^*) \ = \ \text{ims}[\![d]\!](\mathsf{i}_1^*, \mathsf{i}_2^*)$$

[18] $$\text{gra}[\![d]\!](\mathsf{i}^*, \mathcal{G}) \ = \ \langle \mathsf{i}^*, \mathcal{G} \rangle$$
otherwise

The function 'ims' yields the flattened grammars for a list of imports.

[19] $$\text{ims}[\![d]\!](\mathsf{i}^*,) \ = \ \langle \mathsf{i}^*, \emptyset \rangle$$

[20] $$\frac{\text{imp}[\![d]\!](\mathsf{i}_1^*, \mathsf{i}) = \langle \mathsf{i}_3^*, \mathcal{G}_1 \rangle,\ \ \text{ims}[\![d]\!](\mathsf{i}_3^*, \mathsf{i}_2^*) = \langle \mathsf{i}_4^*, \mathcal{G}_2 \rangle}{\text{ims}[\![d]\!](\mathsf{i}_1^*, \mathsf{i}\ \mathsf{i}_2^*) \ = \ \langle \mathsf{i}_4^*, \mathcal{G}_1\ \mathcal{G}_2 \rangle}$$

The function 'imp' yields the flattened grammar associated with the exported grammar of an import. The first list of imports denotes the imports that are already expanded. If a module was already imported it is not imported again. This is a protection against cyclic imports.

[21] $$\frac{\mathcal{G} = \text{if } M \in \mathsf{i}^* \text{ then } \emptyset \text{ else } \text{Exp}(\pi_M(d)) \text{ fi}}{\text{imp}[\![d]\!](\mathsf{i}^*, M) \ = \ \text{gra}[\![d]\!](\mathsf{i}^*\ M, \mathcal{G})}$$

[22] $$\frac{\mathcal{G} = \text{if } M\,\rho \in \mathsf{i}^* \text{ then } \emptyset \text{ else } (\text{Exp}(\pi_M(d)))\ \rho \text{ fi}}{\text{imp}[\![d]\!](\mathsf{i}^*, M\,\rho) \ = \ \text{gra}[\![d]\!](\mathsf{i}^*\ M\,\rho, \mathcal{G})}$$

As we will see in the next section, the renaming ρ that is applied to the exported part of the imported module M in the last equation above is also applied to the imports of that module and hence is applied recursively to all modules imported via M.

9.3.4 Renaming

We extend the definition of renaming to renaming of module sections and imports. This includes the renaming of imports, and hence the renaming of renamings applied to imported modules.

module Modular-Sdf-Renaming
imports Renaming-Sdf-Renaming$^{9.1.3}$ Modular-Sdf-Syntax$^{9.3.1}$
    Modular-Sdf-Projection$^{9.3.2}$
exports
  context-free syntax
    "(" Sections ")" Renamings → Sections
    "(" Imports ")" Renamings → Imports

equations

Renaming sections.

[1] $(s^*)\ \rho = \quad$ **when** $s^* =$

[2] $(\text{exports } \mathcal{G})\ \rho = \text{exports } (\mathcal{G})\ \rho$

[3] $(\text{hiddens } \mathcal{G})\ \rho = \text{hiddens } (\mathcal{G})\ \rho$

[4] $(s_1^+\ s_2^+)\ \rho = (s_1^+)\ \rho +\!\!+ (s_2^+)\ \rho$

Renaming a list of imports implies applying the renaming to all imported modules, i.e., attaching the renaming to each module name in the list of imports.

[5] $(\text{imports } \mathsf{i}^*)\ \rho = \text{imports } (\mathsf{i}^*)\ \rho$

[6] $(M)\ \rho = M\ \rho$

[7] $(\mathsf{i}^*)\ \rho = \quad$ **when** $\mathsf{i}^* =$

[8] $(\mathsf{i}_1^+\ \mathsf{i}_2^+)\ \rho = \mathsf{i}_3^+\ \mathsf{i}_4^+$ **when** $\mathsf{i}_3^+ = (\mathsf{i}_1^+)\ \rho,\ \mathsf{i}_4^+ = (\mathsf{i}_2^+)\ \rho$

If the imported module has already a renaming attached to it, the new renaming is applied to the first, yielding the composition of the two renamings.

[9] $(M\ \rho_1)\ \rho_2 = M\ (\rho_1)\ \rho_2$

9.3.5 Discussion

The modularization presented here is an extension with symbol parameters, import renamings and hidden imports of the modularization of ASF+SDF as implemented in the ASF+SDF Meta-Environment (Klint, 1993). The definition here is a pure 'textual' inclusion semantics of modularization. Hendriks (1991) describes both a textual normalization semantics and an incremental semantics for modular constructs without renamings and hidden imports. The incremental implementation of modularization in the Meta-Environment becomes complicated in the presence of renamings, since items are created on the fly and can no longer be associated with a module. We have not addressed the issue of incremental parser generation and modular parser generation in a setting with renamings.

We deviate from the original design of ASF in that we do not incorporate the 'origin rule' that forbids identification of names that originate from different modules (Bergstra *et al.*, 1989b). This style forbids to have two modules with partly overlapping signatures, e.g., both introducing the same sort or function, that are imported in the same module, even if the overlap is intentional. The definition here is completeley liberal in this respect. Productions that are imported via different routes are identified if they are the same.

In §9.1 we saw that renamings are not guaranteed to preserve well-formedness of trees. A further study of modular properties of grammars in the line of module algebra (Bergstra *et al.*, 1990) should give more insight into properties of good modularization. Some topics for study are: properties of trees and languages under renaming, ambiguity caused by union, interaction of regular expressions and renamings, modular properties of reject productions.

10

The Syntax Definition Formalism SDF2

This chapter presents the assembly of the syntax definition formalism SDF2 from the features designed in previous chapters. This is mainly a matter of defining collecting modules that import the modules defined earlier. However, some features interfere. In some cases the normalization functions have to be extended to cover constructs introduced for other features. In other cases features have to be extended such that the orthogonality of another feature is maintained. The chapter concludes with a comparison of SDF2 to SDF and a discussion of anomalies and possible improvements to the formalism.

10.1 SDF2

Now we put the pieces together and define the syntax definition formalism SDF2, which is a generalization of SDF (Heering *et al.*, 1989). It covers all features available in SDF and adds several new ones. Furthermore, up to some small adaptations, SDF is textually (although not structurally) a subset of SDF2. This means that existing SDF definitions can be used almost literally as SDF2 definitions. The differences can be translated automatically by means of a migration tool.

The combination of features described earlier is achieved basically by combining them by means of imports into collecting modules. For each aspect of the definition, such as syntax, projection and normalization, a collecting module is defined. Here we show the collecting modules for the syntax and normalization of SDF2. The other modules can be found in Appendix A.4. Although we have tried to define features orthogonally, some interference between them is unavoidable. For instance, when we extend the syntax of symbols, normalization functions that deal with symbols are affected and have to be extended accordingly.

10.1.1 Syntax

The syntax of SDF2 is simply the collection of the syntax of all features introduced sofar. The syntax is extended with *lexical* and *context-free* priorities and restrictions, which arise as a result of the combination of Basic-Sdf with Priority-Sdf and Restriction-Sdf. The constructor 'definition' collects a list of modules into a single SDF definition.

The symbols `⟨Start⟩` and `⟨START⟩` serve to define grammars with a single start symbol. In the normalization below productions will be added such that `⟨START⟩` is the union of all sorts of the grammar. The symbol `⟨Start⟩` is used to describe files that consist of a string over the language of `⟨START⟩` followed by the end of file character.

In the Label extension a symbol can be labeled with a literal using the syntax $L : \mathcal{A}$. This extension is not further defined here. The priorities section is extended to deal with this extra symbol constructor.

module Sdf2-Syntax
imports Kernel-Sdf-Syntax$^{7.3.1}$ Basic-Sdf-Syntax$^{8.3.1}$ Modular-Sdf-Syntax$^{9.3.1}$
Regular-Sdf-Syntax$^{8.2.1}$ Priority-Sdf-Syntax$^{8.1.1}$ CC-Sdf-Syntax$^{7.4.2}$
Sorts-Sdf-Syntax$^{7.4.1}$ Literals-Sdf-Syntax$^{7.4.3}$ Label-Sdf-Syntax
Restrictions-Sdf-Syntax$^{8.4.1}$ Alias-Sdf-Syntax$^{9.2.1}$
exports
sorts SDF
context-free syntax

"`⟨START⟩`"	→ Symbol
"`⟨Start⟩`"	→ Symbol
"lexical" "priorities" Priorities	→ Grammar
"context-free" "priorities" Priorities	→ Grammar
"lexical" "restrictions" Restrictions	→ Grammar
"context-free" "restrictions" Restrictions	→ Grammar
"definition" Definition	→ SDF

priorities
Symbol "|"Symbol → Symbol > Literal ":"Symbol → Symbol

10.1.2 Normalization

We define the normalization function that normalizes a syntax definition by applying the normalization functions of the individual features. Here we have to deal with interaction between the normalization functions for the separate features and the constructs added to the formalism in other features.

module Sdf2-Normalization
imports Sdf2-Syntax$^{10.1.1}$ Sdf2-Projection$^{A.4}$ Sdf2-Renaming$^{A.4}$
Basic-Sdf-Normalization$^{8.3.2}$ Modular-Sdf-Normalization$^{9.3.3}$
Priority-Sdf-Normalization$^{8.1.3}$ Regular-Sdf-Normalization$^{8.2.2}$
Literals-Sdf-Normalization$^{7.4.3}$ CC-Sdf-Normalization$^{7.4.2}$
Sorts-Sdf-Normalization$^{7.4.1}$ Sorts-Sdf-Projection$^{7.4.1}$
Restrictions-Sdf-Normalization$^{8.4.3}$ Alias-Sdf-Normalization$^{9.2.3}$
exports
context-free syntax
normalize "⟦" SDF "⟧" "(" ModuleName "," Symbol ")" → Grammar

topsorts "⟦" Grammar "⟧" "(" Symbol ")" → Grammar
topsorts(Symbol, Symbols) → Productions

equations

The normalization of an SDF2 definition is defined by the following equation. The function 'normalize' is parameterized with a module name denoting the top module to be normalized and a sort denoting the topsort of the definition. The definition is normalized by first expanding module M by means of function 'm'. Then the normalization functions 'b' (Basic), 'a' (Alias), 'r' (Regular), 'p' (Priorities), 'l' (Literals) and 'k' (Kernel) are applied to the resulting grammar. The function 'topsorts', defined below, is used to add special productions for the top sorts of the definition and to remove productions not reachable from those top sorts.

$$[1] \qquad \frac{\mathrm{k}[\![\mathrm{l}[\![\mathrm{p}[\![\mathrm{r}[\![\mathrm{topsorts}[\![\mathrm{a}[\![\mathrm{b}[\![\mathrm{m}[\![d]\!](M)]\!]]\!]]\!](\mathcal{A})]\!]]\!]]\!]]\!] = \mathcal{G}}{\mathrm{normalize}[\![\mathrm{definition}\ d]\!](M, \mathcal{A}) \ = \ \mathrm{reachable}(\{\langle\mathtt{Start}\rangle\}, \mathcal{G})}$$

The function 'topsorts' adds a special production for the symbol $\langle\mathtt{Start}\rangle$, which declares that a text over the grammar is a string of sort $\mathcal{A}$ followed by the character representing the end of file. For each declared sort in the definition a production is added that defines that a text can be a string of that sort which starts and ends with layout.

$$[2] \qquad \frac{\mathcal{G}' = \mathrm{syntax}\ \mathcal{A}\ [\backslash\mathtt{EOF}] \to \langle\mathtt{Start}\rangle \ +\!\!+\ \mathrm{topsorts}(\mathcal{A}, \mathrm{S}(\mathcal{G}))}{\mathrm{topsorts}[\![\mathcal{G}]\!](\mathcal{A}) \ = \ \mathcal{G}'\ \mathcal{G}}$$

$$[3] \qquad \mathrm{topsorts}(\mathcal{A}, \alpha^+\ \beta^+) = \mathrm{topsorts}(\mathcal{A}, \alpha^+) +\!\!+\ \mathrm{topsorts}(\mathcal{A}, \beta^+)$$
$$[4] \qquad \mathrm{topsorts}(\mathcal{A},) =$$
$$[5] \qquad \mathrm{topsorts}(\mathcal{A}, \mathcal{B}) = \langle\mathtt{LAYOUT?\text{-}CF}\rangle\ \langle\mathcal{B}\mathtt{\text{-}CF}\rangle\ \langle\mathtt{LAYOUT?\text{-}CF}\rangle \to \mathcal{A}$$

10.1.3 Interaction

Several of the normalization functions are underdefined, i.e., the full SDF2 formalism contains more constructors than the extension for which they have been defined. Therefore, we must extend these functions accordingly.

exports
context-free syntax
"<" Priorities "`-LEXp`" ">" → Priorities
"<" Priorities "`-CFp`" ">" → Priorities
"<" Restrictions "`-LEX`" ">" → Restrictions
"<" Restrictions "`-CF`" ">" → Restrictions

hiddens
variables
"L" → Literal

equations

The normalization function for regular expressions must be extended to the symbol constructors added in other extensions. The first equations express that sorts, character classes, literals and the symbols LAYOUT, ⟨START⟩ and ⟨Start⟩ do not generate any productions.

[6]	$\mathrm{r}[\![\mathcal{S}]\!] = \emptyset$	[7]	$\mathrm{r}[\![\texttt{LAYOUT}]\!] = \emptyset$
[8]	$\mathrm{r}[\![cc]\!] = \emptyset$	[9]	$\mathrm{r}[\![\langle\texttt{START}\rangle]\!] = \emptyset$
[10]	$\mathrm{r}[\![L]\!] = \emptyset$	[11]	$\mathrm{r}[\![\langle\texttt{Start}\rangle]\!] = \emptyset$

The following equations define that the productions generated for some symbol $\mathcal{A}$ should be transformed into productions for lexical (context-free) productions if a lexical (context-free) version of the symbol occurs. This entails that first the productions for $\mathcal{A}$ are generated by the recursive call and that these are transformed by the ⟨_-LEX⟩ (⟨_-CF⟩) function.

[12] $\mathrm{r}[\![\langle\mathcal{A}\texttt{-LEX}\rangle]\!] = \langle\mathrm{r}[\![\mathcal{A}]\!]\texttt{-LEX}\rangle$

[13] $\mathrm{r}[\![\langle\mathcal{A}\texttt{-CF}\rangle]\!] = \langle\mathrm{r}[\![\mathcal{A}]\!]\texttt{-CF}\rangle$

[14] $\mathrm{r}[\![\langle\mathcal{A}\texttt{-VAR}\rangle]\!] = \mathrm{r}[\![\mathcal{A}]\!]$

This is an example of the context-sensitivity of the generation of productions from symbols. The meaning of $\langle \mathrm{Id} * \texttt{-CF}\rangle$ is different from that of $\langle \mathrm{Id} * \texttt{-LEX}\rangle$.

Basic Literals and character classes do not need the ⟨_-LEX⟩ or ⟨_-CF⟩ constructor, because they are lexical by definition

[15]	$\langle cc\texttt{-LEX}\rangle = cc$	[16]	$\langle cc\texttt{-CF}\rangle = cc$	[17]	$\langle cc\texttt{-VAR}\rangle = cc$
[18]	$\langle L\texttt{-LEX}\rangle = L$	[19]	$\langle L\texttt{-CF}\rangle = L$	[20]	$\langle L\texttt{-VAR}\rangle = L$

Basic + Priorities Equations for the normalization of lexical and context-free priorities that were added at the level of SDF2.

[21]
$$\frac{\mathcal{G}_1 = \text{context-free priorities } pr^*,\ \ \mathcal{G}_2 = \text{context-free syntax } p^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \ = \ \langle\text{context-free syntax } p^*, \text{context-free priorities } pr^*\rangle}$$

[22]
$$\frac{\mathcal{G}_1 = \text{context-free priorities } pr^*,\ \ \mathcal{G}_2 = \text{lexical syntax } p^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \ = \ \langle\text{lexical syntax } p^*, \text{context-free priorities } pr^*\rangle}$$

[23]
$$\frac{\mathcal{G}_1 = \text{context-free priorities } pr_1^*,\ \ \mathcal{G}_2 = \text{context-free priorities } pr_2^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \ = \ \langle\text{context-free priorities } pr_1^*,\, pr_2^*,\, \emptyset\rangle}$$

Context-free priorities are priority declarations for context-free productions and are abbreviations of normal priorities in the same way that context-free syntax is an abbreviation for a certain style of normal syntax. The productions in the priorities sections are thus treated with the same ⟨_-CF⟩ functions as context-free productions.

[24] $\mathrm{b_{aux}}[\![\text{context-free priorities } pr^*]\!] = \langle\text{priorities } pr^*\texttt{-CF}\rangle$

[25] $\langle\text{priorities } pr^*\texttt{-CF}\rangle = \text{priorities } \langle\mathrm{norm}[\![pr^*]\!]\texttt{-CFp}\rangle$

[26] $\langle pr^*\texttt{-CFp}\rangle = \quad \textbf{when} \ \ pr^* =$

[27] $\langle pr_1^+,\, pr_2^+\texttt{-CFp}\rangle = \langle pr_1^+\texttt{-CFp}\rangle +\!\!+ \langle pr_2^+\texttt{-CFp}\rangle$

$$[28] \qquad \langle p_1 > p_2\texttt{-CFp}\rangle = \langle p_1\texttt{-CF}\rangle > \langle p_2\texttt{-CF}\rangle$$
$$[29] \qquad \langle p_1 \ \mathit{as}\ p_2\texttt{-CFp}\rangle = \langle p_1\texttt{-CF}\rangle \ \mathit{as}\ \langle p_2\texttt{-CF}\rangle$$

Similarly for lexical priorities.

$$[30] \qquad \mathrm{b_{aux}}[\![\text{lexical priorities } pr^*]\!] = \langle \text{priorities } pr^*\texttt{-LEX}\rangle$$
$$[31] \qquad \langle \text{priorities } pr^*\texttt{-LEX}\rangle = \text{priorities } \langle \text{norm}[\![pr^*]\!]\texttt{-LEXp}\rangle$$
$$[32] \qquad \langle pr^*\texttt{-LEXp}\rangle = \quad \textbf{when} \ \ pr^* =$$
$$[33] \qquad \langle pr_1^+,\, pr_2^+\texttt{-LEXp}\rangle = \langle pr_1^+\texttt{-LEXp}\rangle +\!\!\!+ \langle pr_2^+\texttt{-LEXp}\rangle$$
$$[34] \qquad \langle p_1 > p_2\texttt{-LEXp}\rangle = \langle p_1\texttt{-LEX}\rangle > \langle p_2\texttt{-LEX}\rangle$$
$$[35] \qquad \langle p_1 \ \mathit{as}\ p_2\texttt{-LEXp}\rangle = \langle p_1\texttt{-LEX}\rangle \ \mathit{as}\ \langle p_2\texttt{-LEX}\rangle$$

Basic + Restrictions

$$[36] \qquad \mathrm{b_{aux}}[\![\text{lexical restrictions } \mathit{restr}^*]\!] = \text{restrictions } \langle \mathit{restr}^*\texttt{-LEX}\rangle$$

$$[37] \qquad \langle \mathit{restr}^*\texttt{-LEX}\rangle = \quad \textbf{when} \ \ \mathit{restr}^* =$$
$$[38] \qquad \langle \mathit{restr}_1^+ \ \mathit{restr}_2^+\texttt{-LEX}\rangle = \langle \mathit{restr}_1^+\texttt{-LEX}\rangle +\!\!\!+ \langle \mathit{restr}_2^+\texttt{-LEX}\rangle$$
$$[39] \qquad \langle \alpha \not- \mathit{cc}\texttt{-LEX}\rangle = \langle \alpha\texttt{-LEXs}\rangle \not- \mathit{cc}$$

$$[40] \qquad \mathrm{b_{aux}}[\![\text{context-free restrictions } \mathit{restr}^*]\!] = \text{restrictions } \langle \mathit{restr}^*\texttt{-CF}\rangle$$

$$[41] \qquad \langle \mathit{restr}^*\texttt{-CF}\rangle = \quad \textbf{when} \ \ \mathit{restr}^* =$$
$$[42] \qquad \langle \mathit{restr}_1^+ \ \mathit{restr}_2^+\texttt{-CF}\rangle = \langle \mathit{restr}_1^+\texttt{-CF}\rangle +\!\!\!+ \langle \mathit{restr}_2^+\texttt{-CF}\rangle$$
$$[43] \qquad \langle \not- \mathit{cc}\texttt{-CF}\rangle = \ \not- \mathit{cc}$$
$$[44] \qquad \langle \mathcal{A}\ \alpha \not- \mathit{cc}\texttt{-CF}\rangle = \langle \mathcal{A}\texttt{-CF}\rangle\ \beta \not- \mathit{cc} \quad \textbf{when} \ \ \langle \alpha \not- \mathit{cc}\texttt{-CF}\rangle = \beta \not- \mathit{cc}$$

Labels

$$[45] \qquad \mathrm{r}[\![L : \mathcal{A}]\!] = \mathrm{r}[\![\mathcal{A}]\!]$$
$$[46] \qquad \langle L : \mathcal{A}\texttt{-LEX}\rangle = L : \langle \mathcal{A}\texttt{-LEX}\rangle$$
$$[47] \qquad \langle L : \mathcal{A}\texttt{-CF}\rangle = L : \langle \mathcal{A}\texttt{-CF}\rangle$$
$$[48] \qquad \langle L : \mathcal{A}\texttt{-VAR}\rangle = L : \langle \mathcal{A}\texttt{-VAR}\rangle$$

Hiding Productions

$$[49] \qquad \text{hide}(M, \text{context-free syntax } p^*) = \text{context-free syntax hide}(M, p^*)$$
$$[50] \qquad \text{hide}(M, \text{lexical syntax } p^*) = \text{lexical syntax hide}(M, p^*)$$
$$[51] \qquad \text{hide}(M, \text{variables } p^*) = \text{variables hide}(M, p^*)$$
$$[52] \qquad \text{hide}(M, \text{lexical variables } p^*) = \text{lexical variables hide}(M, p^*)$$

Aliases

$$[53] \qquad \frac{\mathcal{G}_1 = \text{aliases } \mathit{al}^*,\ \ \mathcal{G}_2 = \text{sorts } \alpha}{\mathcal{G}_1 \diamond \mathcal{G}_2 \ = \ \langle \text{sorts } \alpha, \text{ aliases } \mathit{al}^*\rangle}$$

10.2 Comparison to SDF

SDF2 was developed as a generalization of SDF (Heering *et al.*, 1989). We briefly list the differences between the two formalisms.

10.2.1 Semantics

SDF defines the semantics of a syntax definition by means of mappings to other formalisms. The lexical syntax is mapped to a regular grammar. The context-free syntax is mapped to a context-free grammar. From the entire definition a first-order many-sorted algebraic signature is derived. A parse tree for a string according to the grammar is translated to a term or abstract syntax tree over the signature. In SDF2 parse trees are defined by means of a well-formedness predicate on ATerms based directly on (the normal form of) a syntax definition. The strings of the language defined by a grammar are obtained via the function yield. No external formalism is used to define trees. In this way the notions of grammar and signature that were related via mappings in SDF are completely integrated in SDF2.

10.2.2 Lexical and Context-free Syntax

SDF integrates lexical syntax and context-free syntax in one formalism. However, this integration is only at the level of the formalism; on the level of the implementation these are separated. The lexical syntax is mapped to a regular grammar (hence the specification of the lexical syntax should also be regular). The context-free syntax is translated to a context-free grammar. In SDF2 the integration of lexical and context-free syntax is completed. All other features are orthogonal with respect to lexical and context-free syntax. For instance, character classes and regular expressions can be used in exactly the same way in lexical productions and context-free productions.

10.2.3 Lexical Disambiguation

SDF has several built-in lexical disambiguation rules that are applied to the token stream before tokens are passed to the parser. SDF2 has no built-in lexical disambiguation rules, but provides reject productions to express the prefer literals rule and follow restrictions to express longest match disambiguation.

10.2.4 Character Classes

In SDF the syntax of character classes is defined lexically. In SDF2 character classes are defined by means of context-free constructors. This makes the definition of normalization of character classes much easier. The differences with character classes in SDF are: numeric characters have a decimal interpretation instead of an octal interpretation, there is no syntactic limit to the range of numeric characters, all characters except letters and digits have to be escaped using a slash. In SDF2 character classes have a numeric interpretation, that is, each character class is normalized to an ordered and non-overlapping list of numeric characters and ranges of characters.

10.2.5 Lists

SDF only provides list sorts that can be used in the left-hand sides of productions. Furthermore, lists are not orthogonally defined. In the lexical syntax no iteration with separator is provided. In SDF only sorts can be used on the right-hand side of a production. This means that list sorts cannot be the result of functions. In order to define a function with a list as result, a new sort has to be introduced into which the list sort is injected. Furthermore, to concatenate the lists that result from a function a concatenation function should be defined.

SDF2 provides an expressive set of regular expressions that are treated as first-class citizens. Regular expressions can be used where ever any another symbol can be used. In general, all symbols that can be used in the left-hand side of a production can also be used as output symbols.

10.2.6 Priorities

The priorities declarations of SDF2 are the same as in SDF with the following exceptions: No abbreviations of productions in priorities are supported because of the problematic semantics in a setting with modules. No <-chains are provided. The implementation does not provide the multi-set filter interpretation of priorities.

10.2.7 Reuse

SDF does not provide renamings, module parameterization, hidden imports, and aliases.

10.3 Discussion and Concluding Remarks

We have presented the modular design of a family of syntax definition formalisms. The result is a uniform formalism for syntax definition designed for extensibility. A guiding principle in the design is the orthogonality of the features with respect to one another. As as consequence it is easy to replace a feature by a variant or to add a new feature without affecting the design and implementation of all other features.

10.3.1 Parser Generation

The direct motivation for this work was the specification of a parser generator for SDF. Many of the techniques presented in this chapter were originally developed for the translation of SDF to intermediate languages like context-free and regular grammars as prescribed by the SDF reference manual (Heering *et al.*, 1989). Gradually it became clear that the difficulty of this project was due to the monolithic design of SDF. The features presented in this chapter are combined in the formalism SDF2 that is intended to replace SDF. The specification of a parser generator for SDF2 was easier due to the uniform abstract syntax and elimination of cases by normalization. The tables generated by the generator are interpreted by the generic scannerless generalized-LR parser described in Chapter 3.

10.3.2 Disambiguation

Priorities are interpreted as a well-formedness requirement on parse forests, which could be operationalized as a filter on parse forests as prescribed by Heering *et al.* (1989). This approach can be extended to other disambiguation methods as described in Chapter 4.

We have provided some features for disambiguation of ambiguous context-free grammars. There remain a large number of ambiguities that can not be solved with these mechanisms. Some more advanced disambiguation methods are described in Klint and Visser (1994). Here we list some ideas for improvements of the current scheme.

The priority relation $>$ on productions does not allow a distinction between the arguments of the productions to which it applies. In several cases it would be useful to restrict the relation to certain arguments. For instance, the priority declaration

```
T T -> T  >  "let" V "=" T "in" T -> T
```

does correctly forbid the usage of a `let` expression as the first argument of an application. However it also forbids the usage of `let` as the last argument of an application, for which there is no reason. An extension of the notation could be

```
T T -> T  {1}>  "let" V "=" T "in" T -> T
```

to declare the desired disambiguation. There would be no implementation problems with such an extension.

A case for non-standard disambiguation is in ambiguous equations. In §9.3 we gave as an example the specification of the syntax of conditional equations. It can occur that equations are ambiguous due to injections. If a symbol $\mathcal{A}$ is injected in $\mathcal{B}$, then an equation over two $\mathcal{A}$ expressions can be interpreted both as $\mathcal{A}$ and $\mathcal{B}$ equations. A possible interpretation of such an ambiguity is to take both possibilities. This is done in the definition of multi-level algebraic specifications in Part III, where ambiguous equations can occur due to overloading of functions.

In the implementation of SDF in the ASF+SDF Meta-Environment an undocumented disambiguation method is used. As a simplification of the multi-set ordering, trees with fewer injections are preferred over trees with more injections. Such a method is needed to disambiguate conditions of equations. This method has not been implemented as part of the SDF2 tools, but can be added as a post-parse filter without problems.

We have defined follow restrictions and reject productions to express lexical disambiguation rules. We omitted the definition of these methods as an extension of the well-formedness predicate on parse trees. See Chapter 3 for a discussion of the semantics of these methods and for a discussion of automatic lexical disambiguation.

10.3.3 Renaming

Modules associate a name with a grammar. Grammars can be combined by module imports. Export and hiding provide control over visibility of grammars. New with respect to the modularization of SDF are renamings and hidden imports. In the current definition of renaming productions, only the literal skeleton of the production can be changed, but the order of the arguments stays the same. Sometimes it is desirable to change the syntax of a production and also make a permutation of the arguments. A notation for such permutations should be devised by means of some kind of indexing. The problem with such a notation is that the current definition reuses the syntax of productions literally in the definition of renamings. Changing the syntax of productions will thus be applicable everywhere. The label facility (see below) could be used for this purpose. Unfortunately, renamings are not guaranteed to preserve well-formedness of parse trees. Further study is needed to find a set of sufficient requirements on renamings that do guarantee well-formedness.

10.3.4 Labels

A feature that has not discussed are labels. Labels are intended to be used as 'field names' of a record. For instance, consider the following production defining the syntax of assignments in an imperative language:

```
var : Var ":=" value : Exp -> Stat {cons(assign)}
```

The two arguments are labeled with `var` and `value`, respectively. From this information we can derive the following syntax for projection functions based on the field names:

```
Stat "." var   -> Var
Stat "." value -> Exp
```

This should be accompanied by the the defining equations for these functions.

10.3.5 Derived Syntax

Regular expressions are considered as name constructors that are used to make new names out of existing ones. A normalization function adds canonical productions defining the regular operators. For instance, $\mathcal{A}?$ denotes an optional $\mathcal{A}$ and is defined by the productions $\mathcal{A} \to \mathcal{A}?$ and $\to \mathcal{A}?$. However, there is no restriction on the use of these name constructors. Other defining productions can be added by the user. In the context of algebraic specification this means for instance that users can specify functions that have lists ($\mathcal{A}*$) as result.

Regular expressions are an example of derived syntax: Given some symbol or even production in the grammar, other productions are derived. Many other applications of derived syntax could be useful.

Sometimes it is useful to explicitly indicate empty constructs and injections. This could be accommodated by generating syntax for explicitly matching injection functions and ϵ-functions, i.e., if `A -> B` is a production then also `"injA-B" (A) -> B` Similarly for ϵ, if `-> A` a production, then also `"emptyA" -> A`. These constructors should of course match with their origins. This can be done by translating these functions internally to the real injection or ϵ function. See Dinesh (1995) for some interesting remarks on injections in ASF+SDF.

A structure editor provides facilities to manipulate sentential forms. This requires the specification of the syntax of symbol placeholders. For each symbol `A` that is not a literal add a production `"<A>" -> A`.

Another case of this kind is the generation of explicit type casts `S ":" "S" -> S {cast}` (like bracket attribute) to constrain the type of an overloaded entity. This would be similar to the `no-operator` attribute in SDF.

It would be even better to make syntax derivation user-definable by providing schemas such as discussed above.

10.3.6 Polymorphic Syntax

The definition of regular expressions by introducing new productions is an instance of second order quantification. The generalization of this approach to two-level grammars in Chapter 15 provides the syntactic counterpart of the two-level specifications in Meinke (1992a) and the multi-level specifications in Part III. Generic productions are written as production schemata. The syntax of symbol constructors is described by means of a second level grammar.

10.3.7 Dynamic Syntax

Another open problem is the formal description of languages with an extensible syntax. Programs in such languages can contain grammars that define part of the syntax of the program itself. An example of extensible syntax is the syntax of equations in ASF+SDF. Several other instances exist, e.g., Cardelli *et al.* (1994), and Vittek (1994) (Elan). All these approaches treat the meta-language and object language differently. A formal approach to this problem would specify the syntax of the base language and the grammars it can specify and the lifting of these grammars to meta-level grammars.

10.3.8 Design Methodology

We have presented a large specification. We approached this using a rigorous modularization of the specification in a matrix of modules. For each feature the syntax and tools are described in separate modules. In this way it becomes feasible to flexibly include and exclude parts of a language definition. Some parts of the specification such as the ATerm encoding are not very interesting. It would be better if those parts could be generated using a simple rule.

The main technique we have applied is that of definition of features by normalization, i.e., transformation to a subset of the language. The great advantage of normalization is that many features can be provided to enhance the expressiveness of the language while defining the semantics of the formalism on a small set of kernel features in which the other features are expressed. Normalization has also its disadvantages. The semantics of various features is defined indirectly, which makes reasoning about them more troublesome. Furthermore, parse trees over a grammar use the normalized productions, which can look rather different than their origins. It would be desirable to use normalization equations rather than functions in order to be able to reason about equivalence of syntax definitions. The problem with such an approach is the lack of control over normalization. A solution could be the use of strategies such as described in Luttik and Visser (1997).

The modularization of the formalism and hence the modularization of the normalization in separate normalization functions for each feature made the specification of normalization feasible. A normalization function that would in one pass over the grammar normalize it would be a very complex. However, the modularization also hides the interaction between features. When defining a normalization function for an extension of the kernel, only those constructs introduced are normalized. The combination of features prompts the extension of the normalization function to new constructs. Often this is can be achieved by innocent distribution equations, but in some cases the interaction between features is more problematic. In particular, the interaction of renamings with other features needs more study.

Language design is a software engineering process. A language definition gets better developed if it is actually used in a prototype implementation. The parts of the specification of SDF2 that are used in the parser generator, i.e., the normalization, were developed on demand. Especially the fragment of SDF2 that corresponds to SDF was developed first, because most syntax definitions fed to the parser generator were converted SDF definitions. Other parts of the specification, such as well-formedness or equality that are not directly used in tools were developed later. But these parts are important because they define the correctness criteria for implementations. The well-formedness checker can be used to validate the output of a parser for a grammar. The equality checker can be used to validate a matching algorithm for terms.

The design approach we have used for SDF2 has led to an infrastructure for further study of syntax definition and experimentation with new features. It is indeed very easy to extend the specification in order to construct subsets or supersets of the formalism, or to replace a feature by a variant.

Part III

Multi-Level Algebraic Specification

11

Extensions of First-Order Specification

The next chapters introduce a modular, applicative, multi-level equational specification formalism that supports algebraic specification with user-definable type constructors, polymorphic functions and higher-order functions. Specifications consist of one or more levels numbered 0 to n. Level 0 defines the object level terms. Level 1 defines the types used in the signature of level 0. In general, the terms used as types at level n are defined at level $n+1$. This setup makes the algebra of types and the algebra of types of types, etc., user-definable. The applicative term structure makes functions first-class citizens and facilitates higher-order functions. The use of variables in terms used as types provides polymorphism (including higher-order polymorphism, i.e., abstraction over type constructors). Functions and variables can be overloaded. Specifications can be divided into modules. Modules can be imported at several levels by means of a specification lifting operation. Equations define the semantics of terms over a signature. The formalism also allows equations over types, by means of which many type systems can be described. The typechecker presented in Chapter 13 does not take into account type equations.

The specification, in ASF+SDF, of the syntax, type system and semantics of the formalism is presented in three stages: (1) untyped equational specifications (2) applicative one-level specifications (3) modular multi-level specifications. The definition of a typechecker for stages (2) and (3) is divided into four parts: (a) well-formedness judgements verifying type correctness of fully annotated terms and specifications, (b) non well-formedness rules giving descriptive error messages for the cases not covered under (a), (c) a type assignment function annotating the terms in a plain specification with types, and (d) a typechecking function which checks well-formedness after applying type assignment. These functions are defined uniformly for all levels of a specification.

Aside of defining a new specification formalism, the next chapters illustrate the use of ASF+SDF for the design and prototyping of sophisticated specification formalisms.

11.1 Introduction

Algebraic specification and functional programming are closely related paradigms. The foundation of both paradigms is equational logic. Values are represented by *terms* and a program or specification consists of a list of *equations* over these terms. Two terms that are equal according to a specification (by means of equational logic) have the same meaning and can replace each other in any context, a property called *referential transparency*.

The paradigms differ in the aim of a program or specification. An algebraic specification defines a class of algebras that satisfy its equations. A functional program on the other hand defines a method to compute a value from an initial value by executing the equations as rewrite rules. However, this difference is mainly one of emphasis; functional programs can be seen as algebraic specifications that satisfy certain restrictions. Almost all specifications in this book can be executed as rewrite systems. In spite of that, there are many technical differences between actual formalisms. These differences can be divided into semantics and type system.

11.1.1 Semantics

The choice of a semantics for a language is based on the set of required program constructs, which may include equations, conditional equations, λ abstraction, let binding, recursion and fixed-point operators, etc. In this chapter we use pure equational logic as the basis for the specification logic.

The operationalization of an equational algebraic specification by means of term rewriting is aimed at determining whether two terms are equal or at finding a normal form for a term. The strategy used to accomplish this is of no importance. Functional programming languages, emphasizing computation rather than specification, incorporate a rewrite strategy (innermost, outermost, lazy) into their semantics. Furthermore, functional languages make a distinction between functions that transform a value into another and constructors that are used to represent data. In algebraic specification this distinction is not made, e.g., the unary minus function '$-$' can be seen either as a constructor (-1) or as a function $(-0 = 0)$.

11.1.2 Type Systems

A *signature* determines which terms are the subject of a specification or program. A *type system* determines the form of signatures and the *well-formed* terms over a signature. Several issues are of importance in the design of type systems.

Term Structure First-order many-sorted algebraic specifications use a many-sorted algebraic signature to assign types of the form $s_1 \times \cdots \times s_n \rightarrow s_0$ to function symbols f. Terms can be formed by application of such function symbols to a list of terms t_i of sort s_i, resulting in terms of the form $f(t_1, \ldots, t_n)$. This function application construct is called *algebraic*. Such a type system is called first-order because no higher-order functions (having functions as arguments) can be defined. A function symbol can only occur in a term when it is applied to the right number of arguments. Other type systems allow higher-order functions and use an *applicative* term structure — application is of the form $t_1\ t_2$, term t_1 applied to term t_2 — to build terms. Applicative term structure is common in functional languages, whereas algebraic specification formalisms generally use first-order term structures.

Overloading If a function can have a finite number of different types it is said to be *overloaded.* An example of overloading is addition on integers and reals. Overloading is common in frameworks with algebraic term structure, where it is easy to deduce which version of a function is used from the arguments to which it is applied. In applicative frameworks ambiguities caused by overloading are much harder to resolve because functions can occur separate from their arguments. Therefore, overloading was omitted in early functional languages like ML. Most modern functional languages have some restricted form of overloading through type classes (see below).

Polymorphism Parametric polymorphic functions, which were introduced by Milner (1978) in the functional language ML, can have infinitely many types that are instantiations of one generic type. An example of a polymorphic function is the function that computes the length of a list, which is independent of the contents of lists and can therefore be defined for all possible lists at once. Polymorphic functions have a universally quantified type. For instance the type of length is $\forall\alpha.\mathtt{list}(\alpha) \rightarrow \mathtt{int}$.

Restricted polymorphism For some applications this unrestricted polymorphism is too strong. For instance, the polymorphic equality function with type $\forall\alpha.\alpha \times \alpha \rightarrow \mathtt{bool}$ also applies to functions, which is undesirable because function equality is not computable. In Standard ML (Milner *et al.*, 1990) the type of the equality function is defined on the subset of the set of all types for which equality is computable. This idea is generalized by Wadler and Blott (1989) by means of *type classes*, which are predicates on types that divide the set of types into subsets with certain properties that can be used to restrict the polymorphism of functions. For instance, if the class `eq` indicates all types on which equality can be defined, then the type of the equality function can be rephrased as $\forall\alpha.\mathtt{eq}(\alpha) \Rightarrow \alpha \times \alpha \rightarrow \mathtt{bool}$ to express that the type variable can only be bound to types for which the `eq` predicate holds, that is, those that are in the `eq` class. The type classes of Wadler and Blott (1989) are unary predicates on types. Jones (1992) gives a more general formulation of restricted polymorphism in his theory of *qualified types*, in which arbitrary predicates on types are allowed. Special cases of the theory are type classes, subtyping and extensible records.

Type Operators In frameworks with polymorphism the language of types becomes a user-definable set of terms and subject to a type system itself. In a first-order framework the type of lists of integers has a name like `int-list`. In a polymorphic framework one wants to quantify over the type of the contents of lists. By defining a type constructor `list` (a function from types to types), one can denote lists of integers as `list(int)` and arbitrary lists as `list(A)`, where `A` is a variable ranging over types.

Types of Types The language of types built from type constants and type constructors is itself an algebraic language with a signature. In many-sorted algebraic signatures the only type constructors are $\times$ and $\rightarrow$ and the language of types is restricted to types of the form $c_1 \times \cdots \times c_n \rightarrow c_0$, where the c_i are type constants. In polymorphic languages like ML the language of types consists of untyped, first-order terms, i.e., all type constructors have a type of the form $\mathtt{type} \times \cdots \times \mathtt{type} \rightarrow \mathtt{type}$. For instance, `list` is a type constructor that takes a type and constructs a type, i.e., it is declared as $\mathtt{list} : \mathtt{type} \rightarrow \mathtt{type}$. Generalizing the idea of an algebra of type constructors, one can use an arbitrary many-sorted (instead of a one-sorted) signature for the specification of the algebra of types, leading to a two-level signature. Further generalization of this idea leads to a third-level signature that specifies the types of types of types. In this chapter a formalism with multiple levels of signatures is presented.

Higher-Order Polymorphism and Constructor Classes In Hindley/Milner type systems the quantifier in types can only range over types and not over type constructors. Higher-order polymorphic functions can also quantify over type constructors. With such polymorphism it is natural to extend the notion of a type class to a constructor class which restricts quantification over type constructors (Jones, 1995).

There are many other considerations in the design of type systems. Here we restrict our attention to the ones discussed above. See §14.6 for some references to surveys of type systems.

11.2 Multi-Level Specifications

In the chapters in this part we present the formalism MLS, a *modular, applicative, multi-level, equational specification formalism with overloading.* Figure 11.1 illustrates several features of this language by means of a two-level specification of lists and trees with polymorphic `size` and `map` functions. The specification imports the specification of the type `nat` of natural numbers with functions `0`, `s` and `(+)`.

Multi-level A specification consists of arbitrary many levels of one-level specifications. The terms over the signature at level 0 are the 'object' level terms. The types used in the signature of level 0 are terms over level 1. In general, the types in the signature at level n are terms over the signature at level $n+1$, as is depicted in Figure 11.2. The types used in the signature of the highest level are determined by an implicit signature of types consisting only of type constants and the type constructors $\times$ and $\rightarrow$.

The sort declarations at level n determine which of the terms at level $n+1$ can actually be used as type at level n. A term used as type should match one of

```
module list-tree
imports nat;
level 1
  signature
    sorts type;
    functions
      (#), (->)  : type # type -> type;
      list, tree : type -> type;
    variables
       A, B : type;
level 0
  signature
    sorts A;
    functions
      []   : list(A);
      (::) : A # list(A) -> list(A);
      size : list(A) -> nat;
      map  : (A -> B) # list(A) -> list(B);
    variables
      X : A; L : list(A); G : A -> B;
  equations
    size([])       == 0;
    size(X :: L)   == s(size(L));
    map(G, [])     == [];
    map(G, X :: L) == G(X) :: map(G, L);
  signature
    functions
      []   : tree(A);
      node : tree(A) # A # tree(A) -> tree(A);
      size : tree(A) -> nat;
      map  : (A -> B) # tree(A) -> tree(B);
    variables
      X : A; T : tree(A); G : A -> B;
  equations
    size([])               == 0;
    size(node(T, X, T'))   == s(size(T) + size(T'));
    map(G, [])             == [];
    map(G, node(T, X, T')) == node(map(G, T), G(X), map(G, T'))
```

Figure 11.1: Two-level specification of list and tree data types.

the terms declared as sort. These ideas are illustrated in Figure 11.1. The term `type # type -> type` in the first function declaration at level 1 is a term over the implicit signature of the types at the highest level. (Note that $\times$ is written

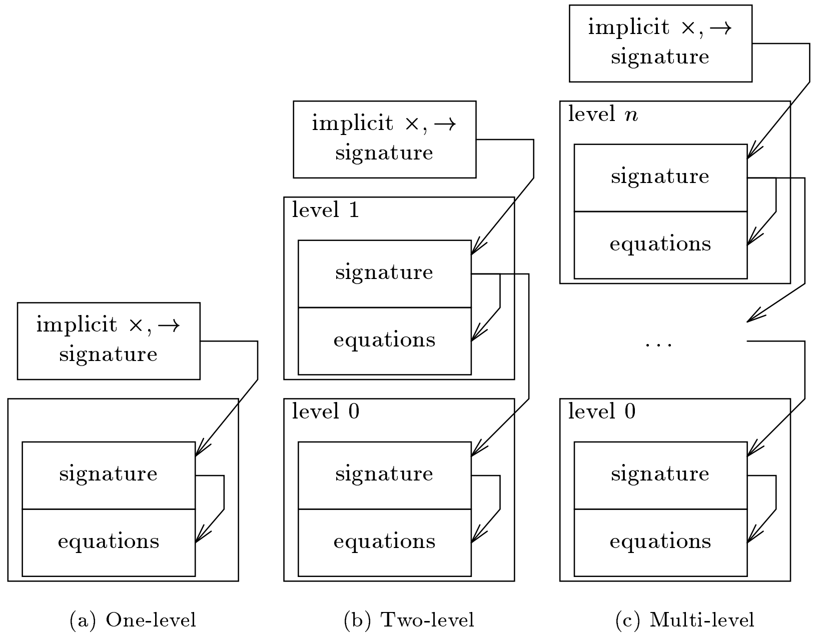


Figure 11.2: Specifications with one, two and multiple levels of signatures.

`#` in ASCII notation.) The term `list(A)` is a term over the signature at level 1: `list` is a function from `type` to `type` and `A` is a `type` variable. Furthermore, `list(A)` matches the sort declaration `A`. Therefore, `list(A)` can be used in the signature at level 0 as a type in the declaration of the functions `[]` (empty list), `(::)` (cons) etc. Level 0, finally, determines the terms for the objects of real interest, such as `[]`, `s(0) :: []`, and `map(s)(0 :: [])`.

The example in Figure 11.1 shows a two-level specification ($n = 1$). The formalism supports arbitrarily many levels. The type constructors available at level 1 can be enriched by means of a third level. In §13.1 several examples of three level specifications are shown.

Polymorphism Terms over a signature can contain variables. A term with variables used as type in a signature denotes a *polymorphic* type. For instance, `size` is a function from `list(A)` to `nat`. This means that for any `type` t, `size` applies to terms of `type list(`t`)`. Quantification is not restricted to types but can also range over type constructors.

Overloading Functions can have two or more related, or completely different, types. This allows the use of function names for different purposes, which is not possible with polymorphism alone. For instance, the functions `size` and `map` are defined for both lists and trees. Equations can also be overloaded. For example, the equations defining the functions `size` and `map` on empty lists and empty trees are exactly the same. Actually, writing this equation once would have sufficed, because all possible interpretations of ambiguous equations are taken into consideration.

Applicative The term structure is applicative, i.e., application is a binary operation on terms. At the functional position an arbitrary term can occur. Functions are first-class citizens and can be arguments of functions. For instance, the function `map` has a function as argument, which it applies to all elements of a list or tree.

Observe that the arrow and product constructors for types are considered normal functions. The arrow in the type of `size` is the same arrow that is declared at level 1 as a binary function on types. There is, however, one difference with other functions: the arrow and product constructors are related to the operations application and pairing. For each arrow type, there is a corresponding application operation that takes a term of type $\tau_1 \to \tau_2$ and a term of type τ_1 and produces a term of type τ_2. Similarly for each product type there is a corresponding pairing operation that takes two terms of types τ_1 and τ_2 and produces a term of type $\tau_1 \times \tau_2$.

Equational Equational axioms over terms express the semantics of terms. Equational logic can be used for reasoning about terms, whereas term rewriting can be used to decide ground equations for appropriate systems of equations or to compute the result of defined functions. The ideas for the multi-level type system in this chapter are also applicable to formalisms with other logics, e.g., conditional equations, Horn clause logic or even first-order logic.

Modular Multi-level specifications can be split into modules by means of a rudimentary module system consisting of module declarations and module references (imports). Operations for manipulating specifications can also be applied to imports, facilitating reuse of specifications at more than one level (see §13.1 for examples).

Type Equations The MLS *formalism* supports equations at all levels of a specification. This means that equations over types can be defined to specify powerful type constructs like recursive types, qualified types, and logical frameworks. However, the typechecker for MLS defined here does not take into account equations over types. Typechecking in the presence of type equations requires $\mathcal{E}$-unification, which is undecidable in general. For restricted forms of equations typechecking with $\mathcal{E}$-unification seems feasible, and might be incorporated in future versions of the MLS typechecker (Visser, 1996b).

11.3 Related Formalisms

The MLS formalism is a generalization of several concepts found in other formalisms. Below we give a brief overview of related formalisms. The landscape of formalisms is summarized by the diagram in Table 11.1.

features	algebraic	# levels	applicative	features
ol	OBJ, Pluss, ASF+SDF	1	OLS	hof
		2	ML, Miranda	hof, p
p, tc	PolySpec		Spectrum, Haskell	hof, p, tc
p, ol	ATLAS	3	Quest	hof, p, st
p, ol	ATLASII	n	MLS	hof, p, ol

Table 11.1: Several algebraic and functional languages classified according to their number of levels and to their term structure (algebraic vs. applicative). The additional features columns list the presence of: ol: overloading, hof: higher-order functions, p: polymorphism, tc: type classes, st: subtypes.

One-Level Monomorphic Algebraic Languages Algebraic specification formalisms such as OBJ (Futatsugi *et al.*, 1985), Pluss (Bidoit *et al.*, 1989) and ASF+SDF (see Chapter 2) have monomorphic many-sorted first-order signatures as type system. The sort space consists of terms of the form $c_1 \times \cdots \times c_n \rightarrow c_0$, with the c_i sort constants. A limited form of polymorphism can be obtained by means of overloading and parameterized modules, but polymorphic higher-order functions are not provided. All these formalisms support arbitrary mixfix notation. OBJ provides *order-sorted* signatures, in which an inclusion relation between sorts can be declared. In ASF+SDF, sort inclusion can be simulated by means of syntaxless unary functions (also called injections). The formalisms OLS and MLS considered here support neither subsorting nor syntaxless functions.

One-Level Monomorphic Applicative Languages The one-level applicative specification language OLS, defined in §12.2 and §12.3, generalizes the sort space of monomorphic algebraic languages to the closure under $\times$ and $\rightarrow$ of the declared sort constants. The extension with respect to the algebraic frameworks discussed above is the support for higher-order functions.

Two-Level Polymorphic Applicative Languages The type system for polymorphic higher-order functions, known as the Hindley/Milner system, was first described by Hindley (1969) as a type assignment algorithm for expressions in combinatory logic. It was extended by Milner (1978) to languages with local declarations. The functional programming language ML (Gordon *et al.*, 1978) was the first language to incorporate this type system. For the introduction of type operators, the type system of ML uses a second level of terms consisting of an untyped, first-order signature. All type operators work on one implicit type (kind) of types. ML is not purely functional because it supports side effects through assignments in expressions. Miranda (Turner, 1985) is one of a number of purely functional languages with a Hindley/Milner type system. Haskell is a general purpose, purely functional programming language (Hudak *et al.*, 1992) with a Hindley/Milner type system using one-sorted first-order user-definable type constructors. Overloading, which is not supported in ML and Miranda, is introduced in a restricted form through type classes (see §11.1.2), which are the main innovation of the language.

The requirement and design specification language Spectrum (Broy *et al.*, 1993) is an algebraic specification formalism with applicative term structure, a two-level type system and sort classes, which is a variant of type classes. The second level is an unsorted signature. The difference with functional languages like Haskell is the use of full first-order logic instead of conditional equations.

Two-Level Polymorphic Algebraic Languages The algebraic specification formalism PolySpec of Nazareth (1995) is a two-level formalism, with an untyped second level of type constructors and predicates (sort classes), which are used to constrain polymorphism similarly to type classes.

Both the algebraic and the applicative two-level languages that we have discussed sofar have an untyped second level: all type constructors operate on the single, implicit sort `type`.

Three-Level Applicative Languages Quest is a three level language inspired by second-order typed λ-calculus (Cardelli, 1993). A Quest program introduces objects at three levels: values at level 0, types and type operators at level 1 and kinds at level 2. Instead of the limited universal type quantification of Hindley/Milner type systems, explicit and nested quantification over types is allowed. Universally quantified types, i.e., polymorphic types, have to be instantiated explicitly. For example, the identity function, declared as $id : \forall\alpha.\alpha \to \alpha$, should first be applied to a type to instantiate the type variable and then to a value, e.g., $id[int](1)$. Cardelli (1993) discusses a rich set of built-in data types including mutable types, array types, exception types, tuple types, option types, recursive types, subtyping, operations at the level of types. Quest does not support overloading.

Three-Level Algebraic Languages The algebraic specification formalism ATLAS of Hearn and Meinke (1994) is a three-level algebraic specification formalism. The main differences with MLS are: (1) ATLAS has an arrow type constructor for the type of functions and a product type constructor for the type of pairs that are primitive at all levels, and that can be used as first-order types of the form $\tau_1 \times \cdots \times \tau_n \to \tau$, which means that term structure is algebraic. Higher-order function application can be simulated by means of a user-defined arrow type constructor and a user-defined application operator and by declaring functions as constants of the user defined arrow type. MLS has an applicative instead of an algebraic term structure, which makes higher-order types and functions more naturally definable. (2) An ATLAS specification consists of three levels for the constructors of 'kinds', 'types' and 'combinators' as the different sorts of terms are called. MLS specifications can have arbitrary many levels instead of the fixed three levels of ATLAS, making the definition of the syntax and type system uniform for all levels and enabling specifications with more or fewer than three levels. (3) ATLAS does not have a module system. (4) ATLAS considers ambiguous equations as erroneous. In MLS all well-formed typings of an equation are considered valid. (5) ATLAS specifications can contain rewrite rules at all levels, which are interpreted by the type assignment mechanism. Although the MLS formalism allows equations at all levels, these are not considered by the type assignment algorithm specified in this chapter.

Multi-Level Algebraic Languages ATLASII is a multi-level and modular redesign of ATLAS (Hearn, 1995). Items (1), (4) and (5) above also hold for ATLASII.

Multi-Level Applicative Languages The specification formalism MLS defined in this chapter is an applicative multi-level language with overloading.

11.4 Outline

The next chapters present the multi-level specification formalism MLS by means of a specification in ASF+SDF of syntax, type system and semantics. In order not to introduce too many concepts and technical details at once, the equational specification formalism is presented in three phases, each enhancing the previous one: (1) an untyped formalism, (2) a one-level applicative formalism without overloading or polymorphism, and (3) a multi-level, applicative formalism with polymorphism and overloading.

In §12.1 the notions of terms and equations for the untyped language are defined. Specifications are lists of equations over a simple term language with application and pairing. In §12.2 this untyped language is extended to a one-level language, after introducing the notions of types and signatures. In §12.3 a typechecker for this specification language is defined as the composition of a type assignment function and a well-formedness checker. The type assignment function takes a plain term and annotates it with types. The well-formedness checker takes a fully annotated term and verifies its well-formedness. The specification is presented in four parts: *Well-formedness* judgements determine whether a fully annotated term is well-formed according to a signature. The complements of the rules for well-formedness give descriptive *error messages* for non-wellformed terms. A *type assignment* function annotates each subterm of a plain term with a type. A *typechecker* combines type assignment and well-formedness checking.

In Chapter 14 one-level specifications are used to form multi-level specifications. The same syntax for terms, signatures and equations is used at all levels. The usefulness of such multi-level specifications is illustrated with several examples of data type specification in Chapter 13. The type system of multi-level specifications is defined with the same four part structure as for one-level specifications. The same ideas apply to the type system, but are complicated by the addition of multiple levels of signatures, polymorphism and overloading. The most important innovation here is that the types of each level of the specification are well-formed terms over the signature at the next level of the specification. This means that types become typed terms. The same typechecking mechanism is used at all levels.

In Appendix B a number of auxiliary tools that are used in the specification are defined. In §B.1 several 'standard' library modules like Layout and Booleans are defined. In §B.2 several utilities on terms such as sets of terms, substitution, matching and unification are defined.

12

Untyped and Simply Typed Specifications

In this chapter we lay the foundations for the definition of multi-level specifications. First we define an untyped equational specification formalism with equational logic and term rewriting. Next we specify a language of signatures that can be used to restrict the terms used in equations. Signatures are interpreted by a well-formedness predicate on fully annotated terms. A type assignment function annotates untyped terms with types according to a signature.

12.1 Untyped Equational Specifications

Equational specifications consist of a list of equations over some term language. Such specifications can be interpreted as a set of axioms for reasoning with equational logic. For many specifications, equality of terms in the context of an equational specification can be made by means of term rewriting. We start with the definition of the term language.

12.1.1 Terms

The *terms* of our specification language are simple applicative terms composed of function symbols (identifiers starting with a lowercase letter, e.g., `map`), variables (identifiers starting with an uppercase letter, e.g., `X`), application ($t_1\ t_2$), and pairing (t_1, t_2). Application is left-associative and has a higher priority than pairing. Pairing is right-associative. For example, `map G empty` denotes `((map G) empty)`, not `map(G(empty))`. Likewise, `plus X, Y` should be read as `(plus X), Y` and not as `plus(X, Y)`. In this chapter we will write the argument of an application between parentheses, e.g., `map(G)(empty)` instead of `map G empty`. These notations are syntactically equivalent according to the following grammar.

module Terms
imports Layout$^{B.1.1}$
exports
  sorts Fun Var Term

lexical syntax
`[a-z0-9][A-Za-z0-9_]*` → Fun
`[A-Z][A-Za-z_]*[0-9']*` → Var
context-free syntax
Var → Term
Fun → Term
Term Term → Term {**left**}
Term "," Term → Term {**right**}
"(" Term ")" → Term {**bracket**}
priorities
Term Term → Term > Term ","Term → Term
variables
[*xy*][*0-9′*]* → Var
"*f*"[*0-9′*]* → Fun
"*t*"[*0-9′*]* → Term

To accommodate the convention of writing binary functions as infix operators, §B.2.1 defines syntax for infix operators. The application of a binary operator $\oplus$ to two arguments t_1 and t_2 is written $t_1 \oplus t_2$. By enclosing a binary operator in parentheses it is converted into a prefix function symbol. Using this property an infix application is translated into a prefix application by the equation $t_1 \ \oplus \ t_2 = (\oplus)(t_1, t_2)$. For example, in Figure 11.1 the expression `size(T) + size(T')` is equivalent to `(+)(size(T), size(T'))` and `X :: L` is equivalent to `(::)(X, L)`. Furthermore, §B.2.1 introduces notation to use an arbitrary term as an infix operator, such that a binary function application of the form $t_1(t_2, t_3)$ can be written as t_2 .t_1. t_3. Finally, if the functions `(::)` and `[]` are used to construct lists, the notation `[`t_1`, ... ,`t_n`]` can be used to represent a list with a fixed number of elements. This notation is translated to t_1 `:: ... ::` t_n `:: []`. Note that using the `[`t_1`, ... ,`t_n`]` notation the tail of the list is always `[]`, i.e., can not be a variable or another term. Similarly, tuple terms of the form `<`t_1`, ... ,`t_n`>` are abbreviations for t_1 `^ ... ^` t_n `^ <>`.

The extension of multi-level signature formalisms with arbitrary mix-fix operators (like `if _ then _ else _`) leads to a multi-level grammar formalism. Such a formalism leads to extra complications in parsing. This is the subject of Chapter 15.

Lists of terms separated by semicolons.
exports
sorts Terms
context-free syntax
{Term ";"}* → Terms
Terms "++" Terms → Terms {**right**}
"(" Terms ")" → Terms {**bracket**}
variables
"*t*""*"[*0-9′*]* → {Term ";"}*
"*t*""+"[*0-9′*]* → {Term ";"}+
"*ts*"[*0-9′*]* → Terms

equations

[1] $$t_1^* +\!\!+\; t_2^* = t_1^*;\, t_2^*$$

12.1.2 Equations

An *equation* is a pair of terms $t_1 \equiv t_2$. In order to avoid confusion between the equality symbol in the object language we are describing and the metalanguage we describe it with, the symbol $\equiv$ is used for specification equations. It is written `==` in examples. We will refer to the left-hand (right-hand) side t_1 (t_2) of an equation by 'lhs' ('rhs'). An *equational specification* is a list of equations.

module Equations
imports Binary-Operators$^{B.2.1}$ Terms$^{12.1.1}$
exports
  sorts Eq Eqs
  context-free syntax
    Term "≡" Term → Eq
    {Eq ";"}* → Eqs
    Eqs "++" Eqs → Eqs {**assoc**}
    "(" Eqs ")" → Eqs {**bracket**}
  variables
    "φ"[*0-9′*]* → Eq
    "φ" "*"[*0-9′*]* → {Eq ";"}*
    "φ" "+"[*0-9′*]* → {Eq ";"}+
    "$\mathcal{E}$"[*0-9′*]* → Eqs
equations

[1] $$\varphi_1^* +\!\!+\; \varphi_2^* = \varphi_1^*;\, \varphi_2^*$$

An example specification is shown in Figure 12.1. The first two equations define the addition operation `(+)` on successor naturals. The last two equations define the function map that applies some function `G` to all elements of a list represented by means of the functions `[]` (empty list) and `(::)` (cons). Observe that some of the parentheses used are optional, e.g., we might as well write `G X` instead of `G(X)`. Recall that we will use the convention of writing the argument of an application between parentheses.

12.1.3 Equational Logic

A term represents a value. In an equational specification a term represents the same value as all terms to which it is equal. In this view the semantics of a specification is the equality relation on terms that it induces. This relation is determined by the following rules of equational logic together with a list of equations (also called axioms). Two terms t_1 and t_2 are equal according to a

```
0 + X           == X;
s(X) + Y        == s(X + Y);
map(G)([])      == [];
map(G)(X :: L)  == G(X) :: map(G)(L)
```

Figure 12.1: Untyped equational specification of addition on successor naturals and map over cons lists.

set of equations $\mathcal{E}$ if the predicate $\mathcal{E} \vdash t_1 \equiv t_2$ holds. Note that predicates are modeled by means of Boolean functions in ASF+SDF. This entails that the specification of a predicate consists of equations over sort Bool. If P is a Boolean function we will write $P(x)$ in texts when we mean $P(x) = \top$.

The rules of equational logic are the reflexivity, symmetry and transitivity rules of equivalence relations; an axiom rule that declares any equation in $\mathcal{E}$ as axiom; a substitution rule that makes any substitution instance of a derivably equation derivable; and congruence rules. The substitution rule [5] uses the notation $\sigma(t)$ for the application to a term t of a substitution σ that maps variables to terms. (See §B.2.7 for the definition of substitution.)

module Equational-Logic
imports Equations$^{12.1.2}$ Substitution$^{B.2.7}$ Booleans$^{B.1.2}$
exports
context-free syntax
Eqs "⊢" Eq → Bool
equations

[1] $$\mathcal{E} \vdash t \equiv t \;=\; \top$$

[2] $$\frac{\mathcal{E} \vdash t_2 \equiv t_1 = \top}{\mathcal{E} \vdash t_1 \equiv t_2 \;=\; \top}$$

[3] $$\frac{\mathcal{E} \vdash t_1 \equiv t_2 = \top,\ \mathcal{E} \vdash t_2 \equiv t_3 = \top}{\mathcal{E} \vdash t_1 \equiv t_3 \;=\; \top}$$

[4] $$\varphi_1^*;\, t_1 \equiv t_2;\, \varphi_2^* \vdash t_1 \equiv t_2 \;=\; \top$$

[5] $$\frac{\mathcal{E} \vdash t_1 \equiv t_2 = \top}{\mathcal{E} \vdash \sigma(t_1) \equiv \sigma(t_2) \;=\; \top}$$

[6] $$\frac{\mathcal{E} \vdash t_1 \equiv t_3 = \top,\ \mathcal{E} \vdash t_2 \equiv t_4 = \top}{\mathcal{E} \vdash t_1\ t_2 \equiv t_3\ t_4 \;=\; \top}$$

[7] $$\frac{\mathcal{E} \vdash t_1 \equiv t_3 = \top,\ \mathcal{E} \vdash t_2 \equiv t_4 = \top}{\mathcal{E} \vdash t_1,\, t_2 \equiv t_3,\, t_4 \;=\; \top}$$

This specification is not executable as a term rewrite system, because it is non-deterministic and not normalizing. This is not surprising since equational derivability is an undecidable property. To determine whether two terms are equal we can make use of several other techniques. In the following subsection we define an evaluation function that implements a simple rewrite strategy that decides (ground) equality for a large class of specifications.

12.1.4 Term Rewriting

Equational specifications can be interpreted as *term rewriting systems* by directing the equations from left to right. This gives a procedure for deciding derivable equality from a set of equations that constitutes a terminating and confluent rewrite system. *Evaluation* of a term in the context of a specification amounts to finding its *normal form*, if it exists, with respect to the term rewriting system. If $\mathcal{E}$ is a list of equations and t is a term, then $t' = \text{eval}(\mathcal{E})[\![t]\!]$ is the normal form of t under $\mathcal{E}$, i.e., t' has no sub-term that matches the left-hand side of an equation in $\mathcal{E}$.

There are a number of strategies used to find normal forms. Here we use a simple left-most innermost rewriting algorithm. This strategy is sound with respect to equational logic, i.e., if two terms have the same normal form they are also derivably equal. The strategy is (ground) complete with respect to confluent and strongly normalizing term rewrite systems, i.e., two terms are derivably equal if and only if they have the same normal form.

Evaluation proceeds as follows. The auxiliary function 'step' tries to find a matching equation for a term. If it finds one, the instantiation of its right-hand side is evaluated. In equation [6] the list of equations is searched (by means of list matching, see Chapter 2) for an equation $t_1 \equiv t_2$ such that the left-hand side t_1 matches the term t, i.e., such that there is a substitution σ such that $\sigma(t_1) = t$. The substitution is found in the condition $t_1 := t = \sigma$. The substitution σ forms the environment for the evaluation of the right-hand side of the equation. If no matching equation is found, 'step' just returns its argument (equation [7]). The function 'eval' itself evaluates a term by first evaluating its direct sub-terms and then applying 'step' to the composition of the resulting normal forms.[1]

module Term-Rewriting
imports Matching$^{B.2.8}$ Equations$^{12.1.2}$
exports
  context-free syntax
    eval "(" Eqs ")" "[[" Term "]]" → Term
    eval "(" Eqs ")" "[[" Term "]]" "_" Subst → Term
    step "(" Eqs ")" "[[" Term "]]" → Term

[1]Note that the underscore _ in the syntax of the function 'eval' is interpreted by the ASF+SDF to LaTeX typesetting program by typesetting the next argument, i.e., the substitution, as a subscript.

equations

[1] $\text{eval}(\mathcal{E})[\![t]\!] = \text{eval}(\mathcal{E})[\![t]\!]_{[]}$
[2] $\text{eval}(\mathcal{E})[\![x]\!]_\sigma = \sigma(x)$
[3] $\text{eval}(\mathcal{E})[\![f]\!]_\sigma = \text{step}(\mathcal{E})[\![f]\!]$
[4] $\text{eval}(\mathcal{E})[\![t_1\ t_2]\!]_\sigma = \text{step}(\mathcal{E})[\![\text{eval}(\mathcal{E})[\![t_1]\!]_\sigma\ \text{eval}(\mathcal{E})[\![t_2]\!]_\sigma]\!]$
[5] $\text{eval}(\mathcal{E})[\![t_1,\ t_2]\!]_\sigma = \text{step}(\mathcal{E})[\![\text{eval}(\mathcal{E})[\![t_1]\!]_\sigma,\ \text{eval}(\mathcal{E})[\![t_2]\!]_\sigma]\!]$
[6] $\text{step}(\mathcal{E})[\![t]\!] = \text{eval}(\mathcal{E})[\![t_2]\!]_\sigma$
$\quad$ **when** $\mathcal{E} = \varphi_1^*;\ t_1 \equiv t_2;\ \varphi_2^*,\ \ t_1 := t = \sigma$
[7] $\text{step}(\mathcal{E})[\![t]\!] = t$ **otherwise**

The following proposition states that evaluation is sound with respect to derivable equality.

Proposition 12.1.1 (soundness of evaluation) *If $\mathcal{E}$ constitutes a terminating term rewrite system and* $\text{eval}(\mathcal{E})[\![t]\!]_\sigma = t'$, *then* $\mathcal{E} \vdash \sigma(t) \equiv t'$ *and if* $\text{step}(\mathcal{E})[\![t]\!] = t'$, *then* $\mathcal{E} \vdash t \equiv t'$

Proof. By simultaneous induction on the definition of eval and step. □

Observe that the specification of evaluation is not sufficiently-complete, because the 'eval' of a non-terminating term cannot be eliminated and thus is a new term constructor. The restriction to terminating rewrite systems in the soundness proposition is necessary because the definition of equational logic does not account for these new term constructors. This could be repaired by introducing an auxiliary sort as the result of evaluation and using conditional equations to define 'eval' as in

$$\frac{\text{eval}(\mathcal{E})[\![t_1]\!] = t_1', \text{eval}(\mathcal{E})[\![t_2]\!] = t_2'}{\text{eval}(\mathcal{E})[\![t_1\ t_2]\!] = \text{step}(\mathcal{E})[\![t_1'\ t_2']\!]}$$

The conditions work as 'retracts' and guarantee that the rule only applies if the evaluation of the subterms terminate, thereby avoiding the pollution of the sort Term. However, this gives a more complicated specification that does not have a better termination behavior and adds nothing to our understanding of term rewriting. Therefore, we leave the specification as it is, with the somewhat loose understanding that it says what we intend for terminating specifications.

12.2 One-Level Specifications

The untyped equations of the previous section do not impose a restriction on the set of terms that they describe. Although we have an intuition about the terms that are meaningful with respect to a specification and those that are not, this is not formalized. For instance, the specification in Figure 12.1 clearly manipulates two categories of terms: numbers composed by `0`, `s` and `(+)` and lists composed by `[]`, `(::)` and `map`. However, `s(map) + 0` is a valid term over this specification, which has no apparent meaning in our intuition about the specification.

Signatures formalize the intuition about the types of terms in specifications and allow one to check that specifications and terms comply with each other. A signature is a list of declarations of functions and variables that is interpreted as a predicate on terms indicating which terms are well-formed. In this section, we extend the untyped equational specification formalism with signatures, leading to the one-level specification formalism OLS.

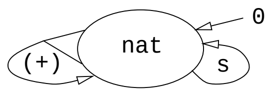
12.2.1 An Example

Before giving the syntax of type terms, signatures and specifications we discuss a simple example of a one-level specification. Figure 12.2(a) presents the specification of natural numbers in OLS. The signature part declares the constant `nat` as a sort and the constant `0`, the unary function `s` and the binary function `(+)`. Furthermore, the signature declares `X` and `Y` as `nat` variables. Together these declarations define the terms of sort `nat`. The equation part defines the meaning of the binary function `(+)` in terms of `0` and `s`.

The signature of this specification is depicted by the *signature diagram* in Figure 12.2(b). The diagram consists an ellipse denoting the set of all terms of sort `nat`. The arrows denote the functions declared in the signature. The constant `0` is denoted by an arrow without origin. The unary function `s` is denoted by an arrow from `nat` to `nat`; it takes a natural number and produces a new one. The binary function `(+)` takes two natural numbers and produces a new one, which is depicted by the forked arrow.

```
signature
  sorts nat;
  functions
    0   : nat;
    s   : nat -> nat;
    (+) : nat # nat -> nat;
  variables
    X, Y : nat;
equations
  0 + X    == X;
  s(X) + Y == s(X + Y)
```

(a)



(b)

Figure 12.2: Specification of successor naturals with addition (a) and corresponding signature diagram (b).

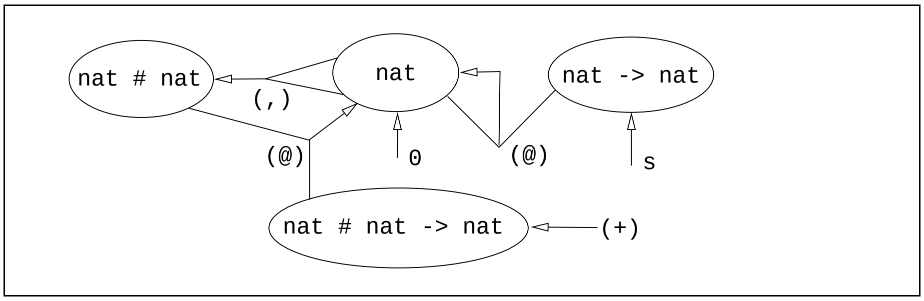


Figure 12.3: Signature diagram of natural numbers in which function and product types and the corresponding application and pairing functions are depicted explicitly. The functions `s` and `(+)` are constants of functional types.

As we will see, the term structure of one-level specifications is actually applicative. This entails that besides `nat`, there are the sorts `nat -> nat`, `nat # nat` and `nat # nat -> nat`. The signature diagram in Figure 12.3 depicts this situation. The functions `s` and `(+)` are constants of sorts `nat -> nat` and `nat # nat -> nat`, respectively. The diagram also shows the role of the implicitly declared pairing `(,)` and application `(@)` functions.

12.2.2 Types

A *type* is an expression that denotes a set of terms. Types in many-sorted signatures are composed of constants, such as `nat`, by means of the type operators product $\times$ and arrow $\rightarrow$. The product type $\tau_1 \times \tau_2$ denotes pairs of terms (t_1, t_2) of type τ_1 and τ_2, respectively. The type $\tau_1 \rightarrow \tau_2$ denotes the type of functions with domain τ_1 and codomain τ_2. The types in polymorphic languages are $\{\times, \rightarrow\}$-types extended with arbitrary terms like `list(nat)`. We will see later that such types can be described by a signature. Anticipating this extension, we use terms extended with the product and arrow operators as types. The variable τ, ranging over terms, will be used to indicate a term used as type.

A *type annotation* of a term is the attachment of a type to each subterm. Annotation is expressed by means of the operator ':'. The term $t : \tau$ denotes the term t annotated with type τ. A term is *fully annotated* if each subterm has a type annotation. For example, the term

```
(s : nat -> nat)(0 : nat) : nat
```

is a fully annotated version of the term `s(0)`. In the context of a signature, a term without annotations is an abbreviation of an annotated term. In the multi-level extension that we will define in §14.1 and §14.3 we will encounter terms with annotations that are themselves annotated, e.g.,

```
[] : ((list : type -> type)(A : type) : type)
```

is the term `[]` annotated with the type `list(A)`, which is itself annotated. Compare the annotation of `list(A)` with that of `s(0)` above.

module Types
imports Terms[12.1.1]
exports
  context-free syntax
    "nil" → Term
    "top" → Term
    Term "×" Term → Term {**right**}
    Term "→" Term → Term {**right**}
    Term ":" Term → Term {**right**}
  priorities
    Term Term → Term > Term "×" Term → Term >
    Term "→" Term → Term > Term ":" Term → Term >
    Term "," Term → Term
  variables
    "τ" [*0-9'*]* → Term
    "τ" "*" [*0-9'*]* → {Term ";"}*
    "τ" "+" [*0-9'*]* → {Term ";"}+

The terms 'nil' and 'top' are auxiliary types that will be used in typechecking. 'nil' denotes the empty type, which is assigned to terms for which no type exists. In our multi-level setting, 'top' will denote the type of top-level types, i.e., terms over the implicit signature on top of a multi-level specification.

The priorities section declares that application has highest priority of all term constructors and that product binds stronger than arrow, which has higher priority than type annotation and pair. For instance, consider the following disambiguations:

text	disambiguated as
`list : type -> type`	`list : (type -> type)`
`list A -> nat`	`(list A) -> nat`
`nat # nat -> nat`	`(nat # nat) -> nat`
`list : type -> type A : type`	`list : ((type -> (type A)) : type)`

12.2.3 Term Analysis

Recall that we have the following term constructors: variable and function symbols, nil, top, application, pairing, product, arrow and annotation. These are all the constructors we will consider in this chapter. All other functions that produce terms should be such that they can always be eliminated (i.e., the specification is assumed to be sufficiently complete). Assuming this property, a default (otherwise) equation over a function with a term as argument ranges over all constructors for which no other equation is defined, and thus is an abbreviation for a list of equations with those other constructors substituted.

For future use we now define several functions for analyzing terms. The sort (Term ⇒ Term) is the sort of functions from terms to terms that is defined in §B.2.3. The basic operation of this sort is the application of a function to a term yielding a term, i.e., (Term ⇒ Term)(Term) → Term. This approach makes it possible to generically define a function that applies a (Term ⇒ Term) function to all terms in a list of terms.

module Term-Analysis
imports Term-Functions$^{B.2.3}$ Terms$^{12.1.1}$ Types$^{12.2.2}$ Binary-Operators$^{B.2.1}$
exports
context-free syntax

spine $\rightarrow$ (Term $\Rightarrow$ Term)
fspine $\rightarrow$ (Term $\Rightarrow$ Term)
term $\rightarrow$ (Term $\Rightarrow$ Term)
type $\rightarrow$ (Term $\Rightarrow$ Term)
dom $\rightarrow$ (Term $\Rightarrow$ Term)
cod $\rightarrow$ (Term $\Rightarrow$ Term)
fun $\rightarrow$ (Term $\Rightarrow$ Term)
arg $\rightarrow$ (Term $\Rightarrow$ Term)
bterm $\rightarrow$ (Term $\Rightarrow$ Term)
bapp $\rightarrow$ (Term $\Rightarrow$ Term)

equations

The type assignment functions that will be specified later add annotations to terms. In order to relate a fully annotated term to its underlying plain term, the function 'spine' removes all annotations from a term. For instance, the spine of `(s : nat -> nat)(0: nat) : nat` is `s(0)`.

[1] $\text{spine}(t : \tau) = \text{spine}(t)$
[2] $\text{spine}(f) = f$
[3] $\text{spine}(x) = x$
[4] $\text{spine}(\text{nil}) = \text{nil}$
[5] $\text{spine}(\text{top}) = \text{top}$
[6] $\text{spine}(t_1, t_2) = \text{spine}(t_1), \text{spine}(t_2)$
[7] $\text{spine}(t_1\ t_2) = \text{spine}(t_1)\ \text{spine}(t_2)$
[8] $\text{spine}(t_1 \times t_2) = \text{spine}(t_1) \times \text{spine}(t_2)$
[9] $\text{spine}(t_1 \rightarrow t_2) = \text{spine}(t_1) \rightarrow \text{spine}(t_2)$

The function 'fspine' is the same as 'spine' except that it does not remove the annotation from a function symbol.

[10] $\text{fspine}(f : \tau) = f : \tau$
[11] $\text{fspine}(t : \tau) = \text{fspine}(t)$ **otherwise**

The other equations are the same as for 'spine'. This function is used to translate annotated terms over a signature with overloading to disambiguated plain terms.

The 'term' of an annotated term is the term without its outermost type annotation. The 'type' of a term is its outermost annotation.

[12] $\text{term}(t : \tau) = t$
[13] $\text{type}(t : \tau) = \tau$

We see that for any term t of the form $t' : \tau$, $\text{term}(t) : \text{type}(t) = t$. To extend this property to arbitrary terms, the 'term' of a term without annotation is defined to be the term itself and the 'type' of a term without annotation is 'top'. To complete the picture it follows that a term with annotation 'top' is equal to the term itself.

[14] $\mathrm{term}(t) = t$ **otherwise**
[15] $\mathrm{type}(t) = \mathrm{top}$ **otherwise**
[16] $t : \mathrm{top} = t$

Now we have for arbitrary terms

[17] $\mathrm{term}(t) : \mathrm{type}(t) = t$

The functions 'dom' and 'cod' give the *domain* and *codomain* of a function type, respectively. The domain of a term that is not an arrow is nil, its codomain is the term itself. nil is a left unit for arrow. This corresponds to the notion that a constant is a function without arguments. Similarly the functions 'fun' and 'arg' give the *function* and *argument* of an application

[18]	$\mathrm{dom}(t_1 \to t_2) = t_1$		[19]	$\mathrm{arg}(t_1\ t_2) = t_2$	
[20]	$\mathrm{dom}(t) = \mathrm{nil}$	**otherwise**	[21]	$\mathrm{arg}(t) = \mathrm{nil}$	**otherwise**
[22]	$\mathrm{cod}(t_1 \to t_2) = t_2$		[23]	$\mathrm{fun}(t_1\ t_2) = t_1$	
[24]	$\mathrm{cod}(t) = t$	**otherwise**	[25]	$\mathrm{fun}(t) = t$	**otherwise**
[26]	$\mathrm{nil} \to t = t$		[27]	$t\,\mathrm{nil} = t$	

We have

[28] $\mathrm{dom}(t) \to \mathrm{cod}(t) = t$ [29] $\mathrm{fun}(t)\ \mathrm{arg}(t) = t$

The functions above are combined in the definition of the function 'bterm' that is used to analyze the types of binary functions. It strips the outermost annotation off an arrow term and off its domain.

[30] $\mathrm{bterm}(t) = \mathrm{term}(\mathrm{dom}(\mathrm{term}(t))) \to \mathrm{cod}(\mathrm{term}(t))$

For example,

```
bterm((((nat : type) # (nat : type)) : type
        -> (nat : type) : type))
= (nat : type) # (nat : type) -> (nat : type)
```

This function will be used for typechecking multi-level specifications. Similarly the function 'bapp' removes the annotations from a binary application

[31] $\mathrm{bapp}(t) = t_1\ (t_2, t_3)$
when $\mathrm{term}(t) = t_1'\ t_2',\ \mathrm{term}(t_1') = t_1,\ \mathrm{term}(t_2') = t_2, t_3$
[32] $\mathrm{bapp}(t) = t$ **otherwise**

For example,

```
bapp(((+) : nat # nat -> nat)
      ((0 : nat, 0 : nat) : nat # nat) : nat)
= (+)((0 : nat), (0 : nat))
```

12.2.4 Syntax of One-Level Specifications (OLS)

A specification consists of a signature and a list of equations. A signature is constructed from sort, function and variable declarations. We start with the definition of declarations.

Declarations A *function declaration* of the form $f : \tau$ assigns the type τ to function symbol f. For example, the type of the addition operator `plus` on natural numbers is declared as `plus : nat # nat -> nat`. An infix operator is declared by declaring its prefix notation as a binary function. For instance, if we use `+` as an infix operator for addition on natural numbers we would declare `(+) : nat # nat -> nat`. A *variable declaration* of the form $x : \tau$ assigns type τ to variable symbol x. For instance, the declaration `X : nat` declares a variable `X` of type `nat`. A *sort declaration* consists of a declaration of function symbols to be used as basic types.

module OLS
imports Terms$^{12.1.1}$ Types$^{12.2.2}$ Binary-Operators$^{B.2.1}$
exports
  sorts Decl Decls
  context-free syntax
    {Fun ","}+ ":" Term → Decl
    {Var ","}+ ":" Term → Decl
    {Decl ";"}* → Decls
    Decls "++" Decls → Decls {**right**}
  variables
    [*f*]"+"[*0-9′*]* → {Fun ","}+
    [*x*]"+"[*0-9′*]* → {Var ","}+
    "*d*"[*0-9′*]* → Decl
    "*d*""*"[*0-9′*]* → {Decl ";"}*
    "*d*""+"[*0-9′*]* → {Decl ";"}+
    "*ds*"[*0-9′*]* → Decls
equations
According to the syntax above, declarations can have the form $f_1, \ldots, f_n : \tau$ declaring in one declaration the function symbols f_i to be of type τ. This notation is merely an abbreviation of a list of declarations $f_i : \tau$ as expressed by the following equations.

[1] $$d_1^*;\, f_1^+, f_2^+ : \tau;\, d_2^* = d_1^*;\, f_1^+ : \tau;\, f_2^+ : \tau;\, d_2^*$$

[2] $$d_1^*;\, x_1^+, x_2^+ : \tau;\, d_2^* = d_1^*;\, x_1^+ : \tau;\, x_2^+ : \tau;\, d_2^*$$

[3] $$d_1^* +\!\!+\, d_2^* = d_1^*;\, d_2^*$$

Signatures An atomic signature is constructed from sort, function and variable declarations by the constructors 'sorts', 'functions' and 'variables', respectively. Signatures can be combined by the signature concatenation operator ';'. The projection functions 'S', 'F' and 'V' yield the list of sorts, function declarations, and variable declarations, respectively, of a signature.

exports
sorts Sig
context-free syntax

"sorts" Terms	$\rightarrow$ Sig	
"functions" Decls	$\rightarrow$ Sig	
"variables" Decls	$\rightarrow$ Sig	
	$\rightarrow$ Sig	
Sig ";" Sig	$\rightarrow$ Sig	{**right**}
"(" Sig ")"	$\rightarrow$ Sig	{**bracket**}
"S"(Sig)	$\rightarrow$ Terms	
"F"(Sig)	$\rightarrow$ Decls	
"V"(Sig)	$\rightarrow$ Decls	

variables
"Σ"[*0-9′*]∗ $\rightarrow$ Sig

equations

Equations [5], [7] and [9] express that atomic signatures with empty declaration lists are equivalent to empty signatures.

[4]	; Σ	=	Σ		[5]	sorts	=	
[6]	Σ;	=	Σ		[7]	functions	=	
[8]	$(\Sigma_1; \Sigma_2); \Sigma_3$	=	$\Sigma_1; \Sigma_2; \Sigma_3$		[9]	variables	=	
[10]	S(sorts *ts*)	=	*ts*		[11]	S()	=	
[12]	S(functions *ds*)	=			[13]	$S(\Sigma_1; \Sigma_2)$	=	$S(\Sigma_1) +\!\!+ S(\Sigma_2)$
[14]	S(variables *ds*)	=						
[15]	F(sorts *ts*)	=			[16]	F()	=	
[17]	F(functions *ds*)	=	*ds*		[18]	$F(\Sigma_1; \Sigma_2)$	=	$F(\Sigma_1) +\!\!+ F(\Sigma_2)$
[19]	F(variables *ds*)	=						
[20]	V(sorts *ts*)	=			[21]	V()	=	
[22]	V(functions *ds*)	=			[23]	$V(\Sigma_1; \Sigma_2)$	=	$V(\Sigma_1) +\!\!+ V(\Sigma_2)$
[24]	V(variables *ds*)	=	*ds*					

Specifications An atomic specification is a signature or a list of equations indicated by the functions 'signature' and 'equations', respectively. Specifications are combined by the operator ';'. The projection functions 'Sg' and 'E' give the signature and equations of a specification.

imports Equations[12.1.2]
exports
sorts Spec
context-free syntax

"signature" Sig	$\rightarrow$ Spec	
"equations" Eqs	$\rightarrow$ Spec	
	$\rightarrow$ Spec	
Spec ";" Spec	$\rightarrow$ Spec	{**right**}
"(" Spec ")"	$\rightarrow$ Spec	{**bracket**}
"Sg"(Spec)	$\rightarrow$ Sig	
"E"(Spec)	$\rightarrow$ Eqs	

variables
"$\mathcal{S}$"[*0-9'*]* → Spec
equations

[25] ; $\mathcal{S}$	$=$	$\mathcal{S}$	[26] signature	$=$	
[27] $\mathcal{S}$;	$=$	$\mathcal{S}$	[28] equations	$=$	
[29] $(\mathcal{S}_1; \mathcal{S}_2); \mathcal{S}_3$	$=$	$\mathcal{S}_1; \mathcal{S}_2; \mathcal{S}_3$			
[30] Sg(signature Σ)	$=$	Σ	[31] E(signature Σ)	$=$	
[32] Sg(equations $\mathcal{E}$)	$=$		[33] E(equations $\mathcal{E}$)	$=$	$\mathcal{E}$
[34] Sg()	$=$		[35] E()	$=$	
[36] Sg$(\mathcal{S}_1; \mathcal{S}_2)$	$=$	Sg$(\mathcal{S}_1)$; Sg$(\mathcal{S}_2)$	[37] E$(\mathcal{S}_1; \mathcal{S}_2)$	$=$	E$(\mathcal{S}_1)$ ++ E$(\mathcal{S}_2)$

We can extend the (Term ⇒ Term) functions to apply to all terms in a specification. By means of these functions we can apply the functions 'spine' and 'fspine' to a fully annotated specification in order to get its underlying plain specification. Accordingly, spine($\mathcal{S}$) denotes the underlying plain specification of specification $\mathcal{S}$.

12.2.5 Specification Semantics

The semantics of specifications is defined by means of an extension of equational logic to terms with type annotations.

Typed Equational Logic Equation [1] states that an equation $t_1 \equiv t_2$ is an axiom of a specification $\mathcal{S}$ if it is an element of the equations of $\mathcal{S}$. The other rules are the same as in the case of untyped equational logic (§12.1.3), except for the congruence rule for annotated terms [2]. Only terms with the same annotation can be equated if they are equal without annotation. Compare this to the congruence rules for application [6] and pairing [7] in §12.1.3, where both arguments can be equal modulo the equations in $\mathcal{E}$. In the case of multi-level specifications we will give an equational logic (§14.1.3) where equations over types play a role.

module OL-Equational-Logic
imports OLS$^{12.2.4}$ Substitution$^{B.2.7}$
exports
context-free syntax
Spec "⊢" Eq → Bool
equations

[1]
$$\frac{\mathrm{E}(\mathcal{S}) = \varphi_1^*;\ t_1 \equiv t_2;\ \varphi_2^*}{\mathcal{S} \vdash t_1 \equiv t_2 \ = \ \top}$$

[2]
$$\frac{\mathcal{S} \vdash t_1 \equiv t_2 = \top}{\mathcal{S} \vdash t_1 : \tau \equiv t_2 : \tau \ = \ \top}$$

The standard rules for reflexivity, symmetry, transitivity, substitution and congruence for the other binary operators are not shown.

Proposition 12.2.1 *Typed equational logic over a list of equations $\mathcal{E}$ is type preserving if the equations in $\mathcal{E}$ are type preserving, i.e., if for each $t_1 \equiv t_2 \in \mathcal{E}$,* $\text{type}(t_1) = \text{type}(t_2)$ *then $\mathcal{E} \vdash t \equiv t'$ implies* $\text{type}(t) = \text{type}(t')$.

Proof. (Sketch) by induction on derivations. The property clearly holds for [1], [2] and equality of types is preserved by reflexivity, symmetry, transitivity, substitution and congruence. □

Typed Term Rewriting In accordance with the rules for typed equational logic, the typed innermost term rewriting function 'eval' applies equations, oriented from left to right, until a term is in normal form. The annotation of a term is not evaluated in equation [2], because the equations of a specification apply only to object terms and not to types.

module OL-Term-Rewriting
imports OLS$^{12.2.4}$ Matching$^{B.2.8}$
exports
  context-free syntax
    eval "(" Spec ")" "⟦" Term "⟧" → Term
    eval "(" Spec ")" "⟦" Term "⟧" "_" Subst → Term
    step "(" Spec ")" "⟦" Term "⟧" → Term
equations

$$
\begin{array}{lrl}
[1] & \text{eval}(\mathcal{S})[\![t]\!] & = \text{eval}(\mathcal{S})[\![t]\!]_{[]} \\
[2] & \text{eval}(\mathcal{S})[\![t : \tau]\!]_\sigma & = \text{step}(\mathcal{S})[\![\text{eval}(\mathcal{S})[\![t]\!]_\sigma : \sigma(\tau)]\!] \\
[3] & \text{step}(\mathcal{S})[\![t]\!] & = \text{eval}(\mathcal{S})[\![t_2]\!]_\sigma \\
 & & \quad \textbf{when } \text{E}(\mathcal{S}) = \varphi_1^*; t_1 \equiv t_2; \varphi_2^*, \;\; t_1 := t = \sigma \\
[4] & \text{step}(\mathcal{S})[\![t]\!] & = t \quad \textbf{otherwise}
\end{array}
$$

The evaluation rules for the other operators are straightforward following Section 12.1.4. Note that recursive applications of eval to the other new operators product and arrow have to be added.

12.3 Typechecking One-Level Specifications

The context-free syntax of specifications defined in the previous section allows many degrees of freedom. In this section we narrow this down to the subset of one-level equational specifications with monomorphic types and no overloading. In §14.3 we will extend this to multi-level signatures with polymorphism and overloading. Here we avoid the complications introduced by multi-level specifications to make it easier to explain the architecture and basic ideas of the specification of the type system.

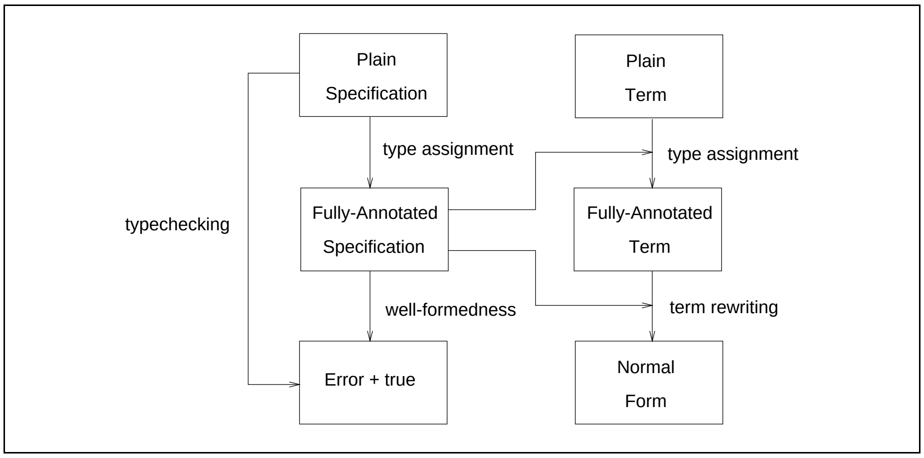


Figure 12.4: Architecture of typechecker for one-level specifications.

In §12.3.2 (Module OLS-WF) the well-formedness of fully annotated specifications is defined. The definition of well-formedness only specifies the correct cases, i.e., it contains a function which yields $\top$ iff the specification is well-formed. It does not deal with erroneous cases. The translation of these to human readable error messages is taken care of in §12.3.3 (Module OLS-NWF).

Since fully annotated specifications are difficult to read and write, one is not expected to actually write such specifications (although it is possible to supply partial annotations in terms to constrain their typing). Instead, plain specifications without annotations are annotated with types by a type assignment function defined in §12.3.4 (Module OLS-TA).

Finally, the typechecker defined in §12.3.5 (Module OLS-TC) first applies the type assignment function to a specification and then checks the result for well-formedness. This setup gives a separation between typechecking and type assignment that saves a great deal of bookkeeping and makes the definitions accessible. Moreover, annotated specifications can be used as input for tools other than a well-formedness checker, for instance a theorem prover or term rewriter. This architecture is illustrated in Figure 12.4.

First we define projection functions to find the type of a function or variable in a signature.

12.3.1 Projection

The projection function π yields the type of the first declaration for a variable or function in a list of declarations. The type of a function symbol f in a signature Σ is $\pi_f(\Sigma)$. The type of a variable symbol x in a signature Σ is $\pi_x(\Sigma)$. Observe that variable declarations in a 'functions' section and function declarations in a 'variables' section are ignored.

module Projection
imports $\text{OLS}^{12.2.4}$
exports
  context-free syntax
    "π" "_" Var "(" Decls ")" $\rightarrow$ Term
    "π" "_" Fun "(" Decls ")" $\rightarrow$ Term
    "π" "_" Var "(" Sig ")" $\rightarrow$ Term
    "π" "_" Fun "(" Sig ")" $\rightarrow$ Term
equations
Looking up a function in a list of declarations. If no declaration is found the term 'nil' is returned.

[1] $$\pi_f(d^*) = \text{nil} \quad \textbf{when} \;\; d^* =$$
[2] $$\pi_f(f : \tau;\ d^*) = \tau$$
[3] $$\pi_f(d;\ d^*) = \pi_f(d^*) \quad \textbf{otherwise}$$

The projection of a variable from a list of declarations is defined similarly.

[4] $$\pi_x(d^*) = \text{nil} \quad \textbf{when} \;\; d^* =$$
[5] $$\pi_x(x : \tau;\ d^*) = \tau$$
[6] $$\pi_x(d;\ d^*) = \pi_x(d^*) \quad \textbf{otherwise}$$

Looking up the type of a function in a signature consists of looking it up in the list of function declarations. The type of a variable is found by looking it up in the list of variable declarations.

[7] $$\pi_f(\Sigma) = \pi_f(\text{F}(\Sigma))$$
[8] $$\pi_x(\Sigma) = \pi_x(\text{V}(\Sigma))$$

12.3.2 Well-formedness

Well-formedness judgements on terms characterize the well-formed, fully annotated terms over a signature, i.e., given a signature Σ the set $\text{T}_{fa}(\Sigma)$ defined as

$$\text{T}_{fa}(\Sigma) = \{t \mid t \in \text{Term} \wedge \Sigma \vdash_{\text{term}} t\}$$

is the set of fully annotated terms t that satisfy the well-formedness judgement $\Sigma \vdash_{\text{term}} t$. The plain terms (without annotation) over a signature can be obtained by taking the spines of the well-formed, fully annotated, terms, i.e., the set $\text{T}(\Sigma)$ of plain terms over Σ defined as

$$\text{T}(\Sigma) = \{\text{spine}(t) \mid t \in \text{Term} \wedge \Sigma \vdash_{\text{term}} t\}$$

In this section we define well-formedness of fully annotated terms. In §12.3.4 we define a type assignment function that produces a fully annotated term for a plain term. Figure 12.5 shows a fully annotated specification.

```
signature
  sorts nat;
  functions
    0   : nat;
    s   : nat -> nat;
    (+) : nat # nat -> nat ;
  variables
    X : nat; Y : nat;
equations
  ((+) : nat # nat -> nat)
     ((0 : nat, X : nat) : nat # nat) : nat
  == X : nat;

  ((+) : nat # nat -> nat)
    (((s : nat -> nat) (X : nat) : nat, Y : nat)
    : nat # nat) : nat
  ==
  (s : nat -> nat) (((+) : nat # nat -> nat)
                     ((X : nat, Y : nat) : nat # nat) : nat)
                   : nat
```

Figure 12.5: A fully annotated one-level specification. This is an annotation of the specification in Figure 12.2.

We define not only the well-formedness of terms, but also the well-formedness of signatures and equations. In general, well-formedness judgements define which syntactically correct expressions are well-formed. The well-formedness of fully annotated one-level specifications is defined by means of the following judgements.

module OLS-WF
imports OLS[12.2.4] Projection[12.3.1] Variables[B.2.6] Error-Booleans[B.1.3]
Term-Analysis[12.2.3]
exports
context-free syntax
"$\vdash_{\mathrm{spec}}$" Spec $\rightarrow$ EBool
"$\vdash_{\mathrm{sig}}$" Sig $\rightarrow$ EBool
"$\vdash_{\mathrm{sorts}}$" Terms $\rightarrow$ EBool
Sig "$\vdash_{\mathrm{decls}}$" Decls $\rightarrow$ EBool
Sig "$\vdash_{\mathrm{sort}}$" Term $\rightarrow$ EBool
Sig "$\vdash_{\mathrm{term}}$" Term $\rightarrow$ EBool
Sig "$\vdash_{\mathrm{eqs}}$" Eqs $\rightarrow$ EBool

The well-formedness of a fully annotated specification $\mathcal{S}$ is defined by means of the judgement $\vdash_{\mathrm{spec}} \mathcal{S}$. It is defined in terms of several other judgements of the form $\Sigma \vdash_{\mathrm{r}} r$, which stands for 'construct r (of type r) is correct with respect to signature Σ'. For instance, the judgement $\Sigma \vdash_{\text{term}} t$ determines whether t is a well-formed term with respect to Σ. Equations defining judgements will, in general, have the form

$$\frac{C_1(q,\Sigma),\ldots,C_m(q,\Sigma)}{\Sigma \vdash_{\text{q}} q(r_1,\ldots,r_n) = \Sigma \vdash_{\text{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\text{r}_n} r_n}$$

to express that a construct q with subconstructs r_i is well-formed if conditions C_i hold for q and Σ and if the subconstructs are well-formed.

Judgements are functions that yield a term of the sort EBool of error Booleans. This sort is a version of the Booleans (defined in §B.1.3) with a constant ⊤ ('true' or 'correct') but with no constant for 'false' or 'incorrect'. Instead all elements of the sort Error act as values representing incorrectness. Two operations are defined on error Booleans. The symmetric conjunction ∧ yields ⊤ in case both arguments are ⊤ and yields the addition of the errors otherwise. The asymmetric conjunction ⇝ yields ⊤ if both arguments are ⊤ and otherwise it prefers the error of the left argument.

In this subsection only the positive cases for the judgements are defined. In the next subsection the other, negative, cases are defined to yield errors that give a description of the well-formedness rule that is violated.

equations

A specification is well-formed if its signature is well-formed and its equations are well-formed with respect to the signature.

[1] $\vdash_{\text{spec}} \mathcal{S} = \vdash_{\text{sig}} \Sigma \rightsquigarrow \Sigma \vdash_{\text{eqs}} \mathrm{E}(\mathcal{S}) \quad \textbf{when} \;\; \mathrm{Sg}(\mathcal{S}) = \Sigma$

Signatures A signature is well-formed if all its sort, function and variable declarations are well-formed.

[2] $\vdash_{\text{sig}} \Sigma = \vdash_{\text{sorts}} \mathrm{S}(\Sigma) \wedge \Sigma \vdash_{\text{decls}} \mathrm{F}(\Sigma) \wedge \Sigma \vdash_{\text{decls}} \mathrm{V}(\Sigma)$

The terms declared as sorts in the sorts section should be constant terms, i.e., function symbols.

[3] $\vdash_{\text{sorts}} f = \top$

[4] $\vdash_{\text{sorts}} \;= \top$

[5] $\vdash_{\text{sorts}} \tau_1^+;\, \tau_2^+ = \vdash_{\text{sorts}} \tau_1^+ \wedge \vdash_{\text{sorts}} \tau_2^+$

A declaration is correct if the type assigned to the function or variable is a well-formed sort (see below) and if the function or variable is not overloaded.

[6] $\Sigma \vdash_{\text{decls}} f : \tau = \Sigma \vdash_{\text{sort}} \tau \quad \textbf{when} \;\; \pi_f(\Sigma) = \tau$

[7] $\Sigma \vdash_{\text{decls}} x : \tau = \Sigma \vdash_{\text{sort}} \tau \quad \textbf{when} \;\; \pi_x(\Sigma) = \tau$

[8] $\Sigma \vdash_{\text{decls}} \;= \top$

[9] $\Sigma \vdash_{\text{decls}} d_1^+;\, d_2^+ = \Sigma \vdash_{\text{decls}} d_1^+ \wedge \Sigma \vdash_{\text{decls}} d_2^+$

Recall that $\pi_f(\Sigma)$ gives the type of f in Σ. The condition $\pi_f(\Sigma) = \tau$ for a declaration $f : \tau$ expresses that there should be only one declaration for f in the signature. If there are more (with different types), the condition will fail when checking the second declaration because $\pi_f(\Sigma)$ will yield the type of the first declaration.

Sorts Sorts are terms composed by $\times$ and $\rightarrow$ from function symbols, which are the basic sorts. A basic sort should be declared in the sorts section as expressed by the condition of equation [10].

[10] $$\Sigma \vdash_{\text{sort}} f = \top \quad \textbf{when} \;\; f \in \mathrm{S}(\Sigma) = \top$$

[11] $$\Sigma \vdash_{\text{sort}} t_1 \times t_2 = \Sigma \vdash_{\text{sort}} t_1 \wedge \Sigma \vdash_{\text{sort}} t_2$$

[12] $$\Sigma \vdash_{\text{sort}} t_1 \rightarrow t_2 = \Sigma \vdash_{\text{sort}} t_1 \wedge \Sigma \vdash_{\text{sort}} t_2$$

Terms A term is well-formed if all its subterms are annotated with a type in a correct way corresponding to the signature. Variables and functions are well-formed if their annotation is equal to their declared type in the signature and if that type is a well-formed sort. This additional condition is needed because $\pi_t(\Sigma)$ yields 'nil' if t is not declared. If the annotation is also 'nil', this would wrongly imply that the term is correct. Since 'nil' cannot be a sort, this extra condition is sufficient. A pair is well-formed if its type is the product of the types of its arguments. An application is well-formed if the its type is the codomain of the type of the function and if the type of the argument is equal to the domain of the type of the function.

[13] $$\frac{\pi_x(\Sigma) = \tau,\ \ \Sigma \vdash_{\text{sort}} \tau = \top}{\Sigma \vdash_{\text{term}} x : \tau \ = \ \top}$$

[14] $$\frac{\pi_f(\Sigma) = \tau,\ \ \Sigma \vdash_{\text{sort}} \tau = \top}{\Sigma \vdash_{\text{term}} f : \tau \ = \ \top}$$

[15] $$\frac{\mathrm{type}(t_1) \times \mathrm{type}(t_2) = \tau}{\Sigma \vdash_{\text{term}} (t_1, t_2) : \tau \ = \ \Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2}$$

[16] $$\frac{\mathrm{type}(t_1) = \mathrm{type}(t_2) \rightarrow \tau}{\Sigma \vdash_{\text{term}} t_1\ t_2 : \tau \ = \ \Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2}$$

There is no need to check the well-formedness of the types of applications and pairs, because equations [16] and [15] preserve well-formedness of type annotations. In equation [15]: if $\mathrm{type}(t_i)$ are well-formed, then their product is also well-formed. In equation [16]: if $\mathrm{type}(t_i)$ are well-formed, then τ must also be well-formed, because it is a subterm of $\mathrm{type}(t_1)$.

Equations An equation is well-formed if both sides have the same type and if all variables used in the right-hand side occur in the left-hand side. The last condition ensures that no new variables are introduced if the equations are interpreted as rewrite rules oriented from left to right.

[17] $$\frac{\mathrm{type}(t_1) = \mathrm{type}(t_2),\ \ \mathrm{vars}(t_2) \subseteq \mathrm{vars}(t_1) = \top}{\Sigma \vdash_{\text{eqs}} t_1 \equiv t_2 \ = \ \Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2}$$

[18] $$\Sigma \vdash_{\text{eqs}} \ = \ \top$$

[19] $$\Sigma \vdash_{\text{eqs}} \varphi_1^+ ; \varphi_2^+ \ = \ \Sigma \vdash_{\text{eqs}} \varphi_1^+ \wedge \Sigma \vdash_{\text{eqs}} \varphi_2^+$$

The following proposition states that a well-formed specification preserves types. This means that if two terms are equal according to a well-formed specification (and the rules of equational logic), they have the same type and that the normal form of a term has the same type as the term that is evaluated.

Proposition 12.3.1 (Type Soundness) *Equational logic and rewriting with well-formed specifications is type preserving: If* $\vdash_{\text{spec}} \mathcal{S}$ *and* $\mathrm{Sg}(\mathcal{S}) \vdash_{\text{term}} t_i$ *then* $\mathcal{S} \vdash t_1 \equiv t_2$ *implies* $\mathrm{type}(t_1) \equiv \mathrm{type}(t_2)$*. Furthermore,* $\mathrm{eval}(\mathcal{S})[\![t_1]\!] = t_2$ *implies* $\mathrm{type}(t_1) = \mathrm{type}(t_2)$.

Proof. By the well-formedness of $\mathcal{S}$ it follows that all equations are type preserving (equation [17]) and by Proposition 12.2.1 it then follows that equational derivations for $\mathcal{S}$ are type preserving. The second part of the proposition follows from the first part and the soundness of evaluation with respect to derivable equality (Proposition 12.1.1). □

The condition $\mathrm{Sg}(\mathcal{S}) \vdash_{\text{term}} t_i$ implies that the terms t_i are fully annotated. Normally, when considering a specification, we think about equality of plain terms. By means of the function spine and the well-formedness judgements we can characterize the plain terms $\mathrm{T}(\Sigma)$ over a signature (see beginning of this subsection). The following proposition states that well-formed specifications can only equate plain terms for which well-formed full annotations exist.

Proposition 12.3.2 *If* $\vdash_{\text{spec}} \mathcal{S}'$, $\mathcal{S} = \mathrm{spine}(\mathcal{S}')$, $t_1 \not\equiv t_2$ *and* $\mathcal{S} \vdash t_1 \equiv t_2$*, then there are* t_1' *and* t_2' *such that* $\mathrm{spine}(t_1') = t_1$, $\mathrm{spine}(t_2') = t_2$, $\mathcal{S}' \vdash_{\text{term}} t'_{\{1,2\}}$ *and* $\mathrm{type}(t_1') = \mathrm{type}(t_2')$.

12.3.3 Non-wellformedness

In the previous section we have defined which specifications are well-formed. In this section we look at the cases not covered by the well-formedness rules, which are, by definition, not well-formed. As an example of the type of error messages generated by these rules, Figure 12.6 shows the errors for an incorrect version of equations of the natural numbers specification from §12.2. We derive equations for the generation of error messages for non-wellformed specifications by looking at which cases were not covered by the equations above. If we had an equation of the form

$$\frac{C_1(q,\Sigma), \ldots, C_m(q,\Sigma)}{\Sigma \vdash_{\text{q}} q(r_1, \ldots, r_n) = \Sigma \vdash_{\text{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\text{r}_n} r_n}$$

the error case will be of the form

$$\frac{\neg C_1(q,\Sigma) \vee \cdots \vee \neg C_m(q,\Sigma)}{\Sigma \vdash_{\text{q}} q(r_1, \ldots, r_n) = \Sigma \vdash_{\text{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\text{r}_n} r_n \rightsquigarrow \texttt{"}q\texttt{" not well-formed}}$$

If either of the conditions does not hold then construct q is not well-formed. But we only want to report this fact if all its sub-constructs are well-formed. Otherwise only the reasons for non-wellformedness of the sub-constructs are reported,

```
equations
  0 + X    == Y;
  s(X) + Y == s + (X, Y)
```

(a)

```
equation  "(+)(0, X) == Y" not well-formed :
  variables "Y" of rhs not in lhs ;
application "(+)(s , X , Y )" not well-formed :
  type of argument "(nat -> nat) # nat # nat"
  does not match type of domain "nat # nat"
```

(b)

Figure 12.6: Non-wellformed specification (a) and errors (b) corresponding to the violations against the well-formedness rules. The signature part of the specification in (a) is not shown here but corresponds to the signature in Figure 12.2(a).

which is expressed by the use of the asymmetric conjunction $\leadsto$. Furthermore, we may choose to generate more precise error messages that are related to the conditions C_i. We then get equations of the form

$$\frac{C_1(q,\Sigma),\ldots,C_{i-1}(q,\Sigma),\neg C_i(q,\Sigma)}{\begin{array}{rl}\Sigma \vdash_{\mathrm{q}} q(r_1,\ldots,r_n) = & \Sigma \vdash_{\mathrm{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\mathrm{r}_n} r_n \\ & \leadsto \texttt{"}q\texttt{" does not have property } C_i\end{array}}$$

Instead of negating the conditions we can use default equations to deal with the remaining cases.

$$\begin{array}{rcl}\Sigma \vdash_{\mathrm{q}} q(r_1,\ldots,r_n) & = & \Sigma \vdash_{\mathrm{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\mathrm{r}_n} r_n \leadsto \texttt{"}q\texttt{" not well-formed} \\ & & \textbf{otherwise}\end{array}$$

module OLS-NWF
imports OLS-WF$^{12.3.2}$ SPEC-Errors$^{B.2.2}$
equations

Declarations No terms other than constants can be declared as sorts.

[1] $\vdash_{\mathrm{sorts}} \tau =$ " τ " not a well-formed sort declaration **otherwise**

[2] $\Sigma \vdash_{\mathrm{decls}} f : \tau =$ function " f " multiply declared **otherwise**
[3] $\Sigma \vdash_{\mathrm{decls}} x : \tau =$ variable " x " multiply declared **otherwise**

Sorts A term is a non-wellformed sort if it is a constant that is not declared or if it is a term that is not a constant, product or arrow.

[4] $\Sigma \vdash_{\text{sort}} f = \text{sort " } f \text{ " not declared} \quad \textbf{when } f \in \mathrm{S}(\Sigma) = \bot$
[5] $\Sigma \vdash_{\text{sort}} t = \text{" } t \text{ " not a well-formed sort} \quad \textbf{otherwise}$

Functions and Variables If the result of looking up a function or variable in the signature is 'nil', it is not declared, otherwise the declared sort is not well-formed.

[6] $\Sigma \vdash_{\text{term}} f : \tau = \text{function " } f \text{ " not declared} \quad \textbf{when } \pi_f(\Sigma) = \text{nil}$
[7] $\Sigma \vdash_{\text{term}} f : \tau = \Sigma \vdash_{\text{sort}} \tau \quad \textbf{otherwise}$
[8] $\Sigma \vdash_{\text{term}} x : \tau = \text{variable " } x \text{ " not declared} \quad \textbf{when } \pi_x(\Sigma) = \text{nil}$
[9] $\Sigma \vdash_{\text{term}} x : \tau = \Sigma \vdash_{\text{sort}} \tau \quad \textbf{otherwise}$

Pair and Application

[10] $\Sigma \vdash_{\text{term}} (t_1, t_2) : \tau = (\Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2)$
$\rightsquigarrow \text{pair " spine}(t_1, t_2) \text{ " not well-formed}$
otherwise

[11] $\Sigma \vdash_{\text{term}} t_1\ t_2 : \tau$
$= (\Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2)$
$\rightsquigarrow$ application " spine(t_1 t_2) " not well-formed
:: if eq(dom(type(t_1)), nil)
then " spine(t_1) " is not a function
else type of argument " type(t_2)
" does not match type of domain " dom(type(t_1)) "
otherwise

Note that the function 'spine' is used to show a term without its type annotations.

Annotation Terms without annotation or with double annotations are never well-formed.

[12] $$\frac{\text{type}(t) = \text{top}}{\Sigma \vdash_{\text{term}} t = \text{term " } t \text{ " not well-formed}}$$

[13] $\Sigma \vdash_{\text{term}} (t : \tau_1) : \tau_2$
$= \Sigma \vdash_{\text{term}} t : \tau_1$
$\rightsquigarrow$ annotation of " spine(t) " with " τ_2 " not well-formed
: should be " spine(τ_1) "

For several constructors in the language of terms we did not formulate any rules because they cannot be used at the level of terms at all.

[14] $\Sigma \vdash_{\text{term}} t_1 \times t_2 : \tau = \text{term " spine}(t_1 \times t_2) \text{ " not well-formed}$
[15] $\Sigma \vdash_{\text{term}} t_1 \to t_2 : \tau = \text{term " spine}(t_1 \to t_2) \text{ " not well-formed}$
[16] $\Sigma \vdash_{\text{term}} \text{top} : \tau = \text{term " top " not well-formed}$
[17] $\Sigma \vdash_{\text{term}} \text{nil} : \tau = \text{term " nil " not well-formed}$

Equations

[18] $\Sigma \vdash_{\text{eqs}} t_1 \equiv t_2$
 $= (\Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2)$
 $\rightsquigarrow$ equation " spine(t_1) $\equiv$ spine(t_2) " not well-formed
 :: if $\neg$ eq(type(t_1), type(t_2))
 then types do not match
 else variables " trms(vars(t_2) / vars(t_1)) " of rhs not in lhs
 otherwise

The following proposition states that the definition of the well-formedness judgement for terms is sufficiently-complete, i.e., all cases are covered by the well-formedness and non-wellformedness rules.

Proposition 12.3.3 *For any term t and signature Σ, $\Sigma \vdash_{\text{term}} t \in \{\top\} \cup \text{Error}$.*

12.3.4 Type Assignment

Figure 12.5 shows that it is a tedious task to write fully annotated specifications. In this subsection we define the type assignment function $\text{Wt}(\Sigma)[\![t]\!]$ that annotates a term with types according to signature Σ. Terms for which no typing exists are assigned the 'nil' type. If a term is already partially annotated, these annotations are checked against the derived annotations. In the one-level framework we are currently dealing with there is not much use for such annotations because terms can have at most one type. However, in the multi-level framework terms can be polymorphically typed and we will also allow functions to be overloaded. In such a situation, partial annotations are useful to enforce a more specific type for a term.

module OLS-TA
imports OLS[12.2.4] Projection[12.3.1] Term-Analysis[12.2.3]
exports
 context-free syntax
 "Wsp" "⟦" Spec "⟧" → Spec
 "Wt" "(" Sig ")" "⟦" Term "⟧" → Term
 "We" "(" Sig ")" "⟦" Eqs "⟧" → Eqs
equations
The function 'Wsp' assigns types to the terms in equations of a specification using its signature.

[1]
$$\frac{\Sigma = \text{Sg}(\mathcal{S})}{\text{Wsp}[\![\mathcal{S}]\!] \;=\; \text{signature } \Sigma;\ \text{equations We}(\Sigma)[\![\text{E}(\mathcal{S})]\!]}$$

Terms Functions and variables are annotated with their types in the signature. The type of a pair is the product of the types of its arguments. The type of an application is the codomain of the type of the function.

[2] $$\frac{\pi_x(\Sigma) = \tau}{\mathrm{Wt}(\Sigma)[\![x]\!] \;=\; x : \tau}$$

[3] $$\frac{\pi_f(\Sigma) = \tau}{\mathrm{Wt}(\Sigma)[\![f]\!] \;=\; f : \tau}$$

[4] $$\frac{\mathrm{Wt}(\Sigma)[\![t_1]\!] = t_3,\ \mathrm{Wt}(\Sigma)[\![t_2]\!] = t_4}{\mathrm{Wt}(\Sigma)[\![t_1, t_2]\!] \;=\; (t_3, t_4) : \mathrm{type}(t_3) \times \mathrm{type}(t_4)}$$

[5] $$\frac{\mathrm{Wt}(\Sigma)[\![t_1]\!] = t_3,\ \mathrm{Wt}(\Sigma)[\![t_2]\!] = t_4}{\mathrm{Wt}(\Sigma)[\![t_1\ t_2]\!] \;=\; t_3\ t_4 : \mathrm{cod}(\mathrm{type}(t_3))}$$

A term that is already partially annotated is handled by first assigning a type to the term without its annotation and then comparing the derived annotation with the given annotation.

[6] $$\frac{\mathrm{Wt}(\Sigma)[\![t]\!] = t'}{\mathrm{Wt}(\Sigma)[\![t : \tau]\!] \;=\; \text{if } \mathrm{eq}(\mathrm{type}(t'), \tau) \text{ then } t' \text{ else } t' : \tau}$$

In case the given type and the derived type are equal, the annotated term is returned. In case the types are different, the term was inconsistently annotated by the user. To be able to report this, the erroneous annotation is attached to the already annotated term. The resulting term is not well-formed, which will be reported by equation [13] in Section 12.3.3. Observe that equation [6] guarantees that we can assign types to fully annotated terms. We have that $\mathrm{Wt}(\Sigma)[\![\mathrm{Wt}(\Sigma)[\![t]\!]]\!] = \mathrm{Wt}(\Sigma)[\![t]\!]$, i.e., type assignment is idempotent.

Terms constructed from 'nil', 'top', '$\times$' and '$\rightarrow$' are assigned the error type 'nil', since these constructors cannot occur in well-formed terms.

[7] $$\mathrm{Wt}(\Sigma)[\![\mathrm{nil}]\!] = \mathrm{nil} : \mathrm{nil}$$
[8] $$\mathrm{Wt}(\Sigma)[\![\mathrm{top}]\!] = \mathrm{top} : \mathrm{nil}$$
[9] $$\mathrm{Wt}(\Sigma)[\![t_1 \times t_2]\!] = \mathrm{Wt}(\Sigma)[\![t_1]\!] \times \mathrm{Wt}(\Sigma)[\![t_2]\!] : \mathrm{nil}$$
[10] $$\mathrm{Wt}(\Sigma)[\![t_1 \rightarrow t_2]\!] = \mathrm{Wt}(\Sigma)[\![t_1]\!] \rightarrow \mathrm{Wt}(\Sigma)[\![t_2]\!] : \mathrm{nil}$$

Equations Both sides of an equation are assigned types.

[11] $$\mathrm{We}(\Sigma)[\![t_1 \equiv t_2]\!] = \mathrm{Wt}(\Sigma)[\![t_1]\!] \equiv \mathrm{Wt}(\Sigma)[\![t_2]\!]$$
[12] $$\mathrm{We}(\Sigma)[\![]\!] =$$
[13] $$\mathrm{We}(\Sigma)[\![\varphi_1^+; \varphi_2^+]\!] = \mathrm{We}(\Sigma)[\![\varphi_1^+]\!] +\!\!+\ \mathrm{We}(\Sigma)[\![\varphi_2^+]\!]$$

In §12.3.2 we saw that well-formedness judgements identify the fully annotated terms that are well-formed with respect to a signature. The type assignment function defined in this section allows us to produce fully annotated terms from plain terms. The following proposition states that for any plain term the type assignment function finds a well-formed, full annotation if one exists.

Proposition 12.3.4 (Correctness of 'Wt') *The type assignment function* Wt *finds a well-formed typing for a term if one exists, i.e., if t is a fully annotated term and* $\Sigma \vdash_{\mathrm{term}} t$ *then* $\mathrm{Wt}(\Sigma)[\![\mathrm{spine}(t)]\!] = t$.

Proof. by induction on terms. (Hint: equations [2] until [5] assign types to terms as required by [13] until [16] in §12.3.2.) □

12.3.5 Typechecking

typechecking can now be defined in terms of type assignment and well-formedness checking. We define three typecheck functions. The first checks a term against a signature, the second checks a list of equations against a signature and the last checks a complete specification. The functions are defined in terms of well-formedness judgements (§12.3.2) and type assignment functions (§12.3.4).

module OLS-TC
imports OLS-NWF$^{12.3.3}$ OLS-TA$^{12.3.4}$
exports
  context-free syntax
    tc "(" Sig ")" "⟦" Term "⟧" → EBool
    tc "(" Sig ")" "⟦" Eqs "⟧" → EBool
    tc "⟦" Spec "⟧" → EBool
equations

[1] $$\mathrm{tc}(\Sigma)[\![t]\!] = \vdash_{\mathrm{sig}} \Sigma \rightsquigarrow \Sigma \vdash_{\mathrm{term}} \mathrm{Wt}(\Sigma)[\![t]\!]$$
[2] $$\mathrm{tc}(\Sigma)[\![\mathcal{E}]\!] = \vdash_{\mathrm{sig}} \Sigma \rightsquigarrow \Sigma \vdash_{\mathrm{eqs}} \mathrm{We}(\Sigma)[\![\mathcal{E}]\!]$$
[3] $$\mathrm{tc}[\![\mathcal{S}]\!] = \vdash_{\mathrm{spec}} \mathrm{Wsp}[\![\mathcal{S}]\!]$$

Now we have seen the complete specification of a typechecker for a monomorphic applicative language. In Chapter 14 we will repeat this exercise for a multi-level polymorphic specification language.

13

Examples of Multi-Level Specifications

A multi-level specification consists of a list of numbered levels each of which is a specification as encounterd in the previous chapter. To structure multi-level specfications a module mechanism is provided to enable reuse of specifications. In this chapter we introduce the multi-level algebraic specification formalism MLS by means of a number of examples motivated by datatype specification.

13.1 Introduction

In the one-level framework of Chapter 12 the algebra of types used for the declaration of functions and variables is the subset of terms consisting of the closure under product ($\times$) and arrow ($\rightarrow$) of a set of sort constants. In such a framework one has higher-order functions (due to the applicative term format) but no polymorphism and no user-definable type constructors.

A *two-level specification* is a pair of specifications, called level 1 and level 0. The signature of the level 1 specification specifies a set of terms (like a one-level signature would) that are used at level 0 as types. In other words the signature at level 1 determines the type algebra of level 0. A type variable can be instantiated to any type. A term that has a type containing variables is polymorphic; it denotes all terms obtained by substituting ground types for type variables. As in the one-level case, the type algebra of signatures at level 1 is determined by the implicit signature generated from the sorts of level 1 and the constructors ($\rightarrow$) and ($\times$).

Multi-level specifications generalize two-level specifications by allowing arbitrary many levels of specifications. The signature at level n uses terms from the signature at level $n + 1$ as types and determines the type algebra of the signature at level $n - 1$. The types used in the highest level are members of the closure of the sorts at that level under ($\times$) and ($\rightarrow$), i.e., there is an implicit signature at the top that is generated by the sort declarations of the highest level. Figure 13.1 illustrates the concepts of one-level, two-level and multi-level specifications. The arrow from a signature means that the terms over that signature

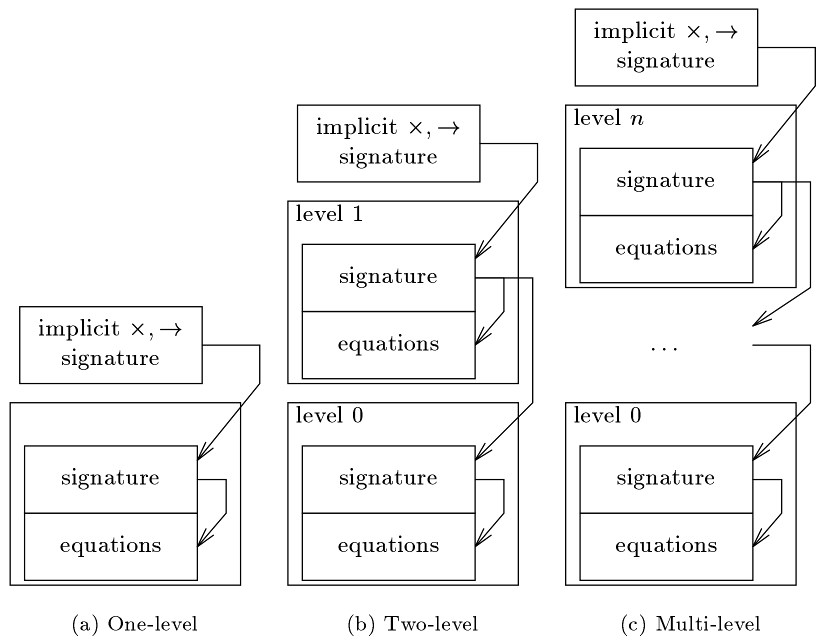


Figure 13.1: Specifications with one, two and multiple levels of signatures.

are used at the target of the arrow.

In Chapter 14 we define the extension of one-level specifications to multi-level specifications. In this chapter we start with an extensive list of examples that introduce the key ideas of the formalism. The examples are motivated by data type specification. For examples of application of multi-level specification to logical frameworks see (Hearn and Meinke, 1994) and (Hearn, 1995).

13.2 One Level

The running example of §12.2 and §12.3, successor naturals with addition, can be specified as a one-level specification. The declaration of sort `nat` generates the implicit sort signature consisting of the basic sort `nat` and the sort operators ($\rightarrow$) and ($\times$). As a consequence, terms like `nat`, `nat -> nat` and `nat # nat -> nat` are sorts that can be used in the signature that declares the functions composing the algebra of natural numbers. The signature is summarized in the diagram in Figure 13.2(a).

Figure 13.2: Signature diagrams of modules `nat` and `type`

```
module nat
level 0
  signature
    sorts nat;
    functions
      0   : nat;
      s   : nat -> nat;
      (+) : nat # nat -> nat;
    variables
      I, J : nat;
  equations
    0 + I    == I;
    s(I) + J == s(I + J);
```

13.3 Two Levels

Just like module `nat` defines a language of `nat` expressions, module `type` below defines a language of `type` expressions built from `type` variables `A`, `B` and `C` by means of the binary operators `(->)` and `(#)`. Examples of such terms are `A`, `A -> A`, `A # B -> A`, etc. These terms have type `type`. The signature of module `type` is summarized by the diagram in Figure 13.2(b).

```
module type
level 1
  signature
    sorts type;
    functions
      (#), (->)  : type # type -> type;
    variables
       A, B, C : type;
```

The difference between module `nat` and module `type` is that the signature of `types` is a level 1 signature. This entails that `type` expressions can be used as sorts at level 0 in signatures of modules that import module `type`.

13.3.1 Polymorphic Functions

The next module `function` introduces several polymorphic operations on functions. It first imports module `type` to use `type` expressions as sorts at level 0. The sorts declaration declares all expressions over level 1 that match the term `A`, as sorts. This means that all terms of type `type` can be used as sorts, but other terms over level 1 cannot (because `A` is a `type` variable). For instance, `A -> A` is a `type` expression, but `(->)`, which is also a term over level 1, is not a `type` expression. Next, the module defines a number of common functions. The *identity* function `i` takes any value to itself. The function `k` creates a *constant function* `k(X)` that always yields `X`. The function `s` is a *duplication* function that copies its third argument. The *composition* `G . H` of two functions `G` and `H` applies `G` to the result of applying `H` to the argument of the composition.

All these functions are *polymorphic*. The types of the functions contain `type` variables, which can be instantiated to arbitrary `type` expressions. The signature is actually an abbreviation of an infinite signature, declaring each function for each possible instantiation of the type variables. For instance, if `nat` is a `type` (as we will define in the next paragraph), then the instantiation `i : nat -> nat` is the identity function on the natural numbers.

```
module function
imports type;
level 0
  signature
    sorts A;
    functions
      i   : A -> A;
      k   : A -> B -> A;
      s   : (A -> B -> C) -> (A -> B) -> A -> C;
      (.) : (B -> C) # (A -> B) -> A -> C;
    variables
      X : A; Y : B; Z : C; G : A -> B; H : B -> C;
  equations
    i(X)         == X;
    k(X)(Y)      == X;
    s(X)(Y)(Z)   == X(Z)(Y(Z));
    (G . H)(X)   == G(H(X));
```

Observe that the specification in module `function` can also be considered as a logical framework in which the types are propositional logic formulas and the types of the functions the axioms of propositional logic, together with the implicit type of the application operator, which represents the modus ponens rule.

13.3.2 Typing Natural Numbers

In module `nat_typed`, the natural numbers as specified in module `nat` are incorporated in the world of `types` by declaring `nat` as a `type` constant. This

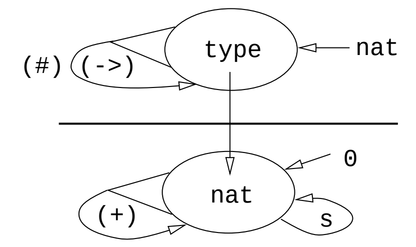


Figure 13.3: Signature diagram of module `nat-typed`.

is illustrated by the diagram in Figure 13.3. This provides the polymorphic functionality defined for arbitrary types to natural numbers.

```
module nat_typed
imports function, nat;
level 1
  signature
    functions
      nat : type;
```

13.4 Polymorphic Data Types

13.4.1 Cartesian Product

The product `A # B` of two types `A` and `B` is the type of pairs `(X, Y)` of elements `X` of `A` and `Y` of `B`. In MLS the pairing constructor function `_,_` is implicitly declared as `(,) : A # B -> A # B`. This means that if at level $n+1$ a declaration for `(#)` is given, then at level n the constructor `_,_` is defined implicitly. The declaration is implicit because binary infix operators are defined in terms of `_,_` by means of the equation $t_1 \oplus t_2 = (\oplus)(t_1, t_2)$. If `_,_` would be treated like an ordinary binary operator this would lead to a circular definition $t_1, t_2 = (,)(t_1, t_2) = (,)((,)(t_1, t_2))$

Module `product` defines a number of general functions on products. The projection functions `exl` and `exr` give the left and right elements of a pair. The product `G # H` of two functions is a function that applies the first function to the first argument of a pair and the second function to the second argument resulting in a new pair. The function `split` takes two functions that split the values of a type `C` into the components of a pair. For instance, the function `swap` defined as `(exr .split. exl)` swaps the elements of a pair, i.e., `(exr .split. exl)(X, Y) == (Y, X)`.[1] The function `curry` converts a binary function (a function on pairs) into a *curried* binary function that first takes its first argument and returns a function that when applied to a second argument

[1] Recall that `T1 .T2. T3` can be written as an abbreviation of `T2(T1,T2)`.

returns the application of the function to its arguments. The function `uncurry` is the inverse of `curry` that uncurries a function, i.e., converts it from a curried binary function to a function on pairs. It is defined in terms of duplication, projection and composition. Finally, the function `pair` is the curried version of the built-in pairing operator `(,)`.

```
module product
imports function;
level 0
  signature
    functions
      exl     : A # B -> A;
      exr     : A # B -> B;
      (#)     : (A -> B) # (A' -> B') -> (A # A') -> (B # B');
      split   : (C -> A) # (C -> B) -> C -> A # B;
      curry   : (A # B -> C) -> A -> B -> C;
      uncurry : (A -> B -> C) -> A # B -> C;
      pair    : A -> B -> A # B;
      swap    : A # B -> B # A;
    variables
      X : A; Y : B; Z : C; G : A -> B; H : B -> C;
  equations
    exl(X, Y)        == X;
    exr(X, Y)        == Y;
    (G # H)(X, Y)    == (G(X), H(Y));
    (G .split. H)(X) == (G(X), H(X));
    curry(G)(X)(Y)   == G(X, Y);
    uncurry(G)       == s(G . exl)(exr);
    pair             == curry(i);
    swap             == (exr .split. exl);
```

13.4.2 Disjoint Sum

The disjoint union or sum `A + B` of two types `A` and `B` contains all elements from `A` and `B`. The elements of both types are tagged by means of injection functions `inl` and `inr`, such that their original type can be reconstructed and such that there are no clashes; the union of `bool` and `bool` contains two elements, while the sum `bool + bool` contains the four elements `inl(t)`, `inl(f)`, `int(t)` and `inr(f)`. The sum `G + H` of two functions `G` and `H` is the function that takes the sum of codomains to the sum of the domains of `G` and `H` by applying `G` to left-tagged values and `H` to right-tagged values. The function `case` applies either of two functions with the same codomain depending on the tag of the value it is applied to.

The signature diagram in Figure 13.4 illustrates the structure of the algebra. Due to polymorphism, the number of sorts of a specification becomes infinite. Therefore, signature diagrams do not provide an accurate description

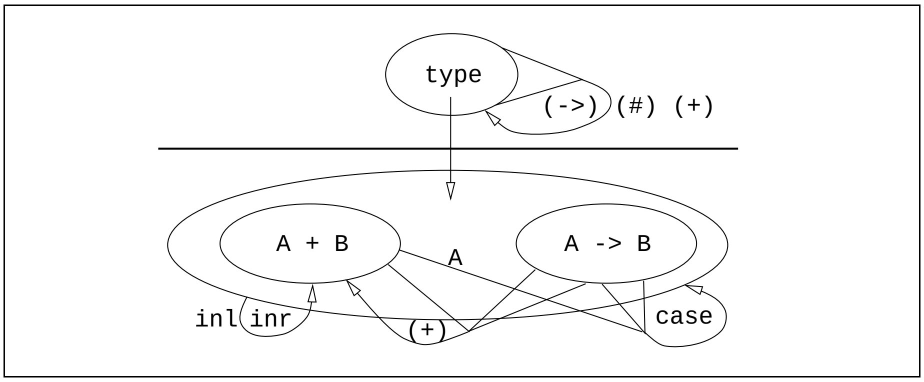


Figure 13.4: Signature diagram of module `sum`.

of the structure of the algebra described by a specification. Nonetheless we will continue to use approximate signature diagrams to give insight in the examples.

```
module sum
imports function;
level 1
  signature
    functions
      (+) : type # type -> type;
level 0
  signature
    functions
      inl  : A -> A + B;
      inr  : B -> A + B;
      (+)  : (A -> B) # (A' -> B') -> (A + A') -> (B + B')
      case : (A -> C) # (B -> C) -> (A + B) -> C;
  equations
    (G + H)(inl(X))      == inl(G(X));
    (G + H)(inr(Y))      == inr(H(Y));
    (G .case. H)(inl(X)) == G(X);
    (G .case. H)(inr(Y)) == H(Y);
```

13.4.3 Lists

A list is a structure built by the functions `[]`, the empty list, and `(::)` (cons) that adds an element to a list. A great number of generic functions have been defined on lists, see for instance (Bird, 1987, 1989). Here we give some common list functions. The function `(*)` (map) applies a function `G` to each element of a list. The function `(/)` (fold right) takes a pair `(G, Z)` of a function and a constant to replace the constructors `[]` and `(::)` such that `(X1 :: ... :: (Xn :: []))` is transformed into `(X1 .G. ... .G. (Xn .G. Z))`. The function `(\)` (fold left) is similar to `(/)` but starts adding the elements at the left side of the list resulting in `((Z .G. X1) .G. ... .G. Xn)`. The fold operations can be seen as signature morphisms consisting of replacements for the list cons function and the empty list. The function `(++)` concatenates the elements of two lists. The function `size` gives the length of a list. The functions `(++)` and `size` are defined in terms of the fold functions `(/)` and `(\)`. Finally, the function `zip` takes a pair of lists into a list of the pairs of the heads of the lists.[2]

```
module list
imports product, nat_typed;
level 1
  signature
    functions
      list : type -> type;
level 0
  signature
    functions
      []   : list(A);
      (::) : A # list(A) -> list(A);
      (*)  : (A -> B) # list(A) -> list(B);
      (/)  : (A # B -> B) # B -> list(A) -> B;
      (\)  : (B # A -> B) # B -> list(A) -> B;
      size : list(A) -> nat;
      (++) : list(A) # list(A) -> list(A);
      zip  : list(A) # list(B) -> list(A # B);
    variables
      L : list(A);
  equations
    G * []                == [];
    G * (X :: L)          == G(X) :: (G * L);

    (G / Z)([])           == Z;
    (G / Z)(X :: L)       == X .G. ((G / Z)(L));

    (G \ Z)([])           == Z;
    (G \ Z)(X :: L)       == (G \ (Z .G. X))(L);

    size                  == (s . exl) \ 0;
    L1 ++ L2              == ((::) / L2)(L1);

    zip([], L)            == [];
    zip(L, [])            == [];
    zip(X :: L, X' :: L') == (X, X') :: zip(L, L');
```

[2]Note that a variable declaration such as `L : list(A)` declares all variables with 'base' `L` as `list(A)` variables, e.g., `L1`, `L2` and `L'` are also declared by this declaration.

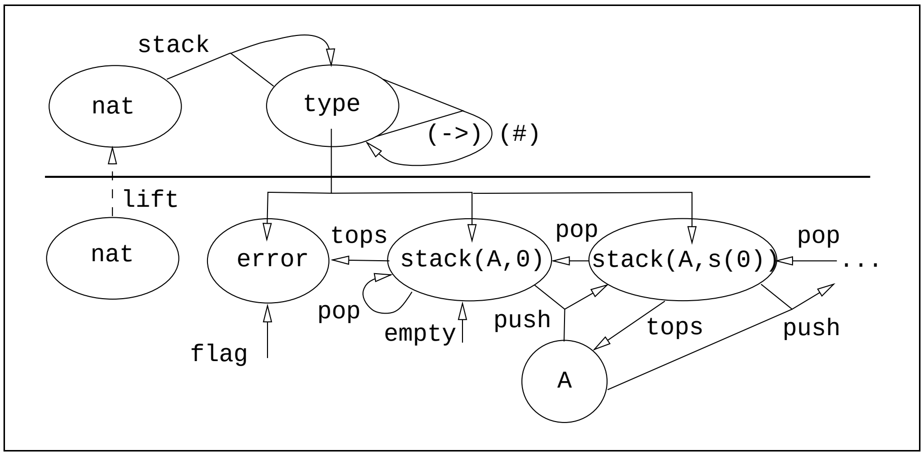


Figure 13.5: Signature diagram of module `stratified-stack`.

13.4.4 Stratified Stacks

All examples we have seen until now use only one sort (`type`) at level 1. The next module gives an example of a specification that uses an additional sort at level 1.

The data type of stacks can be specified by means of (polymorphic) push, pop and top functions. A well-known disadvantage of the normal formulation is that the top of the empty stack is either undefined or part of the type of stack elements, leading to a pollution of that type. All other operations that use the type have to take account of the top of the empty stack as an additional element. Another solution is to take a default value from the type of stack elements as result of the top of the empty stack. The problem of this solution is that the distinction between failure and success of a partial function is lost.

The solution of Hearn and Meinke (1994) is to 'stratify' the type of stacks. The stack type constructor does not just construct a type from a type, but has a natural number as argument that records the number of elements on the stack. The type operator `stack` takes a `type`, which is the type of the elements on the stack, and a `nat`, which represents the height of the stack. The type of stacks is stratified into stacks with elements of type `A` and height `0` indicated by the type `stack(A,0)`, stacks of height `s(0)` indicated by the type `stack(A,s(0))`, etc. A new `type` constant `error` is introduced to represent errors. The usual stack operators are now typed as follows. The `empty` stack has type `stack(A, 0)`, i.e., is a polymorphic constant for stacks with arbitrary types of elements and with height 0. The `push` function takes an `A` element and a stack of `A`'s with height `I` and produces a stack of `A`'s of height `s(I)`. The operations `pop` and `tops` come in two variants, one for empty stacks and one for non-empty stacks. The top of an empty stack (which has type `stack(A,0)`) results in an error element and not in an `A`. The error element is not added to the sort of stack elements.

The natural numbers in the types of stacks are used at level 1 while the specification in module `nat` specifies naturals at level 0. This means that just importing module `nat` is not enough to reuse the specification. The reuse is achieved by the operation *lift* that increases all levels of its argument specification by 1. The signature diagram in Figure 13.5 gives an overview of the signature in module `stratified-stack`.

```
module stratified-stack
imports types;
lift(imports nat);
level 1
  signature
    functions
      error : type;
      stack : type # nat -> type;
level 0
  signature
    functions
      flag  : error;
      empty : stack(A, 0);
      push  : A # stack(A, I) -> stack(A, s(I));
      pop   : stack(A, 0)     -> stack(A, 0);
      pop   : stack(A, s(I))  -> stack(A, I);
      tops  : stack(A, 0)     -> error;
      tops  : stack(A, s(I))  -> A;
    variables
      St : stack(A, I);
  equations
    pop(push(X,St))  == St;
    tops(push(X,St)) == X;
    pop(empty)       == empty;
    tops(empty)      == flag;
```

13.5 Three Levels

The type expressions we have used so far are described by a signature at the highest level of specifications. This entails that only type constructors over the signature $\{\mathtt{type}, (\rightarrow), (\times)\}$ can be constructed. This is not sufficient for all type constructors. For instance, the type of tuples contains a list of types. We can provide more structure in the sort space of types just as we provided more structure in the sort space of values, by building yet another level. Module `kind` introduces the sort `kind` at level 2 and defines `type` to be a `kind` constant.

```
module kind
imports type;
level 2
  signature
    sorts kind;
    functions
      type     : kind;
      (#), (->) : kind # kind -> kind;
    variables
      K : kind;
level 1
  signature
    sorts K : kind;
```

From here on we can proceed by adding useful kind constructors to level 2 and using them in the signatures at level 1. However, to construct tuples we need lists of types. Since there is not yet a definition of `list : kind -> kind`, we would have to redo module lists but now one level higher. Since this is a waste of time we use another approach. Module `type-type` also introduces the constant `type` at level 2, but uses `type` itself as its type! The `type`s defined in module `type` are used as kinds, by lifting the contents of that module. Now we can reuse all `type` constructors defined so far for level 1 at level 2, by simply lifting their specification.

```
module type-type
lift(imports type);
imports type;
level 2
  signature
    functions
      type : type;
```

13.5.1 Tuples

Lists and stacks are homogeneous data types that are parameterized by one sort. All elements of a list or stack are members of the same sort. A tuple on the other hand is a heterogeneous structure with various types of elements. In the next module we define a type constructor `prod` that constructs a generalized product type from a list of types. To construct a list of types we import the definition of level 0 lists and lift it to the level of types. Now we can use the same polymorphic operations on lists that we defined before. A tuple is constructed by means of the functions `<>` (empty tuple) and `(^)`, which adds an element to a tuple. (Recall from §12.1.1 that `<X1, ..., Xn>` is an abbreviation for `X1 ^ ... ^ Xn ^ <>`.) For instance, the tuple `<0, [0], t>` has type `prod([nat, list(nat), bool])`. The first element of a tuple is given by `exl` and the rest by `exr`. Observe that these functions are not partial: they are only well-formed if applied to a non-empty tuple. Now the size of a tuple can be determined by means of the function `size` that is polymorphic for all kinds of tuples.

```
module tuple
imports type-type, nat_typed;
lift(imports list);
level 1
  signature
    functions
      prod : list(type) -> type;
    variables
      LT : list(type);
level 0
  signature
    sorts prod(LT);
    functions
      <>   : prod([]);
      (^)  : A # prod(LT) -> prod(A :: LT);
      exl  : prod(A :: LT) -> A;
      exr  : prod(A :: LT) -> prod(LT);
      size : prod(LT) -> nat;
    variables
      P : prod(LT);
  equations
    exl(X ^ P)  == X;
    exr(X ^ P)  == P;
    size(<>)    == 0;
    size(X ^ P) == s(size(P))
```

13.6 Type Equations

In Hindley/Milner type systems it is not possible to construct the type of stratified stacks nor the type of tuples, because only one sort (`type`) can be used at the level of types. Since the formalism is uniform at all levels, the levels specifying type structure can contain equations over type expressions. This entails that operations on types can be specified. This makes the specification of advanced type constructs possible as will be illustrated in this section. This can be expressed in the MLS formalism, but is not supported by the MLS typechecker defined in Chapter 14.

13.6.1 Type Definitions

Abbreviations are a first application of type equations. A type can be defined as a mnemonic abbrevation of a complicated type. For instance, the following module defines the type of tables that associate keys with values as a list of pairs of keys and values. The function `(.)` is the lookup of a value associated with

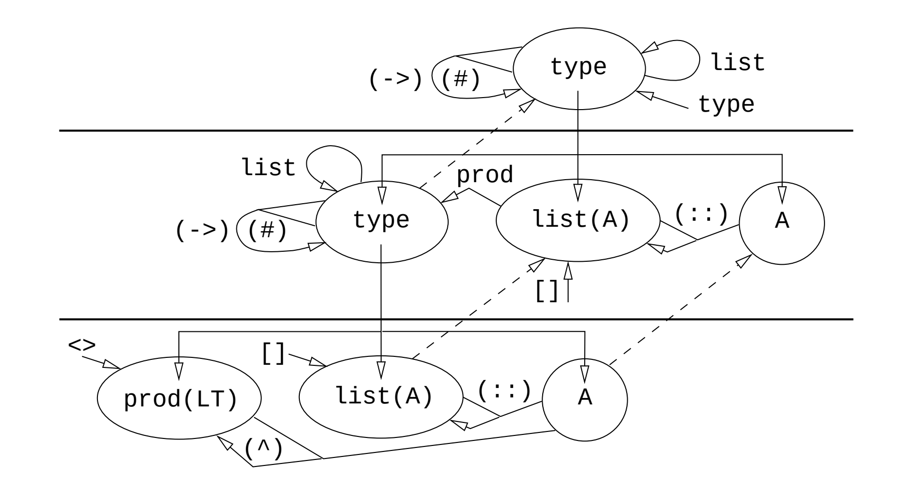


Figure 13.6: Signature diagram of module `tuple` with lifted `list` and `type`.

a key. Its second argument is a pair consisting of the key and a default value that is produced when the key is not defined in the table. The `(:=)` function updates the value of a key in a table. The functions `if` and `eq` are defined in module `bool`.

```
module table
imports list, bool;
level 1
  signature
    functions
      table : type # type -> type;
  equations
    table(A, B) == list(A # B);
level 0
  signature
    functions
      (.)  : table(A, B) # (A # B) -> B;
      (:=) : table(A, B) # (A # B) -> table(A, B);
  equations
    []            . (X, Y)    == Y;
    ((X, Y) :: L) . (X', Y')  == if(eq(X, X'))(Y)(L . X');
    []            := (X, Y)   == [(X, Y)];
    ((X, Y) :: L) := (X', Y') == if(eq(X, X'))
                                   (L := (X', Y'))
                                   ((X, Y) :: (L := (X', Y')));
```

This kind of type equations can be implemented by simply replacing each occurence of `table(T1, T2)` by its definition.

13.6.2 Tuple Zip

Another class of type equations is formed by operations on types. Consider the function `zip` that takes a pair of lists into a list of pairs. Its type can be expressed as

```
zip : list(A) # list(B) -> list(A # B)
```

where `list` is the type constructor for the type of lists, `#` is the type constructor for the type of pairs and `A` and `B` are type variables.

The generalization of this function takes an n-tuple of lists into a list of n-tuples. In an untyped language such as Lisp this can easily be encoded. Typed functional programming languages such as Haskell (Hudak *et al.*, 1992) cannot even define the type of tuples, let alone the type of the function `zip` for tuples. Instead for each n, upto the number that is thought necessary, a function `zip`n

```
zipn : list(A1) # ... # list(An) -> list(A1 # ... # An)
```

is defined together with its defining equations.

Using the tuples of the previous section, the type of the function zip for tuples can be expressed as

```
zip : prod(list * LT) -> list(prod(LT))
```

where `(*)` is the map function that a applies a function, in this case the type constructor `list`, to each element of a list. The map function is used to *extract* the types of list elements in the argument tuple to transfer it to the result tuple. The function `(*)` is a function that lives at level 1 (because of the lifting of module list) and is defined by means of the type equations

```
G * []      == []
G * X :: L == G(X) :: G * L
```

Well-formedness of an expression with `zip` involves the equality of types modulo these equations. For example, in the expression

```
zip(<[0, 2], [true, false], [[s(0)], [0, s(0)]]>)
```

the equality

```
zip : prod(list * [nat, bool, list(nat)])
       -> list(prod([nat, bool, list(nat)]))
=
zip : prod([list(nat), list(bool), list(list(nat))])
       -> list(prod([nat, bool, list(nat)]))
```

relates the type of the argument of `zip` to its declared type. The complete definition of `zip` for tuples is given in the following module.

```
module tuple-zip
imports tuple, list, bool;
level 0
  signature
    functions
      zip   : prod(list * LT) -> list(prod(LT));
      hds   : prod(list * LT) -> prod(LT);
      tls   : prod(list * LT) -> prod(list * LT);
      empty : prod(list * LT) -> bool;
  equations
    hds(<>)            == <>;
    hds((X :: L) ^ P)  == X ^ hds(P);

    tls(<>)            == <>;
    tls([]       ^ P)  == [] ^ tls(P);
    tls((X :: L) ^ P)  == L  ^ tls(P);

    empty(<>)          == false;
    empty([] ^ P)      == true;
    empty((X :: L) ^ P) == empty(P);

    zip(P) == if(empty(P))([])(hds(P) :: zip(tls(P)));
```

In order to reflect the use of type equations in the type assignment for specifications, $\mathcal{E}$-unification has to be applied. Given a set of equations $\mathcal{E}$ and two terms t_1 and t_2, an $\mathcal{E}$-unifier is a substitution σ such that $\mathcal{E} \vdash \sigma(t_1) \equiv \sigma(t_2)$. If the t_i are ground terms this question reduces to $\mathcal{E} \vdash t_1 \equiv t_2$. Here $\mathcal{E}$-unification has to be applied to unify the types of actual argument and domain type of the function `zip` given the equations for `(*)`. For instance, consider the type assignment of the term

```
zip(L ^ (M ^ (N ^ P)))
```

where `L`, `M` and `N` are list variables. The basic assignment to the components of this term are

```
zip : tuple(list * LT1) -> list(tuple(LT1))

(L ^ (M ^ (N ^ P)))
    : tuple(list(A) :: list(B) :: list(C) :: LT2)
```

Now we have to relate the domain of the function to the type of the argument, i.e., we have to solve the following unification problem:

```
tuple(list * LT1)
== tuple(list(A) :: list(B) :: list(C) :: LT2)
```

This means that we have to find a substitution such that these terms are equal considering the type equations for the map function:

```
G * []        == []
G * (A :: LT) == G(A) :: (G * LT)
```

A solution for the unification problem is the substitution

```
[LT1 := A :: B :: C :: LT3  LT2 := list * LT3]
```

This leads to the type assignment

```
zip(L ^ (M ^ (N ^ P))) : list(tuple(A :: B :: C :: LT3))
```

for the original term.

The $\mathcal{E}$-unification problem is undecidable in general (see Jouannaud and Kirchner (1991) for a survey of unification), but for the equations of functions like (*) it seems decidable. However, the type assignment function presented in §14.4.2 does not consider equations over types.

13.6.3 Tuple Functor

In a similar manner we can define the functor `prod` for tuples that takes a tuple of functions into a function on tuples such that the types in the domain tuple corresponds to the domains of the functions and the codomain corresponds to the codomains of the functions. This can be achieved by defining the function `(->)` on lists of types that zips together a list of domains and a list of codomains into a list of function types.

```
module tuple-map
imports tuple;
level 1
  signature
    functions
      (->) : list(type) # list(type) -> list(type);
  equations
    [] -> LT                  == [];
    LT -> []                  == [];
    (A :: LT1) -> (B :: LT2) == (A -> B) :: (LT1 -> LT2);
level 0
  signature
    functions
      prod : prod(LT1 -> LT2) -> prod(LT1) -> prod(LT2);
  equations
    prod(<>)([])        == [];
    prod(G ^ P)(X ^ P') == G(X) ^ prod(P)(P');
```

13.6.4 Tuple Composition

Another well-known problem is the construction of the composition of a tuple of functions into a new function. For instance, given three functions `F`, `G` and `H`, the composition of the tuple `<F, G, H>` is defined such that

```
comp(<F, G, H>)(X) == H(G(F(X)))
```

The type of this function can be specified by means of a function `comp` at level 1 that transforms a list of types into a list of function types such that the codomain of a function type is the domain of the next function type in the list. The function `last` is used to extract the last codomain from the list of types

```
module tuple-composition
level 1
  signature
    functions
      comp : list(type) -> list(type);
      last : type # list(type) -> type;
    equations
      comp([])          == [];
      comp([A])         == [];
      comp(A :: B :: LT) == (A -> B) :: comp(B :: LT);
      last(A, [])       == A;
      last(A, B :: LT)  == last(B, LT);
level 0
  signature
    functions
      comp : prod([])                 -> A -> A;
      comp : prod(comp(A :: B :: LT)) -> A -> last(B, LT);
  equations
    comp(<>)(X)    == X;
    comp(G ^ P)(X) == comp(P)(G(X));
```

13.6.5 Recursive Types

The type definitions of the previous examples can be eliminated by simply applying the equations as rewrite rule. The following example uses recursive type equations to define the type of binary trees in terms of disjoint sums and Cartesian products. The abstract data type is defined by three functions that are expressed in terms of the primitives on sums and products. The characteristic functions of sums are the left and right injection functions `inl` and `inr` and the selection function `case`. The functions for trees are `leaf` that constructs a leaf node from some value, `(+)` that combines to trees into a new tree and `cata` that builds a value from a tree by replacing its constructor functions by new functions.

```
module rec_tree
imports sum, product, bool, nat;
level 1
  signature
    functions
      tree : type -> type;
  equations
    tree(A) == A + (tree(A) # tree(A));
level 0
  signature
    functions
      leaf : A -> tree(A);
      (+)  : tree(A) # tree(A) -> tree(A);
      cata : (A -> B) # (B # B -> B) -> tree(A) -> B;
  equations
    leaf(X) == inl(X);
    T1 + T2 == inr(T1, T2);

    cata(Zero, Plus) == Zero .case.
                        (Plus . (cata(Zero, Plus).exl #
                                 cata(Zero, Plus).exr)
```

It is clear that the definition of `tree` can not be eliminated by simple rewriting, since this leads to an infinite term.

13.6.6 Type Classes

Another class of applications of equations over types is formed by constraints on types. The following module models the restriction of the polymorphism of the equality function by means of a type class like mechanism. The module imports module `bool` that defines the standard operations on the Boolean values `t` (true) and `f` (false). At level 1 a unary boolean function (a predicate) `eq` on types is defined such that the type `nat` is in the `eq` class and such that a list type is in the `eq` class if its content type is in the class. The operator `(=>)` constrains a type by some boolean condition. At level 0 the equality function `eq` is now declared with type `eq(A) => (A # A -> bool)`, which expresses that the function only applies to types in the `eq` class. The function `(!)` is used to apply a function with a constrained type to an argument. It requires that the condition is equal to `t`. This ensures that `eq` cannot be applied to function types or other types not in the `eq` class.

```
module equality
imports type;
lift(imports bool);
imports list, nat_typed, bool_typed;
level 1
  signature
    functions
      eq   : type -> bool;
      (=>) : bool # type -> type;
   equations
     eq(nat)     == t;
     eq(list(A)) == eq(A);
level 0
  signature
    functions
      (!) : (t => A -> B) # A -> B;
      eq  : eq(A) => (A # A -> bool);
  equations
    eq!(0, 0)       == t;
    eq!(0, s(I))    == f;
    eq!(s(I), 0)    == f;
    eq!(s(I), s(J)) == eq!(I, J);

    eq!([], [])          == t;
    eq!(X :: L, [])      == f;
    eq!([], X :: L)      == f;
    eq!(X :: L, X' :: L') == eq!(X, X') /\ eq!(L, L');
```

Here we conclude our introduction to MLS. In the next chapter we proceed to formalize the MLS language.

14

Definition of Multi-Level Specifications

In this chapter we give a formal specification of the syntax and semantics of multi-level specifications, including modular specifications and specification operators such as 'lift'. Typechecking of multi-level specifications differs at several points from typechecking one-level specifications. First of all, types at level n are terms over the signature at level $n + 1$. Secondly, types can be polymorphic. Finally, functions and variables can be overloaded, i.e., have more than one declaration in a signature.

14.1 Syntax and Equational Logic

In this section we define the syntax of multi-level specifications and define the semantics of multi-level specifications.

14.1.1 Syntax

A multi-level specification is either empty, a level composed of a natural number indicating the level and a specification, or a concatenation of multi-level specifications.

module MLS
imports OLS$^{12.2.4}$ Naturals$^{B.1.4}$
exports
  sorts MLS
  context-free syntax
    $\rightarrow$ MLS
    "level" Nat Spec $\rightarrow$ MLS
    MLS ";" MLS $\rightarrow$ MLS {**left**}
    "(" MLS ")" $\rightarrow$ MLS {**bracket**}
  priorities
    Sig ";" Sig $\rightarrow$ Sig > MLS ";" MLS $\rightarrow$ MLS
  variables
    "Γ" [*0-9'*]* $\rightarrow$ MLS

Arrow and Product Functions Since types are terms over a signature, the constructors arrow and product must also be declarable. For this purpose the functions $(\rightarrow)$ and $(\times)$ are introduced with the same notation as used to make other infix operators into prefix functions.

exports
context-free syntax
"$(\rightarrow)$" $\rightarrow$ Fun
"$(\times)$" $\rightarrow$ Fun

Specification Projections As for the OLS case we define several projection functions for decomposing specifications. Most noteworthy is the function 'up' that gives a specification without its lowest level. The projection function π_n gives the specification at level n. The function 'lift' increases the level indicators of all levels by 1. The function 'drop' decreases the indicators of all levels by one and removes the specification at level 0.

exports
context-free syntax

"π" "_" Nat "(" MLS ")"	$\rightarrow$ Spec
max(MLS)	$\rightarrow$ Nat
lift(MLS)	$\rightarrow$ MLS
drop(MLS)	$\rightarrow$ MLS
up(MLS)	$\rightarrow$ MLS
top-sig	$\rightarrow$ MLS
decl(Terms, Term)	$\rightarrow$ Decls

equations
Level concatenation is normalized to a left-associative list. An empty level is equivalent to an empty specification. The empty specification is a unit for composition.

[1] $\Gamma_1; (\Gamma_2; \Gamma_3) = \Gamma_1; \Gamma_2; \Gamma_3$ [2] $; \Gamma = \Gamma$
[3] level n $=$ [4] $\Gamma; = \Gamma$

The projection π_n gives the n-th level of a specification.

[5] $\pi_n() =$
[6] $\pi_n(\text{level } n\ \mathcal{S}) = \mathcal{S}$
[7] $\pi_n(\text{level } m\ \mathcal{S}) =$ **when** $\text{eq}(n, m) = \bot$
[8] $\pi_n(\Gamma_1; \Gamma_2) = \pi_n(\Gamma_1); \pi_n(\Gamma_2)$

The function 'max' gives the index of the highest level of a specification. Note that 'max' is also the maximum function on natural numbers.

[9] $\max() = 0$
[10] $\max(\text{level } n\ \mathcal{S}) = n$ **when** $\mathcal{S} \neq$
[11] $\max(\Gamma_1; \Gamma_2) = \max(\max(\Gamma_1), \max(\Gamma_2))$

Any specification is equal (modulo commutativity of ';') to the concatenation of all levels, i.e., for any specification Γ:

$$\Gamma = \text{level max}(\Gamma)\ \pi_{\max(\Gamma)}(\Gamma); \ldots; \text{level 1}\ \pi_1(\Gamma); \text{level 0}\ \pi_0(\Gamma)$$

The function 'lift' increments all levels by one.

[12] $\text{lift}() =$

[13] $\text{lift}(\text{level } n\ \mathcal{S}) = \text{level succ}(n)\ \mathcal{S}$

[14] $\text{lift}(\Gamma_1;\ \Gamma_2) = \text{lift}(\Gamma_1);\ \text{lift}(\Gamma_2)$

The function 'drop' lowers all levels by one level and drops the lowest level.

[15] $\text{drop}() =$

[16] $\text{drop}(\text{level } 0\ \mathcal{S}) =$

[17] $\text{drop}(\text{level } n\ \mathcal{S}) = \text{level pred}(n)\ \mathcal{S} \quad \textbf{when} \ \ \text{zero}(n) = \bot$

[18] $\text{drop}(\Gamma_1;\ \Gamma_2) = \text{drop}(\Gamma_1);\ \text{drop}(\Gamma_2)$

For π, 'lift' and 'drop' we have (modulo associativity and commutativity of ';')

[19] $\text{lift}(\text{drop}(\Gamma));\ \text{level } 0\ \pi_0(\Gamma) = \Gamma$

A multi-level specification can be considered as a stack of specifications, with 'drop' as the pop operation and π_0 as top. The term lift(_); level 0_ corresponds to pushing a specification on the stack.

The constant 'top-sig' is the implicit signature that determines the sorts of the highest signature.

[20] $\text{top-sig} \ = \ \text{level 0 signature functions } (\times), (\rightarrow) : \text{top} \times \text{top} \rightarrow \text{top}$

The operation 'up' is like 'drop' with an extra property. In case level 0 is not the highest level, i.e., max is not equal to 0, then 'up' just drops level 0. If level 0 is the highest level, 'up' is the signature 'top-sig' extended with the sorts of the highest level declared as constants of type 'top'. This is the implicit signature of the types used at the highest level of a specification. Observe that if $\max(\Gamma) = 0$, then after one iteration $\text{up}(\text{up}(\Gamma)) = \text{up}(\Gamma)$.

[21] $\text{up}(\Gamma) = \text{top-sig; level 0 signature functions decl}(\text{S}(\text{Sg}(\pi_0(\Gamma))), \text{top})$
$\qquad \textbf{when} \ \ \text{zero}(\max(\Gamma)) = \top$

[22] $\text{up}(\Gamma) = \text{drop}(\Gamma) \quad \textbf{when} \ \ \text{zero}(\max(\Gamma)) = \bot$

The function 'decl' constructs a list of declarations from a list of terms and a sort. It is used in the definition of 'up' above to create a declaration for each sort of the highest level. Only the function constants in the list are declared.

[23] $\text{decl}(, \tau) =$

[24] $\text{decl}(f, \tau) = f : \tau$

[25] $\text{decl}(t_1^+;\ t_2^+, \tau) = \text{decl}(t_1^+, \tau) \mathbin{+\!\!+} \text{decl}(t_2^+, \tau)$

[26] $\text{decl}(t, \tau) = \quad \textbf{otherwise}$

14.1.2 Normalization

According to the syntax of signatures and multi-level specifications, specification elements like levels, signatures and declarations can be written in any order and can be repeated. For instance, a specification can contain several sections for `level 0` and a signature can contain several `functions` sections. The function 'norm' below normalizes a specification such that the levels are presented in decreasing order and specifications consists of one signature section and one equations section. Furthermore, signatures are normalized such that they contain a single sorts, functions and variables section. Finally, redundant declarations, sort declarations and equations are removed.

module MLS-Norm
imports MLS[14.1.1]
exports
  context-free syntax
    norm(MLS) $\rightarrow$ MLS
    norm(Spec) $\rightarrow$ Spec
equations

$$[1] \quad \text{norm}(\Gamma) = \text{level } 0 \text{ norm}(\pi_0(\Gamma)) \quad \textbf{when } \max(\Gamma) = 0$$

$$[2] \quad \text{norm}(\Gamma) = \text{lift}(\text{norm}(\text{drop}(\Gamma))); \text{level } 0 \text{ norm}(\pi_0(\Gamma)) \quad \textbf{when } \text{zero}(\max(\Gamma)) = \perp$$

$$[3] \quad \text{norm}(\mathcal{S}) = \text{signature } \Sigma'; \text{equations } \text{E}(\mathcal{S}) \quad \textbf{when } \text{Sg}(\mathcal{S}) = \Sigma, \; \Sigma' = \text{sorts } \text{S}(\Sigma); \text{functions } \text{F}(\Sigma); \text{variables } \text{V}(\Sigma);$$

$$[4] \quad d_1^*;\ d;\ d_2^*;\ d;\ d_3^* = d_1^*;\ d;\ d_2^*;\ d_3^*$$

$$[5] \quad \text{sorts } t_1^*;\ t;\ t_2^*;\ t;\ t_3^* = \text{sorts } t_1^*;\ t;\ t_2^*;\ t_3^*$$

$$[6] \quad \text{equations } \varphi_1^*;\ \varphi;\ \varphi_2^*;\ \varphi;\ \varphi_3^* = \text{equations } \varphi_1^*;\ \varphi;\ \varphi_2^*;\ \varphi_3^*$$

14.1.3 Multi-Level Equational Logic

We redefine equational logic for multi-level specifications. An equation is an axiom if it is an equation at level 0. The equations at higher levels apply to type annotations; in equation [2] it is stated that two annotated terms are equal if their term parts are equal and if the annotations are equal with respect to the next level.

module ML-Equational-Logic
imports $\text{MLS}^{14.1.1}$ $\text{Substitution}^{B.2.7}$
exports
  context-free syntax
    MLS "⊢" Eq → Bool
equations

$$[1] \qquad \frac{\mathrm{E}(\pi_0(\Gamma)) = \varphi_1^*;\ t_1 \equiv t_2;\ \varphi_2^*}{\Gamma \vdash t_1 \equiv t_2\ =\ \top}$$

$$[2] \qquad \frac{\Gamma \vdash t_1 \equiv t_2 = \top,\ \mathrm{up}(\Gamma) \vdash \tau_1 \equiv \tau_2 = \top}{\Gamma \vdash t_1 : \tau_1 \equiv t_2 : \tau_2\ =\ \top}$$

The standard rules for reflexivity, symmetry, transitivity, substitution and congruence for the other binary term operators (application, pair, arrow and product) are not shown.

If only free constructors (functions over which no equations are defined) are used in type annotations, then the types τ_i in equation [2] have to be syntactically equal. In that case multi-level equational logic reduces to the typed equational logic of §12.2.5 and we have

$$\Gamma \vdash t_1 \equiv t_2 = \pi_0(\Gamma) \vdash t_1 \equiv t_2$$

Under the same assumption, term rewriting with a multi-level specification reduces to the typed term rewriting of §12.2.5. Rewriting of annotated terms in a system with type equations is more complicated because $\mathcal{E}$-matching is needed. Given a set of equations $\mathcal{E}$, term t_1 $\mathcal{E}$-matches term t_2 if there exists a substitution σ such that $\mathcal{E} \vdash \sigma(t_2) \equiv t_1$.

Meinke (1992a) gives an equational logic for two levels of equations similar to the multi-level equational logic above. Meinke (1993) considers the rewrite relation resulting from a set of equations over terms and types by taking the transitive, reflexive closure of the equations considered as rewrite rules in both directions.

14.2 Modular Specifications

We define a simple modularization scheme based on syntactic inclusion. It adds considerably to the expressive power of the language by the ability to share specifications at more than one level, as we saw in the examples in §13.1. A module binds a multi-level specification to a module name. An import is a reference to the body of a module. It denotes the specification that would be obtained by replacing the import by the module body. Name clashes between functions imported from different modules are not problematic, because overloading permits such functions to coexist. Functions from different origins with identical names *and* types are identified. Although this seems a reasonable choice, extension with renaming operators would be useful, but is not further considered here.

module MMLS
imports MLS$^{14.1.1}$ MLS-Norm$^{14.1.2}$
exports
  sorts Module Modules
  context-free syntax
    "imports" {Fun ","}* → MLS
    "module" Fun MLS ";" → Module
    Module* → Modules
    Modules "++" Modules → Modules {**right**}
    π "_" Fun "(" Modules ")" → MLS
  variables
    "M"[*0-9'*]* → Module
    "M""*"[*0-9'*]* → Module*
    "M""+"[*0-9'*]* → Module+
equations
Concatenation of module lists

$$[1] \qquad M_1^* \mathbin{+\!\!+} M_2^* = M_1^*\ M_2^*$$

A list of imports denotes the concatenation of the imported specifications.

$$[2] \qquad \text{imports } f_1^+, f_2^+ = \text{imports } f_1^+; \text{imports } f_2^+$$

The projection of a module name in a list of modules yields the module body. If more than one module with the same name exists, the bodies are concatenated.

$$[3] \qquad \pi_f() =$$
$$[4] \qquad \pi_f(\text{module } f\,\Gamma;) = \Gamma$$
$$[5] \qquad \pi_f(\text{module } f'\ \Gamma;) = \quad \textbf{when } \text{eq}(f, f') = \bot$$
$$[6] \qquad \pi_f(M_1^+\ M_2^+) = \pi_f(M_1^+); \pi_f(M_2^+)$$

Note that the function π_f is overloaded: lookup of the type of a function in a list of declarations and lookup of a module in a list of modules.

Modules have a simple syntactic replacement semantics. The normalization function 'flat' flattens all modules in a list of modules, by replacing imports by module bodies.

imports Term-Sets$^{B.2.5}$
exports
  context-free syntax
    flat "(" Modules ")" → Modules
    flat "(" Modules ")" "⟦" Modules "⟧" → Modules
    flat "(" Modules "," TermSet ")" "⟦" MLS "⟧" → MLS
equations
The unary function 'flat', flattens the body of each module in a list of modules with respect to the entire list of modules.

$$[7] \qquad \text{flat}(M^*) = \text{flat}(M^*)[\![M^*]\!]$$

$$
\begin{array}{lrl}
[8] & \text{flat}(M^*)[\![\,]\!] = & \\
[9] & \text{flat}(M^*)[\![M_1^+\ M_2^+]\!] = & \text{flat}(M^*)[\![M_1^+]\!] +\!\!+ \text{flat}(M^*)[\![M_2^+]\!] \\
[10] & \text{flat}(M^*)[\![\text{module}\ f\,\Gamma;]\!] = & \text{module}\ f\,\text{norm}(\text{flat}(M^*, \{\})[\![\Gamma]\!]);
\end{array}
$$

An import of a module is replaced by its body. The imports in the body of a module have to be flattened in turn. A loop caused by cyclic imports is prevented by adding the module name to the set of modules already seen (the second argument of function 'flat'). An import is not expanded if a module was already imported (equation [12]).

$$
[11] \qquad \frac{f \in \Phi = \bot}{\text{flat}(M^*, \Phi)[\![\text{imports}\ f]\!] \ = \ \text{flat}(M^*, \Phi \cup \{f\})[\![\pi_f(M^*)]\!]}
$$

$$
[12] \qquad \frac{f \in \Phi = \top}{\text{flat}(M^*, \Phi)[\![\text{imports}\ f]\!] \ = \ }
$$

Imports inside other constructs are replaced by distributing 'flat' over all operators except 'imports'. Consider, for example, the expression `lift(imports list)` in §13.5.1. Since 'lift' is not defined on imports, the imported module has to be substituted before lifting can be performed.

$$
\begin{array}{lrl}
[13] & \text{flat}(M^*, \Phi)[\![\Gamma]\!] = & \quad \textbf{when}\ \Gamma = \\
[14] & \text{flat}(M^*, \Phi)[\![\Gamma_1; \Gamma_2]\!] = & \text{flat}(M^*, \Phi)[\![\Gamma_1]\!]; \text{flat}(M^*, \Phi)[\![\Gamma_2]\!] \\
[15] & \text{flat}(M^*, \Phi)[\![\text{level}\ n\ \mathcal{S}]\!] = & \text{level}\ n\ \text{flat}(M^*, \Phi)[\![\mathcal{S}]\!] \\
[16] & \text{flat}(M^*, \Phi)[\![\pi_n(\Gamma)]\!] = & \pi_n(\text{flat}(M^*, \Phi)[\![\Gamma]\!]) \\
[17] & \text{flat}(M^*, \Phi)[\![\text{lift}(\Gamma)]\!] = & \text{lift}(\text{flat}(M^*, \Phi)[\![\Gamma]\!]) \\
[18] & \text{flat}(M^*, \Phi)[\![\text{drop}(\Gamma)]\!] = & \text{drop}(\text{flat}(M^*, \Phi)[\![\Gamma]\!]) \\
[19] & \text{flat}(M^*, \Phi)[\![\text{up}(\Gamma)]\!] = & \text{up}(\text{flat}(M^*, \Phi)[\![\Gamma]\!]) \\
[20] & \text{flat}(M^*, \Phi)[\![\text{max}(\Gamma)]\!] = & \text{max}(\text{flat}(M^*, \Phi)[\![\Gamma]\!])
\end{array}
$$

The function 'flat' has to consider all projection operations on specifications and has to be extended to all sorts embedded in specifications by means of distribution equations like the ones above. These equations are not shown.

14.3 Well-Formedness

In this section and the next we define a typechecker for multi-level specifications following the same approach as for one-level specifications. Well-formedness of fully annotated multi-level specifications is defined in §14.3.2. Rules for the complementary cases produce error messages for non-wellformed constructs in §14.3.3. Type assignment functions, defined in §14.4.1 and §14.4.2, produce a fully annotated specification for a plain specification an example of which is shown in Figure 14.1. Finally, the typechecker is defined in §14.5 as the composition of type assignment and well-formedness checking.

Typechecking of multi-level specifications differs at several points from typechecking one-level specifications. First of all, types at level n are terms over the signature at level $n + 1$. Secondly, types can be polymorphic. Finally, functions and variables can be overloaded, i.e., have more than one declaration in a signature.

14.3.1 Projection

We define a new projection function that finds the type of a function or variable in a list of declarations. The difference with the projection function from §12.3.1 is that the function yields the set of all types that are assigned to the function or variable, instead of the first type. If no declaration exists the empty set is produced. Furthermore, π takes a *set* of function or variable names as first argument and yields the set of all types for all functions or variables in the set.

module MLS-Projection
imports MLS$^{14.1.1}$ Renaming$^{B.2.10}$ Term-Sets$^{B.2.5}$
exports
  context-free syntax
    "π" "_" TermSet "(" Decls ")" $\to$ TermSet
    "π" "_" Var "(" MLS ")" $\to$ TermSet
    "π" "_" Fun "(" MLS ")" $\to$ TermSet
equations
The projection function π finds the types of a set of functions or variables in a list of declarations.

[1] $$\pi_{\Phi}() = \{\}$$
[2] $$\pi_{\Phi}(f : \tau) = \text{if } f \in \Phi \text{ then } \{\tau\} \text{ else } \{\}$$
[3] $$\pi_{\Phi}(x : \tau) = \text{if } x \in \Phi \text{ then } \{\tau\} \text{ else } \{\}$$
[4] $$\pi_{\Phi}(d_1^+;\ d_2^+) = \pi_{\Phi}(d_1^+) \cup \pi_{\Phi}(d_2^+)$$

The projection function π applied to a specification finds the type of a function or variable in the function or variable declarations of the signature of the lowest level.

[5] $$\pi_f(\Gamma) = \pi_{\{f\}}(\mathrm{F}(\mathrm{Sg}(\pi_0(\Gamma))))$$
[6] $$\pi_x(\Gamma) = \pi_{\{x;\ \mathrm{base}(x)\}}(\mathrm{V}(\mathrm{Sg}(\pi_0(\Gamma))))$$

In case of a variable not only the type of the variable, but also the type of its 'base' (variable without trailing digits or primes; see §B.2.10) is looked for. This makes it possible to use many variants of a variable with only one declaration. For example, if `A : type` is declared, then `A1, A2, A' : type` are implicitly declared as well. This facility encourages a consistent use of variable names.

14.3.2 Well-Formedness

As in the one-level case in §12.3.2, the well-formedness of fully annotated terms and specifications is defined by several well-formedness judgements—functions that yield an error Boolean value. An example of a fully annotated two-level specification is shown in Figure 14.1.

```
level 1
  signature
    sorts type ;
    functions
      (->) : type # type -> type ;
    variables
       A : type; B : type; C : type ;
level 0
  signature
    sorts A : type;
    functions
      k : (A : type)
          -> ((B : type) -> (A : type) : type) : type;
    variables
      X : A : type;
      Y : B : type;
  equations
    ((k : (A : type)
          -> ((Q : type) -> (A : type) : type) : type)
     (X : A : type) : (Q : type ) -> (A : type) : type)
      (Y : Q : type) : A : type
    == X : A : type
```

Figure 14.1: Example of a fully annotated two-level specification. Observe that the types at level 0 are fully annotated terms over level 1.

module MLS-WF
MLS[14.1.1] MLS-Projection[14.3.1] Error-Booleans[B.1.3] SPEC-Errors[B.2.2]
MLS-TA-Aux[14.4.1] Matching[B.2.8] Term-Analysis[12.2.3]
exports
context-free syntax
"$\vdash_{\text{mls}}$" MLS → EBool
MLS "$\vdash_{\text{spec}}$" Spec → EBool
MLS "$\vdash_{\text{sig}}$" Sig → EBool
MLS "$\vdash_{\text{sorts}}$" Terms → EBool
MLS "$\vdash_{\text{decls}}$" Decls → EBool
MLS "$\vdash_{\text{sort}}$" Term → EBool
MLS "$\vdash_{\text{trm}}$" Term ":" Term → EBool
MLS "$\vdash_{\text{term}}$" Term → EBool
MLS "$\vdash_{\text{eqs}}$" Eqs → EBool
equations
A multi-level specification is well-formed if each level is well-formed. The environment in which a specification is checked includes the specification itself

because that may contain relevant sort declarations.

[1] $\vdash_{\text{mls}} \Gamma = \Gamma \vdash_{\text{spec}} \pi_0(\Gamma) \quad \textbf{when} \ \max(\Gamma) = 0$

[2] $\vdash_{\text{mls}} \Gamma = \vdash_{\text{mls}} \text{up}(\Gamma) \leadsto \Gamma \vdash_{\text{spec}} \pi_0(\Gamma)$
$\textbf{when} \ \text{zero}(\max(\Gamma)) = \bot$

A specification is well-formed if both the signature and the equations are well-formed. The errors in the equations generally depend on errors in the signature. Therefore equation [3] gives precedence to signature errors over equation errors.

[3] $\Gamma \vdash_{\text{spec}} \mathcal{S} = \Gamma \vdash_{\text{sig}} \text{Sg}(\mathcal{S}) \leadsto \Gamma \vdash_{\text{eqs}} \text{E}(\mathcal{S})$

A signature is well-formed if the sorts section contains well-formed sort declarations and if the function and variable declarations are well-formed.

[4] $\Gamma \vdash_{\text{sig}} \Sigma = \Gamma \vdash_{\text{sorts}} \text{S}(\Sigma) \leadsto \Gamma \vdash_{\text{decls}} \text{F}(\Sigma) \wedge \Gamma \vdash_{\text{decls}} \text{V}(\Sigma)$

The terms in a sort declaration at level n should be well-formed terms over level $n+1$.

[5] $$\frac{\text{up}(\Gamma) \vdash_{\text{term}} \tau = \top}{\Gamma \vdash_{\text{sorts}} \tau \ = \ \top}$$

[6] $\Gamma \vdash_{\text{sorts}} \tau_1^+; \tau_2^+ \ = \ \Gamma \vdash_{\text{sorts}} \tau_1^+ \wedge \Gamma \vdash_{\text{sorts}} \tau_2^+$

[7] $\Gamma \vdash_{\text{sorts}} \ = \ \top$

A function or variable declaration is well-formed if its type is a well-formed sort.

[8] $\Gamma \vdash_{\text{decls}} x : \tau = \Gamma \vdash_{\text{sort}} \tau$

[9] $\Gamma \vdash_{\text{decls}} f : \tau = \Gamma \vdash_{\text{sort}} \tau$

[10] $\Gamma \vdash_{\text{decls}} = \top$

[11] $\Gamma \vdash_{\text{decls}} d_1^+; d_2^+ = \Gamma \vdash_{\text{decls}} d_1^+ \wedge \Gamma \vdash_{\text{decls}} d_2^+$

Sorts A term is a sort at level n if it is a term over level $n+1$, and if it matches one of the terms declared as sort at level n.

[12] $$\frac{\text{zero}(\max(\Gamma)) = \bot, \ \{\text{S}(\text{Sg}(\pi_0(\Gamma)))\} \geq t = \top}{\Gamma \vdash_{\text{sort}} t \ = \ \text{up}(\Gamma) \vdash_{\text{term}} t}$$

The predicate $\Phi \geq t$ (§B.2.8) tests whether a term t matches one of the elements of a set of terms Φ, in this case the set of sorts declared at level 0.

For a term to be a sort at the highest level it is sufficient to be a term over the next (implicit) level.

[13] $$\frac{\text{zero}(\max(\Gamma)) = \top}{\Gamma \vdash_{\text{sort}} t \ = \ \text{up}(\Gamma) \vdash_{\text{term}} t}$$

Otherwise all terms from the closure of the basic sorts under arrow and product that are used in function and variable declarations, would have to be declared explicitly as sorts.

Terms A complication with respect to the one-level case is that sorts are also annotated, except for the sorts at the highest level. We could solve this problem by introducing two different well-formedness predicates. Instead we use one predicate and the implicit annotation of terms with 'top'. The auxiliary judgement $\models_{\mathrm{trm}}$ is introduced to treat explicitly and implicitly annotated terms in the same way. The annotation of a term is constructed explicitly by splitting it in its term and type. This has the effect that terms that are annotated implicitly with 'top' can be treated in the same way as terms with explicit annotations.

[14] $$\Gamma \models_{\mathrm{term}} t = \Gamma \models_{\mathrm{trm}} \mathrm{term}(t) : \mathrm{type}(t)$$

The term 'top' has type 'top'. Since 'top' can not be declared as a function, this is the only possible type it can have.

[15] $$\Gamma \models_{\mathrm{trm}} \mathrm{top} : \mathrm{top} \;=\; \top$$

The types of functions and variables should be well-formed sorts. The type of a function should match one of the types with which it is declared. If a variable is declared, its type should match one of its declared types. Variables are allowed to be undeclared. The reason for this exception is that the type assignment algorithm has to invent new variables in some cases to prevent name clashes. A result of this choice is that variables can be used without declaration, if some reasonable type can be inferred for it from the context, or if it is given some suitable annotation.

[16] $$\frac{\pi_f(\Gamma) \geq \tau = \top}{\Gamma \models_{\mathrm{trm}} f : \tau \;=\; \top}$$

[17] $$\frac{\pi_x(\Gamma) = \Phi,\;\; \Phi \geq \tau \vee \mathrm{empty}(\Phi) = \top}{\Gamma \models_{\mathrm{trm}} x : \tau \;=\; \Gamma \models_{\mathrm{sort}} \tau}$$

A pair is well-formed if its type is the product of the types of its left and right components. An application is well-formed if the type of the argument matches the type of the domain of the type of the function and if the type of the annotation matches the type of the codomain.

[18] $$\frac{\mathrm{term}(\tau) = \mathrm{type}(t_1) \times \mathrm{type}(t_2)}{\Gamma \models_{\mathrm{trm}} t_1, t_2 : \tau \;=\; \Gamma \models_{\mathrm{term}} t_1 \wedge \Gamma \models_{\mathrm{term}} t_2}$$

[19] $$\frac{\mathrm{term}(\mathrm{type}(t_1)) = \mathrm{type}(t_2) \to \tau}{\Gamma \models_{\mathrm{trm}} t_1 \; t_2 : \tau \;=\; \Gamma \models_{\mathrm{term}} t_1 \wedge \Gamma \models_{\mathrm{term}} t_2}$$

Products and arrows are well-formed if their prefix versions $(\times)$ and $(\to)$ are declared in the signature as binary functions. The product of the types of the arguments t_1 and t_2 should be the domain and the annotation τ should be the codomain of the declaration of the function. This is checked in the same way as the annotation of a function is checked by matching the annotation of the function to one of its declarations. Because the type of the product or arrow is reconstructed, it is not clear what the annotations for the product and arrow in the types of $(\times)$ and $(\rightarrow)$ should be. For this purpose, the function 'bterm' (§12.2.3) is used to strip the annotation from the declared types.

[20] $$\frac{\mathrm{bterm}{*}(\pi_{(\times)}(\Gamma)) \geq \mathrm{type}(t_1) \times \mathrm{type}(t_2) \rightarrow \tau = \top}{\Gamma \vdash_{\mathrm{trm}} t_1 \times t_2 : \tau \;=\; \Gamma \vdash_{\mathrm{term}} t_1 \wedge \Gamma \vdash_{\mathrm{term}} t_2}$$

[21] $$\frac{\mathrm{bterm}{*}(\pi_{(\rightarrow)}(\Gamma)) \geq \mathrm{type}(t_1) \times \mathrm{type}(t_2) \rightarrow \tau = \top}{\Gamma \vdash_{\mathrm{trm}} t_1 \rightarrow t_2 : \tau \;=\; \Gamma \vdash_{\mathrm{term}} t_1 \wedge \Gamma \vdash_{\mathrm{term}} t_2}$$

Equations An equation is well-formed if both sides have the same type, the variables of the right-hand side are contained in the variables of the left-hand side and all occurrences of a variable on both sides have the same type.

[22] $$\frac{\begin{array}{c}\mathrm{type}(t_1) = \mathrm{type}(t_2),\;\; \mathrm{vars}(t_2) \subseteq \mathrm{vars}(t_1) = \top,\\ \text{var-types}(\mathrm{avars}(t_1,\, t_2)) = []\end{array}}{\Gamma \vdash_{\mathrm{eqs}} t_1 \equiv t_2 \;=\; \Gamma \vdash_{\mathrm{term}} t_1 \wedge \Gamma \vdash_{\mathrm{term}} t_2}$$

[23] $$\Gamma \vdash_{\mathrm{eqs}} \;=\; \top$$

[24] $$\Gamma \vdash_{\mathrm{eqs}} \varphi_1^+ ; \varphi_2^+ \;=\; \Gamma \vdash_{\mathrm{eqs}} \varphi_1^+ \wedge \Gamma \vdash_{\mathrm{eqs}} \varphi_2^+$$

The following proposition states that equality according to a well-formed specification is type preserving, i.e., a term can only be equal to another term if they have the same type.

Proposition 14.3.1 (Type Soundness) *Equational logic is type preserving for well-formed specifications: Let Γ be a fully annotated multi-level specification such that declarations in Γ use only free type constructors. If $\vdash_{\mathrm{mls}} \Gamma$ and $\Gamma \vdash_{\mathrm{term}} t_i$ then $\pi_0(\Gamma) \vdash t_1 \equiv t_2$ implies* $\mathrm{type}(t_1) = \mathrm{type}(t_2)$.

Proof. Since Γ is well-formed, all equations in $\pi_0(\Gamma)$ have equal types in the left-hand side and right-hand side and typed equational logic is type preserving for equations with that property (Proposition 12.2.1). □

The following proposition relates equalities over plain terms to equalities over fully annotated terms.

Proposition 14.3.2 *Equational derivability in a fully annotated specification implies equational derivability in the plain specification: Let Γ be a fully annotated multi-level specification such that declarations in Γ use only free type constructors and such that $\vdash_{\mathrm{mls}} \Gamma$, then $\Gamma \vdash t_1 \equiv t_2$ implies* $\mathrm{spine}(\Gamma) \vdash \mathrm{spine}(t_1) \equiv \mathrm{spine}(t_2)$

In §14.4.3 we discuss the requirements for the reverse implication; when does equality in the plain specification preserve types?

14.3.3 Non-wellformedness (MLS-NWF)

The generation of error messages for the non-wellformed cases is very similar to §12.3.3, therefore only the case of a non-wellformed application is presented.

module MLS-NWF
imports MLS-WF$^{14.3.2}$ SPEC-Errors$^{B.2.2}$

equations

[1] $\Gamma \vdash_{\mathrm{trm}} t_1\ t_2 : \tau$
$= (\Gamma \vdash_{\mathrm{term}} t_1 \wedge \Gamma \vdash_{\mathrm{term}} t_2)$
$\rightsquigarrow$ application " spine(t_1 t_2) " not well-formed
:: if $\neg$ `A` $\rightarrow$ `B` $\geq$ term(type(t_1))
then " spine(t_1) " is not a function
else if $\neg$ eq(dom(term(type(t_1))), type(t_2))
then type of argument " type(t_2)
" does not match type of domain " dom(term(type(t_1))) "
else type of result " spine(τ)
" does not match type of codomain " cod(term(type(t_1))) "
otherwise

14.4 Type Assignment

In the next section we will define the type assignment functions for the multi-level case. First, we define several auxiliary functions that are needed for the definition of type assignment.

14.4.1 Preliminaries

The two major complications in type assignment for multi-level specifications are overloading and polymorphism.

Overloading caused by multiple declarations of variables and functions leads to multiple fully annotated terms for a single plain term. Therefore, the type assignment function for terms yields a set of annotated terms instead of a single term. To assign types to a composite term such as an application, first the subterms are assigned types, resulting in a pair of sets of terms. Each combination from the two sets can form a well-formed application. Therefore, each term in the Cartesian product of the two sets has to be considered.

Join To handle polymorphism correctly, type variables of terms composed by application, pairing etc. have to be renamed before types can be compared, because types are implicitly universally quantified. The function $\bowtie$ (join) combines the function of renaming type variables and producing the cartesian product of two sets. Given two sets Φ_1 and Φ_2 it renames the type variables in the terms in the two sets leading to sets Φ_3 and Φ_4 such that the type variables are disjunct, i.e., $\text{tvars}(\Phi_3) \cap \text{tvars}(\Phi_4) = \{\}$. The operation $\text{rn}\Phi[\Phi']$, given a set of variables Φ', produces a renaming of the variables in the set Φ such that they do not occur in Φ' (see §B.2.10). The result of the operation is the Cartesian product $\Phi_3 \times \Phi_4$, i.e., the set of all pairs (t_1, t_2) of elements from $t_1 \in \Phi_3$ and $t_2 \in \Phi_4$ (see also §B.2.5).

module MLS-TA-Aux
imports Renaming$^{B.2.10}$
exports
  context-free syntax
    TermSet "$\bowtie$" TermSet $\rightarrow$ TermSet {**non-assoc**}
equations

$$[1] \quad \frac{\begin{array}{c}\text{vars}(\Phi_2) = \Phi_2',\ \text{rn tvars}(\Phi_1) \cap \Phi_2'[\text{vars}(\Phi_1) \cup \Phi_2']{*}(\Phi_1) = \Phi_3, \\ \text{vars}(\Phi_3) = \Phi_3',\ \text{rn tvars}(\Phi_2) \cap \Phi_3'[\Phi_2' \cup \Phi_3']{*}(\Phi_2) = \Phi_4\end{array}}{\Phi_1 \bowtie \Phi_2 \ = \ \Phi_3 \times \Phi_4}$$

Selection Once two sets of terms have been joined, the well-formed pairs have to be selected and given a type annotation. This involves tests and type forming operations for each construct applying the test to each element in the set of pairs thereby keeping only the correct ones. This last aspect can be specified generically for all constructs. For each construct we use a function of sort (Term $\Rightarrow$ Bool$\times$TermSet)[1], which given a term produces a pair of a Boolean value indicating whether the term is well-formed and a set of terms resulting from assigning a type to that term. This function can be mapped over a set of terms resulting from the join of two type-assignments by the function '$*$'. It applies the function to each element of the argument set remembering whether a well-formed term was already encountered. If at the end of the list none of the combinations turns out to be well-formed, then the last, non-wellformed one, is returned. This guarantees that type assignment always returns a term. From this non-wellformed term the well-formedness judgements can find out the cause of the error.

imports Term-Analysis$^{12.2.3}$ Term-Functions$^{B.2.3}$
exports
  sorts (Bool $\times$ TermSet) (Term $\Rightarrow$ Bool $\times$ TermSet)
  context-free syntax
    "$\langle$" Bool "," TermSet "$\rangle$" $\rightarrow$ (Bool $\times$ TermSet)
    (Term $\Rightarrow$ Bool $\times$ TermSet) "(" Term ")" $\rightarrow$ (Bool $\times$ TermSet)
    (Term $\Rightarrow$ Bool $\times$ TermSet)
    "$*$" "(" TermSet ")" $\rightarrow$ TermSet
    (Term $\Rightarrow$ Bool $\times$ TermSet)
    "$*$" "(" Bool "," TermSet ")" $\rightarrow$ TermSet
  variables

[1] Note that we instructed ToLATEX to typeset the sort identifier Term2BoolXTermSet as (Term $\Rightarrow$ Bool $\times$ TermSet)

"G"$[0\text{-}9']*$ → (Term ⇒ Bool × TermSet)
equations

$$
\begin{array}{lrl}
[2] & G*(\Phi) = & G*(\bot,\, \Phi) \\
[3] & G*(b,\, \{\}) = & \{\} \\
[4] & G*(b,\, \{t\}) = & \text{if } b \wedge \neg\, b' \text{ then } \{\} \text{ else } \Phi \quad \textbf{when} \;\; G(t) = \langle b',\, \Phi\rangle \\
[5] & G*(b,\, \{t;\, t^{+}\}) = & (\text{if } b' \text{ then } \Phi \text{ else } \{\}) \cup G*(b \vee b',\, \{t^{+}\}) \\
 & & \textbf{when} \;\; G(t) = \langle b',\, \Phi\rangle
\end{array}
$$

For functions of sort (Term ⇒ Bool×Eqs), which yield a list of equations instead of a set of terms, similar functions are defined.

Annotation with a Set of Types Due to overloading, the result of assigning a type to a term is a set of terms instead of a single term. This means that the assignment of types in declarations and type annotations also leads to a set of types. These should be translated to lists of declarations and sets of terms, respectively. The following functions can be used to construct the declaration of a function or variable or the annotation of a term with a set of terms. The ambiguity in a declaration is translated to multiple declarations for the function or variable, i.e., $f : \{\tau_1, \tau_2\} = f : \tau_1; f : \tau_2$. The annotation of a term with a set of terms is translated to the set of the term with all the annotations from the set.

imports MLS$^{14.1.1}$
exports
  context-free syntax
    Fun ":" TermSet → Decls
    Var ":" TermSet → Decls
    Term ":" TermSet → TermSet
  priorities
    Term ":"TermSet → TermSet > TermSet "∪"TermSet → TermSet

$$
\begin{array}{lrl}
[6] & f : \{\} = & \\
[7] & f : \{t\} = & f : t \\
[8] & f : \{t_1^{+};\, t_2^{+}\} = & f : \{t_1^{+}\} \mathbin{+\!\!+} f : \{t_2^{+}\} \\
[9] & x : \{\} = & \\
[10] & x : \{t\} = & x : t \\
[11] & x : \{t_1^{+};\, t_2^{+}\} = & x : \{t_1^{+}\} \mathbin{+\!\!+} x : \{t_2^{+}\} \\
[12] & t : \{\} = & \{t : \text{nil}\} \\
[13] & t : \{\tau\} = & \{t : \tau\} \\
[14] & t : \{t_1^{+};\, t_2^{+}\} = & t : \{t_1^{+}\} \cup t : \{t_2^{+}\}
\end{array}
$$

Variable Type Consistency The function 'var-types' checks whether the types of the variables in a set of terms of the form $x : \tau$ (annotated variables) are consistent, i.e., two occurrences of a variable should have types that are unifiable. If this is the case the function returns a substitution that makes the types of all occurrences of the same variable equal. The function is used as follows: Given a term t, var-types(avars(t)) either gives $\perp$, which indicates that t contains two occurrences of the same variable with incompatible type annotations or a substitution σ that makes all occurrences of the same variable in t the same.

imports Unification$^{B.2.9}$
exports
  context-free syntax
    var-types(TermSet) $\rightarrow$ Subst$_\perp$
    var-eqs(TermSet) $\rightarrow$ Eqs
equations

[15]	$\text{var-types}(\Phi) = \text{mgu}(\text{var-eqs}(\Phi))$
[16]	$\text{var-eqs}(\{\}) =$
[17]	$\text{var-eqs}(\{x : \tau_1;\ t_1^*;\ x : \tau_2;\ t_2^*\}) = \tau_1 \equiv \tau_2 +\!\!+ \text{var-eqs}(\{t_1^*;\ x : \tau_2;\ t_2^*\})$
[18]	$\text{var-eqs}(\{x : \tau_1;\ t^*\}) = \text{var-eqs}(\{t^*\})$ **otherwise**

New Variables The function 'new-var' generates a variable name that is not declared in the signature at level 0. Given a set of variables Φ 'nv' picks the first element of Φ that is not declared in Γ. If all variables are declared, the variables in Φ are renamed by prepending an extra letter (`Q`) to each variable in Φ.

imports MLS-Projection$^{14.3.1}$
exports
  context-free syntax
    new-var "(" MLS ")" $\rightarrow$ Term
    nv "(" MLS ")" "(" TermSet "," TermSet ")" $\rightarrow$ Term
equations

[19]	$\text{new-var}(\Gamma) = \text{nv}(\Gamma)(\{\}, \{\})$
[20]	$\text{nv}(\Gamma)(\Phi, \{\}) = \text{nv}(\Gamma)(\Phi', \Phi')$ **when** $\Phi' = \text{add}(\texttt{Q}, \Phi)$
[21]	$\text{nv}(\Gamma)(\Phi, \{x;\ t^*\}) = \text{if empty}(\pi_x(\Gamma)) \text{ then } x \text{ else nv}(\Gamma)(\Phi, \{t^*\})$

14.4.2 Type Assignment

The basic ideas for type assignment of multi-level specifications are similar to the one-level case. For instance, the type of an application is the codomain of the first (function) argument. The complications are caused by the multi-level aspect (types are typed terms), overloading and polymorphism. The basic idea in dealing with overloading is to create a set of all possible typings for each term; type assignment function 'Wt' returns a TermSet. When terms are combined, all possible combinations of the associated sets have to be considered. The join and select functions of the previous section are applied for this purpose.

Type assignment of multi-level specifications proceeds by first annotating the higher levels and using the resulting annotated specification to assign types to the signature at level 0. The resulting signature can be used to assign types to the equations at level 0.

module MLS-TA
imports MLS$^{14.1.1}$ MLS-TA-Aux$^{14.4.1}$ MLS-Projection$^{14.3.1}$
Term-Analysis$^{12.2.3}$ Matching$^{B.2.8}$
exports
context-free syntax

"Wm" "⟦" MLS "⟧"	→ MLS
"Wsp" "(" MLS ")" "⟦" Spec "⟧"	→ Spec
"Wsg" "(" MLS ")" "⟦" Sig "⟧"	→ Sig
"Wd" "(" MLS ")" "⟦" Decls "⟧"	→ Decls
"Ws" "(" MLS ")" "⟦" Term "⟧"	→ TermSet
"Wss" "(" MLS ")" "⟦" Terms "⟧"	→ Terms
"Wtv" "(" MLS ")" "⟦" Term "⟧"	→ TermSet
"Wt" "(" MLS ")" "⟦" Term "⟧"	→ TermSet
"Wts" "(" MLS ")" "⟦" Terms "⟧"	→ Terms
"We" "(" MLS ")" "⟦" Eqs "⟧"	→ Eqs

equations
Assigning types to a specification consists of assigning types to all levels of the signature and using the resulting signature to assign types to the equations.

[1] $$\frac{\mathrm{zero}(\max(\Gamma)) = \top}{\mathrm{Wm}[\![\Gamma]\!] \ = \ \text{level } 0 \ \mathrm{Wsp}(\mathrm{lift}(\mathrm{up}(\Gamma)))[\![\pi_0(\Gamma)]\!]}$$

[2] $$\frac{\mathrm{zero}(\max(\Gamma)) = \bot, \ \ \mathrm{lift}(\mathrm{Wm}[\![\mathrm{up}(\Gamma)]\!]) = \Gamma'}{\mathrm{Wm}[\![\Gamma]\!] \ = \ \Gamma'; \text{level } 0 \ \mathrm{Wsp}(\Gamma')[\![\pi_0(\Gamma)]\!]}$$

A specification is annotated by first annotating the signature using the higher levels and then annotating the equations using the higher levels extended with the annotated signature.

[3] $$\frac{\mathrm{Wsg}(\Gamma)[\![\mathrm{Sg}(\mathcal{S})]\!] = \Sigma, \ \ \Gamma' = \text{level } 0 \text{ signature } \Sigma}{\mathrm{Wsp}(\Gamma)[\![\mathcal{S}]\!] \ = \ \text{signature } \Sigma; \text{equations } \mathrm{We}(\Gamma;\Gamma')[\![\mathrm{E}(\mathcal{S})]\!]}$$

Assign types to each section of a signature.

[4] $$\frac{\text{sorts } \mathrm{Wss}(\Gamma)[\![\mathrm{S}(\Sigma)]\!] = \Sigma_2, \ \ \Gamma' = \Gamma; \text{level } 0 \text{ signature } \Sigma_2}{\begin{array}{rl}\mathrm{Wsg}(\Gamma)[\![\Sigma]\!] \ = & \Sigma_2; \\ & \text{functions } \mathrm{Wd}(\Gamma')[\![\mathrm{F}(\Sigma)]\!]; \\ & \text{variables } \mathrm{Wd}(\Gamma')[\![\mathrm{V}(\Sigma)]\!]\end{array}}$$

The sorts in the declarations of sorts, functions and variables are treated as terms over the signature at the next level.

[5] $$\mathrm{Wd}(\Gamma)[\![\,]\!] =$$

[6] $$\mathrm{Wd}(\Gamma)[\![f:\tau]\!] = f : \mathrm{Ws}(\Gamma)[\![\tau]\!]$$

[7] $$\mathrm{Wd}(\Gamma)[\![x:\tau]\!] = x : \mathrm{rn\ vars}(\Phi)[\{x\}]{*}(\Phi) \quad \textbf{when} \ \ \mathrm{Ws}(\Gamma)[\![\tau]\!] = \Phi$$

[8] $$\mathrm{Wd}(\Gamma)[\![d_1^+;\, d_2^+]\!] = \mathrm{Wd}(\Gamma)[\![d_1^+]\!] +\!\!+ \mathrm{Wd}(\Gamma)[\![d_2^+]\!]$$

Sorts A sort at level n is a term over level $n+1$. Only the annotations that match a sort declaration are selected in case a declaration is ambiguous. The function 'srt' selects a term if it matches one of the terms in the set in its first argument.

[9] $$\mathrm{Ws}(\Gamma)[\![\tau]\!] = \mathrm{srt}(\{\mathrm{S}(\mathrm{Sg}(\pi_0(\Gamma)))\})*(\mathrm{Wtv}(\mathrm{up}(\Gamma))[\![\tau]\!])$$

[10] $$\mathrm{srt}(\Phi)(\tau) = \langle \top, \{\tau\}\rangle \quad \textbf{when } \Phi \geq \tau = \top$$

[11] $$\mathrm{srt}(\Phi)(\tau) = \langle \bot, \{\tau\}\rangle \quad \textbf{otherwise}$$

A list of sort terms at level n is a list of terms over level $n+1$.

[12] $$\mathrm{Wss}(\Gamma)[\![ts]\!] = \mathrm{Wts}(\mathrm{up}(\Gamma))[\![ts]\!]$$

Terms with Variables The function 'Wt' defined below assigns types to a term without considering the consistency of the types of variables. The function 'Wtv' first assigns a type to a term using 'Wt' and then applies 'var-types' (§14.4.1) to make the types of different occurrences of the same variable equal.

[13] $$\frac{\mathrm{Wt}(\Gamma)[\![t]\!] = \Phi,\ \text{var-types}(\mathrm{avars}(\Phi)) = \sigma_\bot}{\mathrm{Wtv}(\Gamma)[\![t]\!] \ = \ \text{if fail?}(\sigma_\bot) \text{ then } \Phi \text{ else } \Downarrow_\bot(\sigma_\bot)*(\Phi)}$$

Functions and Variables Functions get assigned the type from the declaration in the signature.

[14] $$\mathrm{Wt}(\Gamma)[\![f]\!] = f : \pi_f(\Gamma)$$

The type assignment to variables is somewhat more complicated since undeclared variables are taken into account according to the following rules. Equation [15] deals with variables in types of the top signature. Equation [16] finds the set of declared types Φ for a variable x. If Φ is not empty, i.e., the variable is declared, x is annotated with Φ. If there is no declaration (Φ is empty), a new type variable is generated to assign to x, which is assigned a type as a term over the next level. This is necessary to ensure that a term has the right number of annotations.

[15] $$\frac{\mathrm{up}(\Gamma) = \text{top-sig}}{\mathrm{Wt}(\Gamma)[\![x]\!] \ = \ \{x\}}$$

[16] $$\frac{\begin{array}{c}\mathrm{up}(\Gamma) \neq \text{top-sig},\ \pi_x(\Gamma) = \Phi,\\ \text{if empty}(\Phi) \text{ then } \mathrm{Wt}(\mathrm{up}(\Gamma))[\![\text{new-var}(\mathrm{up}(\Gamma))]\!] \text{ else } \Phi = \{ts\}\end{array}}{\mathrm{Wt}(\Gamma)[\![x]\!] \ = \ x : \{\text{rn vars} *(\ ts)[\{x\}]*(ts)\}}$$

Nil and Top Nil can not occur in well-formed specifications. Top can only occur as a top-level type.

[17] $$\mathrm{Wt}(\Gamma)[\![\mathrm{nil}]\!] = \{\mathrm{nil} : \mathrm{nil}\}$$

[18] $$\mathrm{Wt}(\Gamma)[\![\mathrm{top}]\!] = \{\mathrm{top} : \mathrm{top}\}$$

Auxiliary Functions For the type assignment of non-atomic terms we need the following auxiliary functions.

hiddens

context-free syntax

srt(TermSet)	$\to$ (Term $\Rightarrow$ Bool $\times$ TermSet)
app(MLS)	$\to$ (Term $\Rightarrow$ Bool $\times$ TermSet)
pr(MLS)	$\to$ (Term $\Rightarrow$ Bool $\times$ TermSet)
arr	$\to$ (Term $\Rightarrow$ Bool $\times$ TermSet)
prd	$\to$ (Term $\Rightarrow$ Bool $\times$ TermSet)
ann	$\to$ (Term $\Rightarrow$ Bool $\times$ TermSet)
eqn	$\to$ (Term $\Rightarrow$ Bool $\times$ Eqs)
new-arrow(MLS)	$\to$ Term

equations

Application An application term is assigned the codomain of the type of the function. To this end, both arguments are assigned types and the result terms are joined. The type of the term in the argument position should conform to the argument type of the function.

[19] $$\begin{aligned}\mathrm{Wt}(\Gamma)[\![t_1\ t_2]\!] \ =\ \mathrm{app}(\Gamma) *((&\mathrm{Wt}(\Gamma)[\![t_1]\!] \bowtie \mathrm{Wt}(\Gamma)[\![t_2]\!]) \\ &\bowtie \{\mathsf{a} : \text{new-arrow}(\Gamma)\}\ )\end{aligned}$$

[20] $$\frac{\mathrm{mgu}(\mathrm{type}(t_1) \equiv \tau_1;\ \mathrm{type}(t_2) \equiv \tau_2) = \sigma}{\mathrm{app}(\Gamma)((t_1,\ t_2),\ \mathsf{a} : (\tau_1,\ \tau_2,\ \tau_3)) \ =\ \langle \top,\ \{\sigma(t_1\ t_2 : \tau_3)\}\rangle}$$

[21] $$\mathrm{app}(\Gamma)((t_1,\ t_2),\ \tau) \ =\ \langle \bot,\ \{t_1\ t_2 : \mathrm{nil}\}\rangle \quad \textbf{otherwise}$$

The function 'new-arrow' constructs an arrow type with new variables as domain and codomain, annotates it with types and yields a triple of the arrow type, domain and codomain.

[22] $$\frac{\begin{gathered}\text{new-var}(\mathrm{up}(\Gamma)) = x,\ \ x' = \mathrm{prime}(x),\ \ \tau_0 = x \to x', \\ \text{if zero}(\max(\Gamma)) \text{ then } \{\tau_0\} \text{ else } \mathrm{Wt}(\mathrm{up}(\Gamma))[\![\tau_0]\!] = \{\tau_1;\ t^*\}\end{gathered}}{\text{new-arrow}(\Gamma) \ =\ \tau_1,\ \mathrm{dom}(\mathrm{term}(\tau_1)),\ \mathrm{cod}(\mathrm{term}(\tau_1))}$$

Pair A pair (t_1, t_2) has the product type $\tau_1 \times \tau_2$ if τ_i is the type of t_i. The product is itself a term over the next level.

[23] $$\mathrm{Wt}(\Gamma)[\![t_1,\ t_2]\!] = \mathrm{pr}(\Gamma)*(\mathrm{Wt}(\Gamma)[\![t_1]\!] \bowtie \mathrm{Wt}(\Gamma)[\![t_2]\!])$$

[24] $$\mathrm{pr}(\Gamma)(t_1,\ t_2) = \langle \top,\ \{(t_1,\ t_2) : \mathrm{type}(t_1) \times \mathrm{type}(t_2)\}\rangle \quad \textbf{when}\ \ \mathrm{zero}(\max(\Gamma)) = \top$$

[25] $$\mathrm{pr}(\Gamma)(t_1,\ t_2) = \langle \top,\ t_1,\ t_2 : \mathrm{Wt}(\mathrm{up}(\Gamma))[\![\mathrm{type}(t_1) \times \mathrm{type}(t_2)]\!]\rangle \quad \textbf{when}\ \ \mathrm{zero}(\max(\Gamma)) = \bot$$

Arrow and Product Arrow and product are defined in terms of application of the functions $(\rightarrow)$ and $(\times)$ to their arguments. After type assignment the binary notation is restored for readability.

$$
\begin{array}{lrl}
[26] & \mathrm{Wt}(\Gamma)[\![t_1 \rightarrow t_2]\!] = & \mathrm{arr}{*}(\mathrm{Wt}(\Gamma)[\![(\rightarrow)\ (t_1,\, t_2)]\!]) \\
[27] & \mathrm{arr}(t) = & \langle \top,\, \{t_1 \rightarrow t_2 : \mathrm{type}(t)\}\rangle \\
 & & \textbf{when}\ \ \mathrm{bapp}(t) = (\rightarrow)\ (t_1,\, t_2) \\
[28] & \mathrm{arr}(t) = & \langle \bot,\, \{t\}\rangle \quad \textbf{otherwise} \\
[29] & \mathrm{Wt}(\Gamma)[\![t_1 \times t_2]\!] = & \mathrm{prd}{*}(\mathrm{Wt}(\Gamma)[\![(\times)\ (t_1,\, t_2)]\!]) \\
[30] & \mathrm{prd}(t) = & \langle \top,\, \{t_1 \times t_2 : \mathrm{type}(t)\}\rangle \\
 & & \textbf{when}\ \ \mathrm{bapp}(t) = (\times)\ (t_1,\, t_2) \\
[31] & \mathrm{prd}(t) = & \langle \bot,\, \{t\}\rangle \quad \textbf{otherwise}
\end{array}
$$

Annotation A term $t : \tau$ that already has a type annotation τ, has to be assigned a type that conforms with τ and τ itself should be assigned a type as a term at the next level of Γ.

$$
\begin{array}{lrl}
[32] & \mathrm{Wt}(\Gamma)[\![t : \tau]\!] = & \mathrm{ann}{*}(\mathrm{Wt}(\Gamma)[\![t]\!] \bowtie \mathtt{a} : \mathrm{Wt}(\mathrm{up}(\Gamma))[\![\tau]\!]) \\
[33] & \mathrm{ann}(t : \tau_1,\, \mathtt{a} : \tau_2) = & \langle \top,\, \{\sigma(t : \tau_2)\}\rangle \quad \textbf{when}\ \ \mathrm{mgu}(\tau_1 \equiv \tau_2) = \sigma \\
[34] & \mathrm{ann}(t,\, \mathtt{a} : \tau) = & \langle \bot,\, \{t : \tau\}\rangle \quad \textbf{otherwise}
\end{array}
$$

Lists of Terms

$$
\begin{array}{lrl}
[35] & \mathrm{Wts}(\Gamma)[\![]\!] = & \\
[36] & \mathrm{Wts}(\Gamma)[\![t]\!] = & ts \quad \textbf{when}\ \ \{ts\} = \mathrm{Wt}(\Gamma)[\![t]\!] \\
[37] & \mathrm{Wts}(\Gamma)[\![t_1^+;\, t_2^+]\!] = & \mathrm{Wts}(\Gamma)[\![t_1^+]\!] +\!\!+ \mathrm{Wts}(\Gamma)[\![t_2^+]\!]
\end{array}
$$

Equations An equation is annotated by annotating both sides of the equation. The types of the resulting terms should be unifiable and if this is the case the unifier is applied to both term to make the types equal.

$$
\begin{array}{lrl}
[38] & \mathrm{We}(\Gamma)[\![t_1 \equiv t_2]\!] = & \mathrm{eqn}{*}(\mathrm{Wt}(\Gamma)[\![t_1]\!] \bowtie \mathrm{Wt}(\Gamma)[\![t_2]\!]) \\
[39] & \mathrm{eqn}(t_1,\, t_2) = & \langle \top,\, \sigma_2 \circ \sigma_1(t_1 \equiv t_2)\rangle \\
 & & \textbf{when}\quad \text{var-types}(\mathrm{avars}(t_1,\, t_2)) = \sigma_1, \\
 & & \qquad \mathrm{mgu}(\sigma_1(\mathrm{type}(t_1) \equiv \mathrm{type}(t_2))) = \sigma_2 \\
[40] & \mathrm{eqn}(t_1,\, t_2) = & \langle \bot,\, t_1 \equiv t_2\rangle \quad \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lrl}
[41] & \mathrm{We}(\Gamma)[\![]\!] = & \\
[42] & \mathrm{We}(\Gamma)[\![\varphi_1^+;\, \varphi_2^+]\!] = & \mathrm{We}(\Gamma)[\![\varphi_1^+]\!] +\!\!+ \mathrm{We}(\Gamma)[\![\varphi_2^+]\!]
\end{array}
$$

Correctness The type assignment functions defined above produce a fully annotated specification given an arbitrary plain, partially annotated or fully annotated specification. Type assignment always succeeds, but the resulting specification is not necessarily well-formed. The following propositions state that type assignment produces a well-formed result whenever that is possible. The expression $\Phi \geq t'$ expresses that t' is an instantiation of one of the terms in Φ. Because we can choose t' arbitrarily as long as it is well-formed the proposition states that Wt finds *all most general annotations of* t.

Proposition 14.4.1 (Correctness of Wt**)** *The function* Wt *finds all correct typings for a term if any exist. Let* Γ *be a multi-level specification with free types such that* $\vdash_{\text{mls}} \Gamma$. *Given a term* t, *if there exists a full annotation* t' *of* t, *i.e.,* $\text{spine}(t') = \text{spine}(t)$, *such that* $\Gamma \vdash_{\text{term}} t'$ *and if* $\Phi = \text{Wt}(\Gamma)[\![t]\!]$, *then* $\Phi \geq t'$ *and for all* $t'' \in \Phi$, $\Gamma \vdash_{\text{term}} t''$.

Proof. by induction on t. □

If no functions are overloaded, terms have a single full annotation. The previous proposition states that this single annotation is 'principal', i.e., the most general type assignment of the term.

Proposition 14.4.2 *Let* Γ *be a fully annotated multi-level specification with free types such that* $\vdash_{\text{mls}} \Gamma$ *and such that for each* f, $|\pi_f(\Gamma)| \leq 1$, *then we have* $|\text{Wt}(\Gamma)[\![t]\!]| = 1$.

Similarly, we have that Wm finds a well-formed full annotation for a specification if one exists.

Proposition 14.4.3 (Correctness of Wm**)** *If* $\vdash_{\text{mls}} \Gamma$ *then* $\vdash_{\text{mls}} \text{Wm}[\![\text{spine}(\Gamma)]\!]$.

The result of type assignment is an expression over the original language to which type assignment can again be applied.

Proposition 14.4.4 *Type assignment is idempotent, i.e.,*

$$\bigcup_{t' \in \text{Wt}(\Gamma)[\![t]\!]} \text{Wt}(\Gamma)[\![t']\!] = \text{Wt}(\Gamma)[\![t]\!].$$

14.4.3 Disambiguation and Confluence

We saw in §14.3.2 that well-formedness of a specification ensures that derivable equality is type preserving. As a corollary, term rewriting with a well-formed specification is type preserving. Furthermore, the type assignment function for multi-level specifications yields a well-formed annotation of a specification if one exists. However, we have not yet looked at the consequences of overloading resolution by type assignment for term rewriting. Is the plain term rewrite system the same as the annotated rewrite system? Although this is the case for some specifications, in general the answer to this question is no.

Non-Confluence Caused by Overloading Due to overloading, the plain term rewrite system (TRS) of a specification can be non-confluent while the annotated TRS is confluent. A TRS is confluent if it does not matter which matching equation is taken for a rewrite step. For example, the following module `eqda` defines equality on Boolean values and on lists in the style of the data algebra of

Bergstra and Sellink (1996). Module `list_access` extends module `list` from §13.1 with the function `empty` for testing emptiness of a list and the functions `hd` and `tl`, which give the head and tail of a list. The variables `X` and `Y` are generic variables.

```
module eqda
imports bool, list_access;
level 0
  signature
    functions
      eq : A # A -> bool;
    equations
      eq(X, Y) == X <-> Y;
      eq(X, Y) == (empty(X) /\ empty(Y))
                  \/ ((~(empty(X)) /\ ~(empty(Y)))
                      /\ (eq(hd(X), hd(Y)) /\ eq(tl(X), tl(Y))));
```

The plain term rewrite system of this module is not confluent because the two `eq` equations have the same left-hand side but completely unrelated right-hand sides. For instance, either equation can be used to rewrite the term `eq(t,f)`. Only if the first equation is chosen the expected result is achieved. The TRS of the module becomes confluent if we consider its full annotation. The types of variables `X` and `Y` in the right-hand sides force the right types in the left-hand sides. The annotation of `eq` in the first equation becomes `bool # bool -> bool` and in the second equation `list(A) # list(A) -> bool`.

The next example shows that even while the plain TRS is confluent it can have different normal forms than the annotated TRS. The function `(/)` is used as constructor for positive rational numbers and as defined exclusive or function for the Booleans. When regarded as a plain TRS, rationals of the form `X/Y` are rewritten anyway.

```
  signature
    functions
      (/)             : nat  # nat  -> rat;
      (\/), (/\), (/) : bool # bool -> bool;
  equations
    X / Y == (~X /\ Y) \/ (X /\ ~Y)
```

These examples clearly show that, in general, types are needed to disambiguate the equations of specifications. However, in many cases where matching is used and constructors and defined functions do not have overlapping names, overloading is resolved by the choice of constructors in the left-hand side of an equation. An example is the definition of the generalization of `zip` to tuples, for which it is not even clear how typed rewriting should be done, but untyped rewriting does not go wrong. Although it is often clear by examination whether types can be discarded, it is not clear how this property can be tested. For rewriting purposes it seems to be sufficient to annotate only functions with their type, i.e., apply function 'fspine' to the specification which removes all annotations except those of functions. It is not clear whether all ambiguities due to overloading are resolved in the fspine of a fully annotated specification.

Ambiguous Equations Due to overloading an untyped equation can actually denote several typed equations. An example is the equation `size([]) == 0` in Figure 11.1 on page 221. As another example consider the overloaded numerical operations in module `num` below. It is clear that the equations for addition that involve `0` and `s` are valid for both naturals and integers. The type assignment function 'We' produces all annotations of an equation for which the types of left-hand side and right-hand side match.

```
module num
level 0
  signature
    sorts nat; int;
    functions
      0   : nat;              0    : int;
      s   : nat -> nat;       s, p : int -> int;
      (+) : nat # nat -> nat;  (+)  : int # int -> int;
      i   : nat -> int;
    variables
      X, Y : nat; X, Y : int;
  equations
    0    + Y  == Y;          s(p(X)) == X;   i(0)    == 0;
    s(X) + Y  == X + s(Y);   p(s(X)) == X;   i(s(X)) == s(i(X));
    p(X) + Y  == X + p(Y);
```

14.5 Typechecking

The typecheck function for multi-level specifications is again constructed from a well-formedness predicate and a type assignment function. The main typecheck function checks a multi-level specification. In addition there are two predicates to check terms and equations over a multi-level signature.

module MLS-TC
imports MLS-TA$^{14.4.2}$ MLS-NWF$^{14.3.3}$
exports
  context-free syntax
    tc "⟦" MLS "⟧" → EBool
    tc "(" MLS ")" "⟦" Term "⟧" → EBool
    tc "(" MLS ")" "⟦" Eqs "⟧" → EBool
equations

[1] $$\mathrm{tc}[\![\Gamma]\!] \;=\; \vdash_{\mathrm{mls}} \mathrm{Wm}[\![\Gamma]\!]$$

[2] $$\frac{\mathrm{Wm}[\![\Gamma]\!] = \Gamma',\ \ \mathrm{Wt}(\Gamma')[\![t]\!] = \{t';\, t^*\}}{\mathrm{tc}(\Gamma)[\![t]\!] \ = \ \vdash_{\mathrm{mls}} \Gamma' \leadsto \Gamma' \vdash_{\mathrm{term}} t'}$$

[3] $$\frac{\mathrm{Wm}[\![\Gamma]\!] = \Gamma'}{\mathrm{tc}(\Gamma)[\![\mathcal{E}]\!] \ = \ \vdash_{\mathrm{mls}} \Gamma' \leadsto \Gamma' \vdash_{\mathrm{eqs}} \mathrm{We}(\Gamma')[\![\mathcal{E}]\!]}$$

14.5.1 Typechecking Modular Specifications

Finally, we define typechecking of a list of modules. The approach is rather crude. First all modules are flattened, then the MLS of each module is typechecked. This is of course rather expensive because code is duplicated. Observe that with this approach types and equations are disambiguated *after* being imported. This entails that newly introduced function declarations of existing functions may cause previously unambiguous equations to become ambiguous.

module MMLS-TC
imports MMLS$^{14.2}$ MLS-TC$^{14.5}$
exports
  context-free syntax
    tc "$[\![$" Modules "$]\!]$" $\to$ EBool
    tc1 "$[\![$" Modules "$]\!]$" $\to$ EBool
equations

[1] $$\mathrm{tc}[\![M^*]\!] = \mathrm{tc1}[\![\mathrm{flat}(M^*)]\!]$$
[2] $$\mathrm{tc1}[\![]\!] = \top$$
[3] $$\mathrm{tc1}[\![\text{module } f\,\Gamma;]\!] = \text{errors in module " } f \text{ " :: } \mathrm{tc}[\![\Gamma]\!]$$
[4] $$\mathrm{tc1}[\![M_1^+\ M_2^+]\!] = \mathrm{tc1}[\![M_1^+]\!] \wedge \mathrm{tc1}[\![M_2^+]\!]$$

This concludes the specification of the syntax, semantics and typechecking of modular multi-level specifications.

14.6 Discussion and Concluding Remarks

14.6.1 Related Work

In §11.1 we discussed several formalisms related to the formalism MLS described in this chapter. Here we give some pointers to other related issues.

Type Surveys Cardelli and Wegner (1985) give an informal introduction to types in programming languages including polymorphism, existential types and subtypes. Cardelli (1993) discusses a wide variety of programming features and their types, including mutable types, exception types, tuple types, option types, recursive types and subtypes. Mosses (1993) surveys the usage of sorts in first-order algebraic specification frameworks, discussing order-sorted algebra and partial functions. Mitchell (1990) gives a survey of type systems for programming languages. Cardelli (1997) provides a more informal introduction to type systems.

Typechecking in ASF+SDF The specification formalism ASF+SDF has been applied to the description or design of several languages. We give some pointers to papers that describe specifications of type systems similar to the one described in this chapter. Hendriks (1989) describes (in the first ASF+SDF specification) the polymorphic type inference algorithm of Milner (1978) in the language Mini-ML. Van Deursen (1996) describes the specification of a typechecker for Pascal. Hillebrand and Korver (1995) give a specification of the well-formedness of μCRL specifications. μCRL is a process specification formalism with a monomorphic algebraic specification language for the specification of data in processes. Vigna (1995, 1996) specifies a typechecker and compiler for the categorical programming language IMP(G). A special feature of the language is the associativity of the built-in type constructors $\times$ and $+$. The typechecker makes extensive use of list matching in ASF+SDF to handle this associativity. In full MLS, associativity of type constructors can be expressed by means of equations over types like $A \times (B \times C) = (A \times B) \times C$. Type checking such specifications requires $\mathcal{E}$-unification.

Polymorphic Typechecking The type inference algorithm of Milner (1978), also described in Damas and Milner (1982), forms the core of all typecheckers for polymorphic languages. The basic idea of that algorithm is also used in the type assignment of terms in multi-level specifications. Although Milner (1978) mentions overloading as a possible orthogonal extension of his type inference algorithm, such an extension is not described in the literature. Ambiguities due to overloading in pure Hindley/Milner systems are difficult to resolve if no restriction on the type(s) of functions is given by means of a signature, because then each occurrence of a function can have a different type. The overloading that is achieved by means of type classes (Wadler and Blott, 1989), or more generally, qualified types (Jones, 1992), is actually not overloading in the sense used in this chapter. Rather, type classes provide the means to restrict the set of types over which the universal quantifier in the type of a polymorphic function ranges and they give an account of 'non-parametric' function definitions of such restricted polymorphic functions.

Types in Algebraic Specification The basic type system of monomorphic many-sorted algebraic specification is explained in any introduction to algebraic specification or universal algebra, see for instance Wechler (1992). Mosses (1993) surveys the many variations and extensions of monomorphic type systems for algebraic specification. Extensions of many sorted algebraic specification where the space of types is defined by means of an algebraic specification have been studied by various authors (Poigné, 1986, Möller, 1987, Meinke, 1992a). Meinke (1992b) develops a theory for universal algebra in higher types. Meinke (1993) gives the operational semantics of ATLAS via term rewriting and proves its equivalence to the denotational semantics (i.e., initial model).

14.6.2 Extensions

The formalism MLS presented in this chapter is a uniform and simple specification formalism for sophisticated abstract data type specification. Some aspects important for specification and execution of specifications have not yet been attended. We discuss several extensions to the formalism and the issues they raise for further research.

Implicit Functions ATLAS provides *implicit* functions, which entails that functions declared as {`implicit`} do not have to be written explicitly in terms (Hearn and Meinke, 1994, Hearn, 1995). This is used, for instance, to hide the explicitly defined application function for user-defined function types. When used for unary functions, this boils down to chain rules of grammars. For example, by introducing an operator `inc` as

```
inc : nat -> int {implicit}
```

the naturals are embedded in the integers. The equations

```
0    + X == X;
s(X) + Y == s(X + Y)
```

then apply both to naturals and integers. This feature gives rise to infinite ambiguities. Consider the declaration

```
inc  : A -> list(A) {implicit};
(++) : list(A) # list(A) -> list(A) {implicit}
```

Given these declarations we can write lists such as `inc(X) ++ inc(Y) ++ inc(Z)` as `X Y Z`. The problem is that the inclusion operator `inc` is applicable to any term, i.e., we can interpret `X` as `inc(X)`, as `inc(inc(X))`, etc. It is clear that this infinite ambiguity is recurrent and could somehow be represented in a finite manner. How this should be achieved is not clear.

In ATLAS only unary and binary functions can be declared as `implicit`. Implicit constants, which are not allowed in ATLAS, are analogous to empty productions in context-free grammars and make the typechecking problem undecidable. For instance, if we declare

```
empty : list(A) {implicit}
```

then the list `X` can be interpreted as `inc(X)`, as `empty ++ inc(X)`, as `empty ++ inc(X) ++ empty`, etc. The implicit constant can be inserted anywhere and arbitrarily many times in the term.

Grammars as Signatures A generalization of the implicit types of ATLAS and the infix functions of MLS is the use of arbitrary mixfix function declarations. The motivation for the development of MLS is to prepare the extension of the syntax definition formalism SDF of Heering *et al.* (1989). In SDF, context-free grammars are used as monomorphic algebraic signatures, providing flexible notation for functions and constructors. Like normal monomorphic algebraic signatures, SDF does not support polymorphism nor higher-order functions.

The first step towards an extended SDF is made in Part II, where the design of Heering *et al.* (1989) is rationalized by orthogonally defining its features such that the formalism can be seen as an instance of a family of formalisms. A syntax definition formalism can be created by choosing a set of features. Many features are expressed as conservative extensions of pure context-free grammars by normalizing extended grammars to context-free grammars. As part of this approach, the disambiguation of ambiguous context-free grammars by means of priorities is seen as an instance of a more general view of disambiguation by means of disambiguation filters — functions that select a subset of a set of possible parse trees — see Chapter 4.

In the definition of MLS we have abstracted from the use of grammars as signatures in order to get a clear picture of a multi-level type system without the complications caused by grammars. It is clearly desirable to extend MLS with arbitrary mix-fix operators and disambiguation capabilities like priorities to enhance the notation defined in signatures. However, the generalization of multi-level specifications to multi-level grammars is not straightforward if arbitrary grammars are allowed. The addition of chain and empty productions to signatures makes the parsing problem undecidable in general. Such rules are the cause of infinite ambiguities (sentences can have infinitely many parses) already in context-free grammars. However, in multi-level grammars the set of all parses for a sentence might not be finitely representable. Due to overloading, terms in MLS can have more than one full annotation (the analogon of a parse tree), but always finitely many.

In Chapter 15 the extension of context-free grammars to two-level grammars and the correspondence of two-level grammars with two-level first-order signatures are studied for the purpose of polymorphic syntax definition — polymorphic notation for algebraic specification. There a restriction on multi-level grammars is formulated that guarantees that parsing is decidable.

Type Equations Type equations are not interpreted by the type assignment algorithm presented in this chapter. This is a pity, because many type features from programming languages and abstract data types can be expressed in MLS by means of type equations. In §13.6.2 the generalization of the `zip` function to tuples of lists is defined by means of functions at the level of types (the map function `(*)`). In §13.6.6 type classes are expressed as type predicates. In the same way the more general qualified types of Jones (1992) can be expressed. There are many other applications of type equations. *Type definitions* of the form

```
parser(A, B) == list(A) -> (B # list(A))
```

can be used to define a type in terms of other types. The original constructor can be eliminated. *Recursive type definitions* of the form

```
list(A) == empty + (A # list(A))
```

can be used to define recursive types. These type constructors can not be eliminated, because the unfolding of the type results in an infinite term. The *associative type constructors* of Vigna (1995, 1996) can be expressed by the equations

```
A # (B # C) == (A # B) # C;
A + (B + C) == (A + B) + C
```

Jones (1992) also discusses *record types* as a special case of qualified types by providing operations for looking up the type of a field in, and for removing a field from a record type.

Simple type definitions can be accounted for by rewriting. For the other cases of type equations $\mathcal{E}$-unification is required. $\mathcal{E}$-unification is undecidable in general [see Jouannaud and Kirchner (1991) for a survey of unification]. However, if the equations are known to belong to a certain class, a solution strategy based on that knowledge might be found. For instance, a simple approach to $\mathcal{E}$-unification led to a unification algorithm that terminates for the unification of the types in the generalization of the `zip` function in §13.6.2 (see Visser (1996b)). All the other examples of type equations mentioned above are embedded in the typechecking of various programming languages. These typecheckers thus use some kind of $\mathcal{E}$-unification optimized for the special case. For instance, Nipkow and Prehofer (1995) describe a typechecking algorithm for type classes in terms of unification with constraint resolution. If a union of these solutions exists such that many cases of type equations can be dealt with more generically, MLS provides an expressive framework for specification of advanced type systems.

Modules The formalism has a rudimentary modularization scheme based on syntactic inclusion, i.e., imports are expanded before typechecking. Is it possible to keep the module structure while typechecking? Furthermore, consider using arbitrary terms as module names. An import of a module name provides a term that is at least as specific as a module name. The parameters of the module are determined by matching the actual module name against the declared module name. Function renaming operators applicable to imports would be another useful extension.

Rewriting A first experiment has been conducted with translating the level 0 equations of a multi-level specification to the first-order rewrite rule language of the Epic term rewrite compiler of Walters and Kamperman (1996). Terms are translated to first-order terms by keeping the same term structure as in the specification, i.e., terms are built by application, pairing, product, arrow and annotation from functions and variables. Research issues here include: When are annotations necessary? The translation is correct for the subset of MLS that uses only free type constructors in declarations. If type equations are allowed, rewriting with type annotations is complicated because matching has to consider type equations. Can this be expressed in the rewrite system itself?

14.6.3 Conclusions

In this chapter we have defined the syntax, semantics and type system of the modular, applicative, multi-level equational specification formalism MLS. Each level of an MLS specification is an applicative equational specification that uses terms over the next level as types. This is a generalization of type systems with two and three levels that have separate definitions for each level. The type system of MLS is orthogonal and uniform (typechecking is the same for each level) and combines parametric polymorphism with overloading. These features form a formalism for the definition of advanced generic data types.

The formalism is completely specified in ASF+SDF. The Meta-Environment made it possible to interactively experiment with design choices and develop the formalism and its prototype implementation in a short period of time (about four months). The typesetting and literate programming facilities provided by the Meta-Environment played an important role in the design process. This chapter demonstrates a number of specification techniques applicable in other specifications, including innermost term rewriting, the separation of well-formedness rules and non-wellformedness rules producing descriptive error messages, type assignment by annotation, module import normalization, and a library of functions on terms, such as sets, substitution, unification and matching.

One of the shortcomings of ASF+SDF is the poor reusability of specifications, due to a lack of abstraction features such as polymorphism and parameterized modules. If ASF+SDF would be equipped with the higher-order functions and polymorphism of MLS, specifications could reuse more standard data types directly. On the other hand, MLS does not provide the syntax definition support of SDF. A formalism that combines the notational facilities of SDF with the typing facilities of MLS into Multi-Level ASF+SDF, will be a powerful tool for designing and prototyping languages.

Part IV

Polymorphic Syntax Definition

15

Polymorphic Syntax Definition

Context-free grammars are used in several algebraic specification formalisms instead of first-order signatures for the definition of the structure of algebras, because grammars provide better notation than signatures. The rigidity of these first-order structures enforces a choice between strongly typed structures with little genericity or generic operations over untyped structures. In two-level signatures level 1 defines the algebra of types used at level 0 providing the possibility to define polymorphic abstract data types. Two-level grammars are the grammatical counterpart of two-level signatures. This chapter discusses the correspondence between context-free grammars and first-order signatures, the extension of this correspondence to two-level grammars and signatures, examples of the usage of two-level grammars for polymorphic syntax definition, a restriction of the class of two-level grammars for which the parsing problem is decidable, a parsing algorithm that yields a minimal and finite set of most general parse trees for this class of grammars, and a proof of its correctness.

15.1 Introduction

In the algebraic approach to programming language specification, languages are considered as algebras. A sentence, program or expression in a language is an object of its algebra. The constructs for composition of expressions from smaller expressions and the operations that interpret, translate, transform or analyze expressions are the operations of the algebra. Algebraic specifications describe algebras by means of a finite structure that describes the sorts of the algebra, its operations and the relations between the operations. Any algebra that has the structure prescribed by the specification and that satisfies its relations is a model of the specification. Therefore, a specification always describes a class of algebras instead of precisely the intended algebra. There are many formalisms for algebraic specification. Depending on the expressive power of a formalism the class of algebras described by a specification can be narrowed down to the intended algebra. First-order algebraic specifications consist of a first-order signature and a set of equations over the terms generated by the signature. A first-order signature consists of a finite set of sorts and a finite number of operations over those sorts.

Grammars describe languages by means of a finite structure that describes the syntactic categories of a language and the sentences of its categories. Context-free grammars and first-order signatures generate the same class of algebras. Parse trees or abstract syntax trees can be considered as terms over a signature and the language of terms over a signature can be described by a context-free grammar (Hatcher and Rus, 1976, Goguen *et al.*, 1977). This correspondence is exploited in several algebraic specification formalisms by allowing the use of signatures with mix-fix operators (Futatsugi *et al.*, 1985, Bidoit *et al.*, 1989) or even arbitrary context-free grammars (Heering *et al.*, 1989) instead of just prefix function signatures. This provides concrete notation for functions and constructors in data type specifications and it enables definition of operations on programming languages directly in their syntactic constructs.

The rigidity of first-order signatures and context-free grammars makes it difficult to generically describe properties of an algebra. For example, an algebra with lists of integers and lists of strings can be specified with a first-order signature by declaring a sort LI (list of integers) and a sort LS (list of strings) and by defining operations like the empty list, cons, head, tail and concatenation on both sorts. However, if these list sorts have the same properties independent of the contents of the lists for some operations, this can not be expressed in a first-order specification. Similarly, if for both list sorts an operation exists that applies a function to each element of a list, this can not be expressed in a generic way in a first-order specification. This lack of genericity makes it difficult to develop libraries with specifications of common data types and generic language constructs.

A higher type algebra (Meinke, 1992b) is an algebra with an algebraic structure imposed on the set of sorts, i.e., the set of sorts is itself an algebra with operations. These sort operators are interpreted as functions from collections of carrier sets to collections of carrier sets. For instance, the sorts LI and LS above can be seen as sorts constructed from the sorts I (integer) and S (string) by the sort operator L that constructs the sort of sequences of integers and strings, respectively. In such algebras more generic statements about (classes of) objects and operations of the algebra can be made. For example, one can say that, for an arbitrary sort x, the tail function is a function from Lx to Lx that yields the argument sequence without its first element, where we abstract from the fact that x is equal to I or S. One could say that higher type algebras provide a higher resolution in the sort space of an algebra. Algebraic specifications in higher types (Poigné, 1986, Möller, 1987, Meinke, 1992a, Hearn and Meinke, 1994, Visser, 1996a) describe higher type algebras by means of two (or more) levels of signatures. Each level specifies the sort operations for the next level, i.e., the terms over the signature at level $i + 1$ are the sort expressions of the signature at level i. Sort expressions with variables are polymorphic sorts that describe all sorts obtained by substituting sorts for the variables. Polymorphic sorts are used to specify polymorphic functions that uniformly apply to many sorts.

In this chapter we discuss polymorphic syntax definition by means of context-free and two-level grammars. We argue that the grammatical counterpart of algebraic specifications with two-levels are two-level grammars. This correspondence can be extended to multi-level signatures leading to multi-level grammars. The connections between the various formalisms are summarized by the following diagram:

$$\begin{tikzcd}
\boxed{\text{First-Order Signature}} \arrow[r, "(3)"'] \arrow[d, leftrightarrow, "(2)"] & \boxed{\text{Two-Level Signature}} \arrow[r, "(4)"'] \arrow[d, leftrightarrow, dashed] & \boxed{\text{Multi-Level Signature}} \arrow[d, leftrightarrow, dashed] \\
\boxed{\text{Context-Free Grammar}} \arrow[r, "(1)"] & \boxed{\text{Two-Level Grammar}} \arrow[r, dashed] & \boxed{\text{Multi-Level Grammar}}
\end{tikzcd}$$

where we refer to the following literature: (1) van Wijngaarden *et al.* (1976), Pereira and Warren (1980) (2) Hatcher and Rus (1976), Goguen *et al.* (1977), Futatsugi *et al.* (1985), Heering *et al.* (1989) (3) Poigné (1986), Meinke (1992b, 1992a), Hearn and Meinke (1994) (4) Hearn (1995), Visser (1996a).

The rest of this chapter is structured as follows. §15.2 contains a review of first-order signatures, context-free grammars and their correspondence and gives some examples of data type specification with context-free grammars. §15.3 defines two-level grammars and the parsing problem for two-level grammars. §15.4 illustrates how two-level grammars can be used for polymorphic syntax definition. §15.5 discusses several properties of two-level grammars including a characterization of a large class of grammars for which the parsing problem is decidable, although membership of the class is undecidable. §15.6 defines a parsing algorithm, with a correctness proof, for this class of two-level grammars that yields for each string a minimal and finite representation of the set of all parse trees for the string. §15.7 discusses related grammar formalisms and type systems and §15.8 concludes the chapter.

15.2 Signatures and Grammars

In this section we review many-sorted algebras, context-free grammars, the correspondence between first-order signatures and context-free grammars and the use of context-free grammars in the algebraic specification of languages and data types.

15.2.1 Many-Sorted Algebra

Many-sorted algebras or Σ-algebras were introduced by Higgins (1963) as a generalization of the theory of abstract algebra. Here we give the basic constructs needed in this paper. For a further introduction to the theory of universal algebra see for instance Meinke and Tucker (1992), who also give several example applications. A note on notation: We will frequently use the notion of a family, which is a collection of sets indexed by some, finite or infinite, index set. If F is a family indexed by I, we denote by $F(i)$ the set at index $i \in I$ and write $F = (F(i) \mid i \in I)$. If we want to indicate that x is an element of some $F(i)$, we loosely write $x \in F$ identifying F with $\bigcup_{i \in I} F(i)$.

Definition 15.2.1 (Signature) A *many-sorted signature* Σ is a pair $\langle S, F \rangle$ where $S = \mathrm{S}(\Sigma) \subseteq \mathbf{S}$ is a set of sort names and $F = \mathrm{F}(\Sigma) \subseteq \mathbf{O} \times \mathrm{S}(\Sigma)^+$ a set of function declarations (with $\mathbf{S}$ and $\mathbf{O}$ some sets of sort names and operation names, respectively). We write $f : \tau_1 \times \cdots \times \tau_n \to \tau_0$ if $\langle f, \tau_1 \ldots \tau_n \tau_0 \rangle \in \mathrm{F}(\Sigma)$. $\Sigma \cup \mathrm{V}$ is the extension of a signature Σ with a $\mathrm{S}(\Sigma)$-indexed family of sets of variables V. We write $x : \tau$ if $x \in \mathrm{V}(\tau)$. The class of all signatures is denoted by SIG.

Definition 15.2.2 (Σ-Algebra) A Σ-*algebra* $\mathcal{A}$ is an $\mathrm{S}(\Sigma)$-indexed family of carrier sets $A(\tau)$ and an assignment of each $f : \tau_1 \times \cdots \times \tau_n \to \tau_0$ in $\mathrm{F}(\Sigma)$ to an $\mathcal{A}$ function $f_{\mathcal{A}} : A(\tau_1) \times \cdots \times A(\tau_n) \to A(\tau_0)$ such that $f_{\mathcal{A}}(a_1, \ldots, a_n) \in A(\tau_0)$ if $a_i \in A(\tau_i)$ $(1 \leq i \leq n)$. $\mathrm{Alg}(\Sigma)$ denotes the collection of all Σ-algebras.

An equational many-sorted algebraic specification consists of a signature and a set of equations that define the relations between objects of the algebras described by the specification. Note that the theory of universal algebra does not limit algebras to have finitely many operations or sorts, but that an algebraic specification must be a finite structure. The following example illustrates the definitions above. We use the keywords `sorts`, `functions` and `variables` to indicate the declaration of $\mathrm{S}(\Sigma)$, $\mathrm{F}(\Sigma)$ and V, respectively. We write `#` for $\times$ and `->` for $\to$. Furthermore, we make use of modular specifications consisting of modules that can import other modules, where a module with imports denotes the pointwise union of the imported and importing specification.

Example 15.2.3 The following is an example of a first-order algebraic specification of the algebra of natural numbers.

```
module naturals
  sorts nat;
  functions
    zero : nat;
    succ : nat -> nat;
    add  : nat # nat -> nat;
  variables
    I, J : nat;
  equations
    add(zero, I)    = I;
    add(succ(I), J) = succ(add(I, J))
```

Definition 15.2.4 (Terms) The $\mathrm{S}(\Sigma)$-indexed family $\mathrm{Tree}(\Sigma)$ of *well-formed terms (or trees) over signature* Σ is defined by the inference rules below such that $t \in \mathrm{Tree}(\Sigma)(\tau)$ iff $\Sigma \vdash t : \tau$.

$$\frac{x \in \mathrm{V}(\tau)}{\Sigma \cup \mathrm{V} \vdash x : \tau} \tag{Var1}$$

$$\frac{f : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathrm{F}(\Sigma),\ \Sigma \vdash t_i : \tau_i\ (1 \leq i \leq n)}{\Sigma \vdash f(t_1, \ldots, t_n) : \tau} \tag{App1}$$

Definition 15.2.5 (Homomorphism) A Σ-*homomorphism* $h : \mathcal{A} \to \mathcal{B}$ is an $\mathrm{S}(\Sigma)$-indexed family of functions h_τ such that for any $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathrm{F}(\Sigma)$, $h_\tau(f_{\mathcal{A}}(a_1, \ldots, a_n)) = f_B(h_{\tau_1}(a_{\tau_1}), \ldots, h_{\tau_n}(a_n))$. A Σ-algebra $\mathcal{A}$ is *initial* in $\mathrm{Alg}(\Sigma)$ if for any $\mathcal{B} \in \mathrm{Alg}(\Sigma)$ there is a unique homomorphism from $\mathcal{A}$ to $\mathcal{B}$.

Because there is a unique homomorphism $h_{\mathcal{A}} : \mathrm{Tree}(\Sigma) \to \mathcal{A}$ for any $\mathcal{A} \in \mathrm{Alg}(\Sigma)$, i.e., $h_{\mathcal{A}}(f(t_1, \ldots, t_n)) = f_{\mathcal{A}}(h_{\mathcal{A}}(t_1), \ldots, h_{\mathcal{A}}(t_n))$, we have

Proposition 15.2.6 $\mathrm{Tree}(\Sigma)$ *is an inital algebra in* $\mathrm{Alg}(\Sigma)$.

Definition 15.2.7 (Substitution) A *substitution* $\sigma : \mathrm{V} \to \mathrm{Tree}(\Sigma \cup \mathrm{V})$ is a $\mathrm{S}(\Sigma)$-indexed function mapping variables to terms. The function $\bar{\sigma} : \mathrm{Tree}(\Sigma \cup \mathrm{V}) \to \mathrm{Tree}(\Sigma \cup \mathrm{V})$ is the homomorphic extension of a substitution σ that replaces all variables in a term by their σ images. A term t is an *instance* of term t' or t' is *more general* than t, written as $t' \geq t$, if there is some substitution σ such that $t = \bar{\sigma}(t')$. A term t is *strictly more general* than t', $t > t'$, if $t \geq t'$ and not $t' \geq t$. In case $t' \geq t$ we also say that t *matches* t' and that σ is the match. A substitution σ is a *unifier* for two terms t and t' if $\bar{\sigma}(t) = \bar{\sigma}(t')$. A unifier σ is a *most general unifier* for t and t' if for each unifier σ' we have that $\bar{\sigma}(t) \geq \bar{\sigma}'(t)$. A substitution σ is a *renaming* of t if $\bar{\sigma}(t) \geq t$, i.e., if $range(\sigma|vars(t)) \subseteq \mathrm{V}$, with $vars(t)$ the set of variables in t. Two terms t and t' are equal up to renaming of variables ($t \doteq t'$) if there is a renaming σ such that $\sigma(t) = t'$.

The $\mathrm{S}(\Sigma)$-indexed family of equations of an algebraic specification with signature Σ is a subfamily of the family $\mathrm{Eq}(\Sigma \cup \mathrm{V})$ such that $\mathrm{Eq}(\Sigma \cup \mathrm{V})(\tau) \subseteq \mathrm{Tree}(\Sigma \cup \mathrm{V})(\tau)^2$. A Σ-algebra $\mathcal{A}$ satisfies an equation $t_1 = t_2$, if for any substitution σ, $h \circ \bar{\sigma}(t_1) = h \circ \bar{\sigma}(t_2)$ in $\mathcal{A}$, where h is the unique homomorphism $h : \mathrm{Tree}(\Sigma) \to \mathcal{A}$.

15.2.2 Context-Free Grammars

Context-free grammars can be used to define languages, i.e., sets of strings and analyses of strings in the form of parse trees. The structure of parse trees corresponds to the structure of terms over a signature as we shall see in the next subsection. However, grammars provide more flexible notation for terms.

Definition 15.2.8 (Grammar) A *context-free grammar* $\mathcal{G}$ is a triple $\langle S, L, P \rangle$ with $S = \mathrm{Ss}(\mathcal{G})$ a finite set of *sort symbols* or *nonterminals*, $L = \mathrm{Sl}(\mathcal{G})$ a finite set of *literals* or *terminals*, with $\mathrm{Ss}(\mathcal{G}) \cap \mathrm{Sl}(\mathcal{G}) = \emptyset$ and $\mathrm{S}(\mathcal{G}) = \mathrm{Ss}(\mathcal{G}) \cup \mathrm{Sl}(\mathcal{G})$ the set of *symbols* of $\mathcal{G}$, and $P = \mathrm{P}(\mathcal{G}) \subseteq \mathrm{S}(\mathcal{G})^* \times \mathrm{Ss}(\mathcal{G})$ a finite set of *productions*. We write $\alpha \to \tau$ for a production $\langle \alpha, \tau \rangle \in \mathrm{P}(\mathcal{G})$. $\mathcal{G} \cup \mathrm{V}$ is the extension of a grammar with variables. We write $x \to \tau$ if $x \in \mathrm{V}(\tau)$. The class of all context-free grammars is denoted by CFG.

Observe that productions are reversed in order to make them look like function declarations in a signature—conventionally a production $\alpha \to \tau$ is written as $\tau \to \alpha$ or $\tau ::= \alpha$. Also note that in the conventional definition of context-free grammars a single symbol has the role of start symbol from which all sentences of the grammar are generated. In the definition above all sort symbols are start symbols. Rus and Jones (1995) make a distinction between context-free grammars that have a single start symbol or *axiom*, *algebraic grammars* that have all nonterminals as start symbol, and *multi-axiom grammars* with a subset of the nonterminals as start symbol. In that terminology our grammars might more appropriately be called algebraic grammars. However, in our definition of language generated by a grammar (below), we distinguish the sets generated by each nonterminal, whereas in the definition of Rus and Jones (1995) the language of a grammar is the union of all strings generated by all axioms, weakening the expressive power of the formalism. With Goguen *et al.* (1977) we stick with the familiar 'context-free grammar'.

As concrete syntax for grammars in examples we adopt the style of the syntax definition formalism SDF (Heering *et al.*, 1989). The keywords `sorts`, `syntax` and `variables` indicate the sets of sort symbols, context-free productions and variables declarations, respectively. Strings of characters between double quotes represent the literals of the grammar and identifiers are used as sort symbols. The sort symbols are explicitly declared in the sorts section, whereas literals are implicitly declared by their usage in productions. Grammars can be divided in modules and modules can import other modules. A module with imports denotes the pointwise union of the imported and importing grammar.

Example 15.2.9 The following specification uses a context-free grammar as signature in the specification of succesor naturals. This specification is similar to the specification in Example 15.2.3, but in the equations we can use the more natural infix notation familiar from mathematics.

```
module naturals-cfg
  sorts nat;
  syntax
    "0"          -> nat;
    "s" nat      -> nat;
    nat "+" nat -> nat {left};
    "(" nat ")" -> nat {bracket};
  variables
    "I" -> nat; "J" -> nat;
  equations
    0    + I = I;
    s(I) + J = s(I + J)
```

The attributes attached to the productions are meant for disambiguation. The attribute `left` indicate the left associativity of the addition function and the attribute `bracket` indicates that parentheses around a natural number behave as the identity function. Disambiguation will be further discussed below.

Definition 15.2.10 (Parse Trees) The $\mathrm{S}(\mathcal{G})$-indexed family $\mathrm{Tree}(\mathcal{G})$ of *parse trees over grammar* $\mathcal{G}$ is defined by the inference rules below such that $t \in \mathrm{Tree}(\mathcal{G})(\tau)$ iff $\mathcal{G} \vdash t : \tau$.

$$\frac{L \in \mathrm{Sl}(\mathcal{G})}{\mathcal{G} \vdash L : L} \tag{Lit2}$$

$$\frac{x \in \mathrm{V}(\tau)}{\mathcal{G} \cup \mathrm{V} \vdash \mathrm{var}(x, \tau) : \tau} \tag{Var2}$$

$$\frac{\tau_1 \ldots \tau_n \to \tau \in \mathrm{P}(\mathcal{G}),\ \mathcal{G} \vdash t_i : \tau_i\ (1 \leq i \leq n)}{\mathcal{G} \vdash \mathrm{app}(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n]) : \tau} \tag{App2}$$

Example 15.2.11 As an example of this inference relation consider the following parse tree for the sentence `0 + I` over the grammar of Example 15.2.9.

app(`nat "+" nat` $\to$ `nat`, [app(`"0"` $\to$ `nat`, [`"0"`]), `"+"`, [var(`"I"`, `nat`)]])

Rule (App2) defines the construction of *application* tree nodes for productions of a grammar. Observe that the complete production is used as label in such application nodes.

Because the structure of parse trees is different from terms over a signature, we redefine the notion of substitution.

Definition 15.2.12 (Substitution) A substitution $\sigma : (\mathrm{V} \times \mathrm{S}(\mathcal{G})) \to \mathrm{Tree}(\mathcal{G} \cup \mathrm{V})$ is a $\mathrm{S}(\mathcal{G})$-indexed family of functions mapping variables to trees. The extension $\bar{\sigma}$ of σ to trees is defined as

$$\begin{aligned}
\bar{\sigma}(L) &= L \\
\bar{\sigma}(\tau)(\mathrm{var}(x, \tau)) &= \sigma(\mathrm{var}(x, \tau)) \\
\bar{\sigma}(\tau)(\mathrm{app}(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n])) &= \\
&\mathrm{app}(\tau_1 \ldots \tau_n \to \tau, [\bar{\sigma}(\tau_1)(t_1), \ldots, \bar{\sigma}(\tau_n)(t_n)])
\end{aligned}$$

All other notions defined in Definition 15.2.7 are defined in the same way for parse trees.

Definition 15.2.13 (Language) The *language* $\mathrm{L}(\mathcal{G})$ generated by a context-free grammar $\mathcal{G}$ is the $\mathrm{S}(\mathcal{G})$-indexed family of strings such that $\mathrm{L}(\mathcal{G})(\tau) = \mathrm{yield}(\mathrm{Tree}(\mathcal{G})(\tau))$, where the function $\mathrm{yield} : \mathrm{Tree}(\mathcal{G} \cup \mathrm{V}) \to (\mathrm{S}(\mathcal{G}) \cup \mathrm{V})^*$ is defined by

$$\begin{aligned}
\mathrm{yield}(L) &= L \\
\mathrm{yield}(\mathrm{var}(x, \tau)) &= x \\
\mathrm{yield}(\mathrm{app}(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n])) &= \mathrm{yield}(t_1) \ldots \mathrm{yield}(t_n)
\end{aligned}$$

and applied to a set of trees denotes the pointwise extension to sets.

Definition 15.2.14 (Parsing) A parser for a context-free grammar $\mathcal{G}$ is a function $\Pi(\mathcal{G}) : \mathrm{S}(\mathcal{G})^* \to \mathcal{P}(\mathrm{Tree}(\mathcal{G}))$ that maps a string of symbols to a subfamily of $\mathrm{Tree}(\mathcal{G})$ such that

$$\Pi(\mathcal{G})(w)(\tau) = \{t \in \mathrm{Tree}(\mathcal{G})(\tau) \mid \mathrm{yield}(t) = w\}$$

A recognizer is a predicate $\in \mathrm{L}(\mathcal{G})$ that decides whether a string is in the language generated by $\mathcal{G}$ or more specifically a predicate $\in \mathrm{L}(\mathcal{G})(\tau)$ that decides whether a string is in the language generated by sort symbol τ.

15.2.3 Correspondence of Signatures and Grammars

There is a correspondence between the trees generated by first-order signatures and context-free grammars such that grammars can be used to describe the structure of algebras (Hatcher and Rus, 1976, Goguen *et al.*, 1977, Heering *et al.*, 1989).

Proposition 15.2.15 *There are mappings* $\mathrm{grm} : \mathrm{SIG} \to \mathrm{CFG}$ *and* $\mathrm{sig} : \mathrm{CFG} \to \mathrm{SIG}$ *such that* $\mathrm{Tree}(\mathrm{grm}(\Sigma)) \cong \mathrm{Tree}(\Sigma)$ *and* $\mathrm{Tree}(\mathrm{sig}(\mathcal{G})) \cong \mathrm{Tree}(\mathcal{G})$.

Proof. Define grm such that for a signature Σ a grammar is constructed that expresses the syntax of terms over a signature Σ by taking as nonterminals the sorts of Σ and as literals the operator symbols of Σ, parentheses and commas.

$$\begin{aligned}
\mathrm{Ss}(\mathrm{grm}(\Sigma)) &= \mathrm{S}(\Sigma)\\
\mathrm{Sl}(\mathrm{grm}(\Sigma)) &= \{\texttt{"f"} \mid f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathrm{F}(\Sigma)\} \cup \{\texttt{"("}, \texttt{")"}, \texttt{","}\}\\
\mathrm{P}(\mathrm{grm}(\Sigma)) &= \{\texttt{"f""("}\tau_1\texttt{","}\ldots\texttt{","}\tau_n\texttt{")"} \to \tau_0\\
&\qquad\qquad\qquad \mid f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathrm{F}(\Sigma)\}
\end{aligned}$$

Now we can translate terms over Σ to parse trees over $\mathrm{grm}(\Sigma)$ by means of the function $i_{\mathrm{grm}} : \mathrm{Tree}(\Sigma) \to \mathrm{Tree}(\mathrm{grm}(\Sigma))$ as follows:

$$\begin{aligned}
i_{\mathrm{grm}}(\tau)(f(t_1, \ldots, t_n)) = \mathrm{app}(&\texttt{"f""("}\tau_1\texttt{","}\ldots\texttt{","}\tau_n\texttt{")"} \to \tau_0,\\
&[\texttt{"f""("}i_{\mathrm{grm}}(\tau_1)(t_1)\texttt{","}\ldots\texttt{","}i_{\mathrm{grm}}(\tau_n)(t_n)\texttt{")"}])
\end{aligned}$$

for each $f : \tau_1 \times \cdots \times \tau_n \to \tau \in \mathrm{P}(\Sigma)$. Define sig such that a grammar is translated to a signature in which the productions of the grammar have the role of function names.

$$\begin{aligned}
\mathrm{S}(\mathrm{sig}(\mathcal{G})) &= \mathrm{S}(\mathcal{G})\\
\mathrm{F}(\mathrm{sig}(\mathcal{G})) &= \{\texttt{"}\alpha \to \tau\texttt{"} : \tau_1 \times \cdots \times \tau_n \to \tau \mid \alpha \to \tau \in \mathrm{P}(\mathcal{G}), \alpha = \tau_1 \ldots \tau_n\}\\
&\cup\ \{\texttt{"L"} :\to L \mid L \in \mathrm{Sl}(\mathcal{G})\}
\end{aligned}$$

Now we can translate parse trees to terms by means of the function $i_{\mathrm{sig}} : \mathrm{Tree}(\mathcal{G}) \to \mathrm{Tree}(\mathrm{sig}(\mathcal{G}))$ as follows:

$$
\begin{aligned}
i_{\mathrm{sig}}(L)(L) &= L() \\
i_{\mathrm{sig}}(\tau)(\mathrm{var}(x,\tau)) &= x \\
i_{\mathrm{sig}}(\tau)(\mathrm{app}(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n])) &= \\
&\texttt{"}\tau_1 \ldots \tau_n \to \tau\texttt{"}(i_{\mathrm{sig}}(\tau_1)(t_1), \ldots, i_{\mathrm{sig}}(\tau_n)(t_n))
\end{aligned}
$$

It is clear that i_{grm} and i_{sig} are isomorphisms.[1] □

The following proposition tells us that we can use context-free grammars as many-sorted algebraic signatures, where productions play the role both of function symbol and type declaration. We can thus speak of the class of algebras $\mathrm{Alg}(\mathcal{G})$ generated by a context-free grammar $\mathcal{G}$, where the grammar symbols are interpreted as carrier sets and productions as algebraic operations. It is clear that the family $\mathrm{Tree}(\mathcal{G})$ of parse trees over $\mathcal{G}$ is an initial algebra in $\mathrm{Alg}(\mathcal{G})$. The language $\mathrm{L}(\mathcal{G})$ is also an element of $\mathrm{Alg}(\mathcal{G})$, with yield as the unique homomorphism from $\mathrm{Tree}(\mathcal{G}) \to \mathrm{L}(\mathcal{G})$. However, $\mathrm{L}(\mathcal{G})$ is not necessarily initial in $\mathrm{Alg}(\mathcal{G})$. A context-free grammar is ambiguous if there is some string $w \in \mathrm{L}(\mathcal{G})$ for which more than one parse tree exists.

Proposition 15.2.16 $\mathrm{L}(\mathcal{G})$ *is initial in* $\mathrm{Alg}(\mathcal{G})$ *iff* $\mathcal{G}$ *is unambiguous.*

For if $\mathcal{G}$ is ambiguous, yield is not injective, hence not an isomorphism. This entails that algebraic properties do not apply to the strings used to denote trees. For example, in a grammar of arithmetic expressions with the production e "$-$" $e \to e$, the composition of the strings x, $-$ and $y - z$ does not correspond with the composition of their trees, i.e., $x - (y - z)$, but with $(x - y) - z$, which usually has a different semantic interpretation. We could require the use of unambiguous grammars. However, it is undecidable whether a context-free grammar is ambiguous. There are decidable subclasses of CFG, e.g., the $\mathrm{LR}(k)$ grammars, that are unambiguous, but these classes are much more restrictive than the class of unambiguous grammars and, moreover, not closed under union of grammars, which is a handicap when developing modular specifications. Furthermore, to disambiguate a grammar it is often necessary to introduce new sort symbols and to restrict the possibility to compose expressions.

In Chapter 4 a method for disambiguation of context-free grammars by means of *disambiguation filters* is proposed. A filter $\mathcal{F}(\mathcal{G})$ selects a subset from the parse trees for a string, i.e., $\mathcal{F}(\mathcal{G})(\Pi(\mathcal{G})(w)) \subseteq \Pi(\mathcal{G})(w)$. A filter is completely disambiguating if for each string w, $|\mathcal{F}(\mathcal{G})(\Pi(\mathcal{G})(w))| \leq 1$.

Proposition 15.2.17 *If a filter* $\mathcal{F}(\mathcal{G})$ *is completely disambiguating, then* $\mathcal{F}(\mathcal{G}) \circ \Pi(\mathcal{G})$ *is an injection* $\mathrm{L}(\mathcal{G}) \to \mathrm{Tree}(\mathcal{G})$.

However, the trees that are not selected by the filter become unreachable with this method, i.e., $\mathcal{F}(\mathcal{G}) \circ \Pi(\mathcal{G})$ is not surjective. A solution to this problem is to try to add bracket productions, which are interpreted as identity functions, to the grammar such that all trees become reachable.

[1] Note that grm and sig are not isomorphisms from SIG to CFG and vice versa: $\Sigma \neq \mathrm{sig}(\mathrm{grm}(\Sigma))$ and $\mathcal{G} \neq \mathrm{grm}(\mathrm{sig}(\mathcal{G}))$.

Proposition 15.2.18 *If* $\mathrm{Tree}(\mathcal{G}) \cong \mathrm{Tree}(\mathcal{G} \cup \mathcal{G}_{br}) / =_{br}$ *and* $\mathcal{F}(\mathcal{G} \cup \mathcal{G}_{br})$ *is completely disambiguating, then* $\mathrm{L}(\mathcal{G}) \cong \mathrm{Tree}(\Sigma)$.

For a further discussion of this topic we refer to Chapter 4. In the sequel we will assume that we are dealing with such grammars that we can use strings to denote trees. In examples we use a simple method for disambiguation by priority and associativity declarations. For instance, in the grammar of naturals above we used the production attribute `left` to declare the addition operator as left associative. Furthermore, the `bracket` attribute declares the production `"(" nat ")" -> nat {bracket}` as the identity function on natural numbers and makes all trees in Tree(`nat`) reachable by means of strings.

15.2.4 Data Type Specification

By means of grammars as signatures we have a flexible framework for syntax definition in the algebraic specification of data types, for example, the typical stack constructors might be defined as `"[]" -> stack; "push" int "on" stack -> stack`. In algebraic specification of programming languages, context-free grammars can be used for instance to specify the syntax of a programming language as in `var ":=" exp -> stat` and the syntax of operations on programs such as typecheckers `decl "|-" exp -> bool; decl "|-" stat -> bool` that characterize the well-typed expressions and statements, interpreters `"eval" "[[" stat "]]" "(" env ")" -> env` that interpret statements as functions from environments to environments and compilers `"trans" "[[" stat "]]" -> smc` that translate statements to stack machine code.

The disadvantage of first-order signatures and context-free grammars is the rigid monomorphic typing scheme. For instance, we can not express that for each sort τ, the sort τ^* of sequences of τs can be constructed and that for each function $f : \tau_1 \to \tau_2 \in \mathrm{F}(\Sigma)$ the function $f^* : \tau_1^* \to \tau_2^*$ extends f to sequences such that $f^*(a_1 \ldots a_n) = f(a_1) \ldots f(a_n)$. The consequence is that for each special case of a generic construct such as sequences and for each instance of a generic function such as $\_^*$, a separate definition has to be given.

One solution to overcome this rigidity is to loosen the typing requirements. In Visser (1993) terms of typed combinatory logic are encoded as simple untyped applicative terms. In Van den Brand *et al.* (1997a) a similar structure is defined for the representation of parse trees and other structured data by means of a generic format for term representation. We study a combination of the ideas from those papers. The following grammar of generic terms (aterms) is defined by Van den Brand *et al.* (1997a) to represent parse trees and abstract syntax trees over arbitrary grammars. A term is a function symbol (afun), an application of a function symbol to a list of arguments $F(T_2, \ldots, T_n)$, or a list of terms $[T_1, \ldots, T_n]$. A function symbol is a literal or an identifier. Identifier function symbols have to defined explicitly. The module `literals` that is imported in module `aterms` defines the syntax of literals, i.e., strings of characters between double quotes.

```
module aterms
  imports literals
  sorts aterms atermlist afun aterm
  syntax
    aterm                -> aterms;
    aterm "," aterms     -> aterms;
    "[" "]"              -> atermlist;
    "[" aterms "]"       -> atermlist;
    literal              -> afun;
    afun                 -> aterm;
    afun "(" aterms ")"  -> aterm;
    atermlist            -> aterm;
  variables
    "T" -> aterm; "Ts" -> aterms; "Tl" -> aterms;
```

With this term structure it is possible to define higher-order functions. For instance, the following module defines the function `*` that applies a function F to each element of a list of terms and the function `:` that adds an element to the front of a list. Functions that are passed as arguments to higher-order functions are also represented as terms. The function `@` defines the application of such symbolically represented functions to their arguments.

```
module listops
imports aterms;
  syntax
    aterm ":" atermlist -> atermlist {right};
    aterm "*" atermlist -> aterm {right};
    "map"               -> afun;
    aterm "@" aterm     -> aterm {left};
    "(" aterm ")"       -> aterm {bracket};
  variables
    "Fun" -> aterm;
  equations
    T : []        = [T];
    T : [Ts]      = [T, Ts];
    Fun * []      = [];
    Fun * [T]     = [Fun @ T];
    Fun * [T, Ts] = (Fun @ T) : (F * [Ts]);
    map(Fun) @ T  = Fun * T;
```

Such a definition works well as long as sensible terms are considered. However, `([] * map)`, the empty list mapped over the function `map`, is also a syntactically correct term, but does not have a clear interpretation. We would rather forbid this term on the basis of some typing rule without losing the genericity of the term structure.

One application of the generic term structure of aterms, is the representation of parse trees. We add the following function symbols

```
module atrees
  imports aterms
  syntax
    "var" -> afun; "app" -> afun; "prod" -> afun; "lit" -> afun;
```

The following proposition shows how this language can be used to represent parse trees over arbitrary grammars. Observe that we use the concrete syntax of aterms to represent elements of Tree(atrees).

Proposition 15.2.19 *For any CFG $\mathcal{G}$, there is an injection* $\lceil \_ \rceil : \mathrm{Tree}(\mathcal{G}) \to \mathrm{Tree}(\mathtt{aterms})$ *such that* $\mathrm{Tree}(\mathcal{G})$ *is isomorphic with its* $\lceil \_ \rceil$ *image in* aterms*, i.e.,* $\mathrm{Tree}(\mathcal{G}) \cong \lceil \mathrm{Tree}(\mathcal{G}) \rceil$.

Proof. Given some CFG $\mathcal{G}$ first define $\lceil \_ \rceil : \mathrm{S}(\mathcal{G}) \to \mathrm{Tree}(\mathrm{atrees})$ as

$$\begin{aligned} \lceil L \rceil &= \mathtt{lit}(\texttt{"}L\texttt{"}) \\ \lceil \tau \rceil &= \texttt{"}\tau\texttt{"} \text{ if } \tau \in \mathrm{Ss}(\mathcal{G}) \end{aligned}$$

then define $\lceil \_ \rceil : \mathrm{Tree}(\mathcal{G}) \to \mathrm{Tree}(\mathrm{atrees})$ as

$$\begin{aligned} \lceil L \rceil &= \texttt{"}L\texttt{"} \\ \lceil \mathrm{var}(x, \tau) \rceil &= \mathtt{var}(\texttt{"}x\texttt{"}, \lceil \tau \rceil) \\ \lceil \mathrm{app}(\tau_1 \ldots \tau_n \to \tau_0, [t_1, \ldots, t_n]) \rceil &= \\ &\mathtt{app}(\mathtt{prod}([\lceil \tau_1 \rceil, \ldots, \lceil \tau_n \rceil], \lceil \tau_0 \rceil),\ [\lceil t_1 \rceil, \ldots, \lceil t_n \rceil]) \end{aligned}$$

Now we have $\mathrm{Tree}(\mathcal{G}) \cong \lceil \mathrm{Tree}(\mathcal{G}) \rceil$. □

As a result, any sentence in a context-free language can be represented as a string in the fixed language of aterms preserving the structure assigned to it by the context-free grammar describing the language. For example, the parse tree for the string `s 0` according to the grammar for natural numbers is translated as follows:

$$\lceil \mathrm{app}(\texttt{"s"}\ \mathrm{nat} \to \mathrm{nat}, [\texttt{"s"}\ \mathrm{app}(\texttt{"0"} \to \mathrm{nat}, [\texttt{"0"}])]) \rceil \ =$$

```
app(prod([lit("s"), "nat"], "nat"),
    ["s", app(prod([lit("0")], "nat"), ["0"])])
```

The resulting string does not only have a fixed syntax, it is also self descriptive. The grammar $\mathcal{G}$ can be derived from the aterm that encodes a parse tree. With this encoding we can define very generic, language independent operations on parse trees like substitution, unification and searching of subtrees. Again, the disadvantage of this scheme is that there are (many) aterms that are not encodings of parse trees, e.g., `"abc"("def")` is a syntactically correct aterm but is not an element of $\lceil \mathrm{Tree}(\mathcal{G}) \rceil$ for any $\mathcal{G}$. Therefore, specifications and programs that manipulate aterms encoding parse trees have to type check the terms they receive and have to preserve well-formedness of the terms they process and construct.

15.3 Two-Level Grammars

Context-free grammars provide either a strongly typed but rigid syntactic structure or a generic but untyped structure. Two-level grammars provide a method for polymorphic syntax definition that supports definition of generic structures with type constraints. Two-level grammars have been defined in several variants after the original formulation for the definition of the syntax of Algol68 in van Wijngaarden *et al.* (1976). Here we introduce a definition of two-level grammars that is straightforwardly formulated as two levels of context-free grammars, where level 1 defines the syntax of the nonterminals of level 0. The productions at level 0 of a two-level grammar are *production schemata* that uniformly describe sets of context-free productions in the same way that polymorphic functions in a framework like ML (Milner, 1978) describe collections of functions. Given the extension of context-free grammars to two-level grammars, it is straightforward to generalize two-level grammars to multi-level grammars, in the same way as multi-level specifications are defined in Part III. In this chapter we will restrict our attention to two-level grammars.

Definition 15.3.1 (Two-Level Grammar) A *two-level grammar* Γ is a pair $\langle \mathcal{G}_1, \mathcal{G}_0 \rangle$ of context-free grammars such that the sort symbols of $\mathcal{G}_0$ are terms, possibly with variables, over $\mathcal{G}_1$, i.e., $\mathrm{Ss}(\mathcal{G}_0) \subseteq \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1)$.

The following definition gives the meaning of finite two-level grammars in terms of, possibly infinite, context-free grammars.

Definition 15.3.2 A two-level grammar Γ corresponds to a, possibly infinite, context-free grammar $[\![\Gamma]\!]$ that is derived from Γ by taking all substitutions of symbols $\mathrm{S}([\![\Gamma]\!]) = \{\bar{\sigma}(\tau) \mid \tau \in \mathrm{S}(\mathcal{G}_0),\ \sigma : \mathrm{V}_1 \to \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1)\}$ and productions $\mathrm{P}([\![\Gamma]\!]) = \{\bar{\sigma}(\tau_1) \to \bar{\sigma}(\tau_2) \mid \tau_1 \to \tau_2 \in \mathrm{P}(\mathcal{G}_0),\ \sigma : \mathrm{V}_1 \to \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1)\}\rangle$.

Through the translation of a two-level grammar Γ to a CFG $[\![\Gamma]\!]$ we immediately have the definitions of the term algebra $\mathrm{Tree}([\![\Gamma]\!])$ and the language $\mathrm{L}([\![\Gamma]\!])$. Another characterization of the trees generated by a two-level grammar is given by means of inference rules in the following definition.

Definition 15.3.3 The $\mathrm{S}(\mathcal{G}_0)$-indexed family $\mathrm{Tree}(\Gamma)$ of *parse trees over two-level grammar* Γ is defined by the inference rules below such that $t \in \mathrm{Tree}(\Gamma)(\tau)$ iff $\Gamma \vdash t : \tau$.

$$\frac{L \in \mathrm{Sl}(\mathcal{G}_0)}{\Gamma \vdash L : L} \qquad \text{(Lit3)}$$

$$\frac{x \in \mathrm{V}(\tau'), \tau' \geq \tau}{\Gamma \cup \mathrm{V} \vdash \mathrm{var}(x, \tau) : \tau} \qquad \text{(Var3)}$$

$$\frac{p \in \mathrm{P}(\mathcal{G}_0),\ p \geq \tau_1 \ldots \tau_n \to \tau,\ \Gamma \vdash t_i : \tau_i\ (1 \leq i \leq n)}{\Gamma \vdash \mathrm{app}(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n]) : \tau} \qquad \text{(App3)}$$

Recall from Definition 15.2.7 that the relation $p \geqq p'$ holds if production p' is an instance of p, i.e., p is more general than p'.

We observe that the two ways of defining the terms generated by a two-level grammar are equivalent.

Proposition 15.3.4 $\Gamma \vdash t : \tau$ *iff* $[\![\Gamma]\!] \vdash t : \tau$

Proof. ($\Rightarrow$) by induction on t: (i) if $t = L$ then $[\![\Gamma]\!] \vdash L : L$ by (Lit2) (ii) if $t = \mathrm{var}(x, \tau)$ then $[\![\Gamma]\!] \vdash \mathrm{var}(x, \tau) : \tau$ by (Var2) (iii) if $t = \mathrm{app}(p', [t_1, \ldots, t_n])$, by induction hypothesis $[\![\Gamma]\!] \vdash t_i : \tau_i$, and by (App3) there is some $p \in \mathrm{P}(\mathcal{G}_0)$ such that $p \geqq p'$, i.e., there is some σ such that $\bar{\sigma}(p) = p'$, but then $p' \in \mathrm{P}([\![\Gamma]\!])$, therefore, by (App2), $[\![\Gamma]\!] \vdash \mathrm{app}(p', [t_1, \ldots, t_n])$. ($\Leftarrow$) similarly. □

Corollary 15.3.5 $\mathrm{Tree}([\![\Gamma]\!]) = \mathrm{Tree}(\Gamma)$ *and* $\mathrm{L}(\Gamma) = \mathrm{L}([\![\Gamma]\!])$

Definition 15.3.6 (Substitution) A two-level substitution φ is a pair $\langle \sigma_1, \sigma_0 \rangle$ of a type substitution $\sigma_1 : (\mathrm{V}_1 \times \mathrm{S}(\mathcal{G}_1)) \to \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1)$ and an object substitution $\sigma_0 : (\mathrm{V}_0 \times \mathrm{S}(\mathcal{G}_0)) \to \mathrm{Tree}(\mathcal{G}_0 \cup \mathrm{V}_0)$. The extension $\bar{\varphi}$ of φ to level 0 trees is defined as

$$\bar{\varphi}(L) = L$$

$$\bar{\varphi}(\tau)(\mathrm{var}(x, \tau)) = \sigma_0(\mathrm{var}(x, \bar{\sigma}_1(\tau)))$$

$$\begin{aligned}\bar{\varphi}(\tau)(\mathrm{app}(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n])) = \\ \mathrm{app}(\bar{\sigma}_1(\tau_1) \ldots \bar{\sigma}_1(\tau_n) \to \bar{\sigma}_1(\tau), [\bar{\varphi}(\tau_1)(t_1), \ldots, \bar{\varphi}(\tau_n)(t_n)])\end{aligned}$$

All other notions defined in Definition 15.2.7 are defined in the same way for parse trees. A two-level substitution with σ_0 equal to the identity function is also denoted by $\bar{\bar{\sigma}}$, i.e., a function that substitutes type variables throughout a term.

Definition 15.3.7 (Parsing) Given a two-level grammar Γ and a string w the parsing problem is to find the set of parse trees $\Pi(\Gamma)(w)$ such that

$$\Pi(\Gamma)(w)(\tau) = \{t \mid \Gamma \vdash t : \tau \wedge \mathrm{yield}(t) = w\}$$

Discussion 15.3.8 According to the definition above, *trees* over level 1 are used as sort symbols in level 0. However, if we write such grammars, we want to use strings instead of trees, i.e., $S(\mathcal{G}_0) \subseteq \mathrm{L}(\mathcal{G}_1 \cup \mathrm{V}_1) \cup \mathbf{S}$ instead of $\mathrm{S}(\mathcal{G}_0) \subseteq \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1) \cup \mathbf{S}$. This entails that the syntax of two level grammars is not fixed, the syntax of the symbols of level 0 is determined by level 1. To parse a two-level grammar we first have to parse level 1 with a parser for a context-free grammar formalism in order to construct a parser for level 0. Note that we use the same, SDF style, notation for productions and modules at both levels.

15.4 Examples

In this section we discuss several examples of two-level grammars. The syntax of grammars is the adaptation of the syntax of the multi-level specifications of Part III to grammars, i.e., function declarations become productions. It is not our intention to explain every detail of the notation used, but we do want to illustrate the general utility of two-level grammars for specification of data types.

15.4.1 Naturals

Module `nat` defines the syntax of natural number expressions. Level 1 introduces the sort `type` and the `type` constant `nat`. The expression `nat` can then be used as sort at level 0. Consider for example the production `"s" nat -> nat` of level 0. The expression `nat` in this production, is the constant `"nat" -> type` defined at level 1.

```
module nat
level 1
  sorts type;
  syntax
    "nat" -> type;
level 0
  syntax
    "0"          -> nat;
    "s" nat      -> nat;
    nat "+" nat -> nat {left};
    "(" nat ")" -> nat {bracket};
  variables
    "I" -> nat; "J" -> nat;
  equations
    0    + I = I;
    s(I) + J = s(I + J);
```

15.4.2 Booleans and Polymorphic Conditional

The grammar in module `nat` defines monomorphic syntax for natural numbers. Each production has one instance, i.e., the production itself. The following module defines the data type of Booleans. At level 1 the `type` constant `bool` is introduced, which is used as sort at level 0. In addition to the ordinary Boolean connectives, the module defines a polymorphic conditional for any `type`. The `type` variable `A` in the if-then-else-fi production can be instantiated with any `type` expression. The production actually denotes the set of all instantiations of this production. Furthermore, the module defines a polymorphic bracket function.

```
module bool
imports nat;
level 1
  syntax
    "bool" -> type;
  variables
    "A" -> type; "B" -> type; "C" -> type;
level 0
  syntax
    "true"                           -> bool;
    "false"                          -> bool;
    "not" bool                       -> bool;
     bool "\/" bool                  -> bool {left};
    "if" bool "then" A "else" A "fi" -> A;
    "(" A ")"                        -> A {bracket};
  priorities
    "not" bool -> bool > bool "\/" bool -> bool
  variables
    "B" -> bool; "X" -> A; "X'" -> A;
  equations
    not true   = false;
    not false  = true;
    true  \/ B = true;
    false \/ B = B;

    if true  then X else X' fi = X;
    if false then X else X' fi = X';
```

15.4.3 Polymorphic Lists

Most grammar formalisms provide a built-in notion of lists. The next example shows how such notation can be introduced with two-level grammars. Module `list` introduces type operators at level 1 denoting the type of polymorphic lists. The operators `{_ _}+` and `{_ _}*` denote the type of non-empty and possibly-empty lists with separators, respectively. The operators `"_+"` and `"_*"` denote the type of non-empty and possibly-empty lists without separators, respectively. The latter two operators are defined in terms of the former two by means of the equations that define lists without separators as lists with empty separators, where `empty` is a `sep`.

At level 0 polymorphic constructor functions for these types are defined. A non-empty list of `A`s separated by `Sep`s is either an `A` or two lists concatenated by a `Sep`. The first equation expresses that `Sep` concatenation associates to the right. An `{A Sep}*` list is either empty or a non-empty list of `A`s. `{A Sep}*`-lists can be concatenated by means of the operator `[_ Sep _]`. Note that `"^"` is used as a variable to denote separators.

```
module list
imports bool;
level 1
  sorts regtype, sep;
  syntax
    "{" type sep "}" "*" -> regtype;  type "*" -> regtype;
    "{" type sep "}" "+" -> regtype;  type "+" -> regtype;
    "[" regtype "]"      -> type;     "empty"  -> sep;
  variables
    "Sep" -> sep; "R" -> regtype;
  equations
    A* = {A empty}*; A+ = {A empty}+;
level 0
  syntax
    A                              -> {A Sep}+;
    {A Sep}+ Sep {A Sep}+          -> {A Sep}+ {right};
                                   -> {A Sep}*;
    {A Sep}+                       -> {A Sep}*;
    "[" {A Sep}* Sep {A Sep}* "]"  -> {A Sep}*;
                                   -> empty;
    "(" R ")"                      -> R {bracket};
  variables
    "L" -> {A Sep}*; "Lp" -> {A Sep}+; "^" -> Sep;
  equations
    (Lp1 ^ Lp2) ^ Lp3 = Lp1 ^ (Lp2 ^ Lp3);
    [    ^ L  ]       = L;
    [L   ^    ]       = L;
    [Lp1 ^ Lp2]       = Lp1 ^ Lp2;
```

Observe again how expressions over the syntax defined at level 1 are used as sorts at level 0. For instance, in the production `{A Sep}+ -> {A Sep}*`, the syntax of the expression `{A Sep}+` is defined by the production `"{" type sep "}" "+" -> regtype` and by the variables `"A" -> type` and `"Sep" -> sep`.

We have introduced a new sort `regtype` at level 1 as the sort of list type constructors in order to avoid an infinite chain caused by the injection of arbitrary types in the corresponding list type. If we would have declared the list type constructor as

```
"{" type sep "}" "+" -> type
```

the production `A -> {A Sep}+` would give rise to the productions

```
          {A Sep}+ -> {{A Sep}+ Sep}+
    {{A Sep}+ Sep}+ -> {{{A Sep}+ Sep}+ Sep}+
```

etc., causing each expression to have infinitely many non-unifiable types. By introducing the new sort `regtype`, lists are not automatically embedded in types, i.e., `A` does not unify with `{A Sep}+` because their sorts are different.

The usage of list types is illustrated in the following grammar of a fragment of an imperative language. A statement is either an assignment, a while-do loop or a list of statements separated by semicolons.

```
module while
imports list, exp;
level 1
  syntax
    "var"  -> type; "exp" -> type;
    "stat" -> type; "‘;’" -> sep;
level 0
  syntax
    var ":=" exp              -> stat;
    "while" exp "do" stat     -> stat;
    "begin" {stat ‘;’}* "end" -> stat;
    ";"                       -> ‘;’;
```

The expression `{stat ‘;’}*` is defined by the polymorphic productions in module `list`, which have the following instantiations.

```
  stat                          -> {stat ‘;’}+;
  {stat ‘;’}+ ‘;’ {stat ‘;’}+   -> {stat ‘;’}+;
                                -> {stat ‘;’}*;
  {stat ‘;’}+                   -> {stat ‘;’}*;
```

15.4.4 Polymorphic Operations

Now that we have a polymorphic definition of list construction we can also define polymorphic functions over lists. For instance, the length function that computes the number of elements of a list can be polymorphically defined by the following specification:

```
module length
imports list;
level 0
  syntax
    "length" "(" {A Sep}* ")" -> nat;
  equations
    length()          = 0;
    length(X)         = s(0);
    length(Lp1 ^ Lp2) = length(Lp1) + length(Lp2);
```

15.4.5 Higher-Order Functions

Another example of a type constructor is the arrow `=>` of function types. A term of sort `A => B`, i.e., a function from `A` to `B`, can be applied to a term of sort `A` yielding a `B`.

```
module arrow
imports list;
level 1
  syntax
    type "=>" type -> type {right};
level 0
  syntax
    (A => B) "(" A ")"    -> B;
    (A => B) "*" {A Sep}* -> {B Sep}*;
  variables
    "F" -> (A => B);
```

The higher-order function `*` (map) takes as arguments a function from `A` to `B` and a list of `A`s and applies the function to each element of the list.

```
  equations
    F * ()          = ();
    F * (X)         = F(X);
    F * (Lp1 ^ Lp2) = [(F * Lp1) ^ (F * Lp2)];
```

If we want to pass functions such as length and map themselves as arguments to some higher-order function we need to define the combinators (curried versions) associated with the functions as follows:

```
  syntax
    "if"      -> (bool => A => A => A);
    "flength" -> ([{A Sep}*] => nat);
    "(*)"     -> (A => B) => ([{A Sep}*] => [{B Sep}*]);

    (A   => [R']) "(" A ")" -> R';
    ([R] => B)    "(" R ")" -> B;
    ([R] => [R']) "(" R ")" -> R';
  equations
    if(B)(X)(Y) = if B then X else Y fi;
    flength(L)  = length(L);
    (*)(F)(L)   = F * L;
```

Observe the usage of the operator `[_]` that injects `regtypes` into types in order to reuse the functionality for `type` expressions. We added extra application operators to apply functions like `flength` to lists.

These examples illustrate how two-level grammars provide user-definable syntax for sort symbols and generic definition of polymorphic mix-fix functions and constructors over data types. More advanced examples of two- and multi-level specifications (with prefix function signatures instead of grammars) can be found in Hearn and Meinke (1994), Hearn (1995) and Part III.

15.5 Properties

We have seen how two-level grammars can be used for polymorphic syntax definition in algebraic specification. To actually use two-level grammars in an executable specification formalism, it is necessary that we can parse strings over the language specified by a grammar. Unfortunately, the parsing problem for two-level grammars is in general undecidable as shown by the following theorem.

Theorem 15.5.1 (Sintzoff, 1967) *For every semi-Thue system T we can construct a Van Wijngaarden Grammar W such that the set $S(T)$ generated by T is the set $S(W)$ generated by W.*

Corollary 15.5.2 (Sintzoff, 1967) *Every recursively enumerable set is generated by a Van Wijngaarden grammar.*

Corollary 15.5.3 (Sintzoff, 1967) *The problem of determining, of a given string, whether or not it is generated by a given Van Wijngaarden grammar, is recursively unsolvable.*

Although the version of two-level grammars defined in this chapter is somewhat weaker because it uses trees instead of strings as nonterminals at level 0, these constructs can be translated to our two-level grammars. From these theorems it follows that we cannot construct terminating parsers for arbitrary two-level grammars in a general way. However, for the purpose of polymorphic syntax definition we are interested only in restricted forms of the formalism.

One view on two-level grammars is that they are used to abbreviate frequently occurring patterns in context-free grammars, but that in the end we want only a finite context-free grammar from a ground subgrammar and the appropriate instantiations of generic productions. For instance, the grammar of the programming language in module `while` gives rise to the instantiation of the list construction functions for `{stat ';'}*` and to the instantiation of the list and map functions for lists of statements. This is the effect that is reached when reuse of functions is obtained by means of parameterized modules for which only finitely many instantiations are requested. Although it is clear by looking at a grammar, which instantiations of productions are needed for the implementation of a certain subgrammar, we have not yet found a syntactic characterization of productions such that such subgrammar operations are possible. A promising approach might be the extension of the layering operations of Hatcher and Rus (1976) and Rus and Jones (1995) to two-level grammars.

In context-free grammars empty (ϵ) and chain productions are the cause of infinite ambiguities. In two-level grammars they are the cause of the undecidability of the formalism. In the rest of this section we investigate the restriction of the usage of such productions in order to achieve a subclass of the two-level grammars with a decidable parsing problem that still allows the kind of grammars as shown in §15.4.

Definition 15.5.4 (ϵ-elimination) The conventional method for eliminating ϵ-productions from context-free grammars applied to two-level grammars works by adding productions to the level 0 grammar $\mathcal{G}_0$ according to the rule

$$\frac{\alpha A \beta \to B \in \mathcal{G}_0, \epsilon \to A' \in \mathcal{G}_0, \sigma(A') = \sigma(A)}{\sigma(\alpha\beta) \to \sigma(B) \in \mathcal{G}_0}$$

where σ is a most general unifier of A and A'. After no more productions can be added, all ϵ-productions are removed. Define $ee(\Gamma)$ to be the result of removing ϵ-productions from two-level grammar Γ by the above procedure.

Note that ϵ-elimination preserves both the language and the trees generated by the grammar (if ϵ-trees are identified).

Proposition 15.5.5 $\mathrm{L}(ee(\Gamma)) = \mathrm{L}(\Gamma)$ *and* $\mathrm{Tree}(\Gamma) \cong \mathrm{Tree}(ee(\Gamma))$

Deussen (1975) shows that this method can turn finite two-level grammars into infinite ones. Consider the following grammar that gives type `<a`^{n}`>` to each sentence $\mathtt{a}^m$ with $m \leq n$.

```
level 1
  sorts type, list;
  syntax
    "<" list ">" -> type
    "a"          -> list
    list "a"     -> list
  variables
    "L" -> list
level 0
  syntax
    <L> <a> -> <L a>;
    "a"     -> <a>;
            -> <a>;
```

If we try to eliminate the last production by substituting it in the first production we get the productions

```
   <L> -> <L a>;    %% <a> can be empty
       -> <a a>;    %% <L> unifies with the rhs of -> <a>
       -> <a a a>;  %% <L> unifies with the rhs of -> <a a>
```

and all other productions of the form `->` `<a`^{n}`>` for $n > 0$. However, for many applications ϵ-productions can be eliminated. For instance, the production `-> A*` in the list grammar in §15.4 can be eliminated by means of the procedure outlined above, resulting in a finite two-level grammar defining the same language.

In a similar fashion chain productions can be eliminated from grammars.

Definition 15.5.6 (Chain Elimination) To eliminate chain production from a two-level grammar Γ, first take the transitive closure of all chains in the level 0 grammar $\mathcal{G}_0$:

$$\frac{\tau_1 \to \tau_2 \in \mathcal{G}_0, \tau_3 \to \tau_4 \in \mathcal{G}_0, \sigma(\tau_2) = \sigma(\tau_3)}{\sigma(\tau_1) \to \sigma(\tau_4) \in \mathcal{G}_0}$$

then use chain productions as substitutions

$$\frac{\tau_1 \to \tau_2 \in \mathcal{G}_0, \alpha\tau_3\beta \to \tau_4 \in \mathcal{G}_0, \sigma(\tau_2) = \sigma(\tau_3)}{\sigma(\alpha\tau_1\beta \to \tau_4) \in \mathcal{G}_0,}$$

and finally remove all chain productions from $\mathcal{G}_0$ resulting in $ce(\Gamma)$.

This procedure also preserves the language and trees generated by a grammar.

Proposition 15.5.7 $\mathrm{L}(\Gamma) = \mathrm{L}(ce(\Gamma))$ *and* $\mathrm{Tree}(\Gamma) \cong \mathrm{Tree}(ce(\Gamma))$.

Also chain elimination does not terminate for all grammars. Take for instance the grammar for lists in the previous section. If we redefine the syntax of the list operators as

```
"{" type sep "}" "*" -> type;
"{" type sep "}" "+" -> type
```

then we have that `A` unifies with `{A Sep}+` resulting in infinitely many productions

```
A                -> {A Sep}+;
{A Sep}+         -> {{A Sep}+ Sep}+;
{{A Sep}+ Sep}+  -> {{{A Sep}+ Sep}+ Sep}+;
...
```

An `A` is a singleton list of `As`, which is a singleton list of lists of `As`, etc.

So we see that ϵ-elimination and chain elimination will not terminate for arbitrary grammars. However, for the grammars for which it succeeds we have the following corollary from Theorem 15.6.8 that we will prove in the next section.

Corollary 15.5.8 *If* Γ *is a finite two-level grammar without* ϵ*- and chain productions, then the question* $w \in \mathrm{L}(\Gamma)$ *is decidable.*

The intuition behind this result is that with ϵ- and chain-free grammars at most n reductions can be done for a string of length n. Based on the same idea, the next definition defines a characterization of a larger class of two-level grammars for which the parsing problem is decidable.

Definition 15.5.9 (Finite Chain Property) A two-level grammar Γ has the *finite chain property* if it is (1) ϵ-free, (2) its chain productions are non-cyclic and have a finite transitive closure and (3) it does not contain redundant productions, where a production $p \in \mathrm{P}(\mathcal{G}_0)$ is redundant if there is some $p' \in \mathrm{P}(\mathcal{G}_0)$ such that $p \neq p'$ and $p \geq p'$.

In the next section we will define a parsing algorithm for two-level grammars and prove that it is a decision procedure for membership of languages defined by finite chain two-level grammars.

On the positive side we have a subclass of the two-level grammars with a decidable parsing problem. On the negative side, membership of the class itself is undecidable.

Proposition 15.5.10 *It is undecidable whether a two-level grammar satisfies the finite chain property.*

However, decidability of the finite chain property is not essential for using two-level grammars for language specification. The situation can be compared to ambiguity of context-free grammars. Although it is undecidable whether a context-free grammar is ambiguous, it is a good formalism for defining unambiguous languages. A large class of grammars is evidently non-ambiguous and for others ambiguities will turn up when working with the grammars.

The examples presented in §15.4 satisfy the finite chain property, except for the empty production `-> {A Sep}*` for lists. As remarked above this production is not a problematic ϵ-production because it can be eliminated from the grammar. In general we can follow the following procedure for determining whether a grammar has the finite chain property: (1) Try to eliminate ϵ-productions by the method of Definition 15.5.4. (2) Try to eliminate chain rules by means of the method in Definition 15.5.6. (3) If this terminates we know that the grammar has the finite chain property and that we can parse with it (see next section). (4) If either step (1) or step (2) takes too long, this is a hint that it does not terminate. In such cases we can inspect the list of ϵ-productions or chain productions added by the elimination procedures. These traces will give a clue about the productions that cause the nontermination, because these will lead to a repetition of similar productions, as we saw in the example above. This information can be used to redesign the grammar such that it satisfies the finite chain property.

15.6 Parsing

In this section we define a parsing algorithm for finite chain two-level grammars. The parsing algorithm below is a parallel bottom-up parsing algorithm that computes all parse trees for a sentence. This procedure is similar to the Hindley-Milner type assignment procedure used in functional languages, that assigns to each expression a single principal type (Damas and Milner, 1982). The difference is that in two-level grammars strings can have more than one principal type due to ambiguities. It will turn out that for finite chain two-level grammars there are only finitely many principal types for a string. We first define a function that gives the type of a parse tree.

Definition 15.6.1 The type of a parse tree is defined as:

$$\begin{aligned} \text{type}(L) &= L \\ \text{type}(\text{var}(x, \tau)) &= \tau \\ \text{type}(\text{app}(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n])) &= \tau \end{aligned}$$

Next we define the data structure of parse configurations that is used in parsing.

Definition 15.6.2 A parse configuration $(\vec{t} \bullet a_1 \ldots a_n)_\Phi$ is an element of the set $\text{Tree}(\Gamma \cup \text{V})^* \times \text{Sl}(\Gamma)^* \times \mathit{Set}(\text{V}_1)$, i.e., a triple consisting of a list of trees $\vec{t} = t_1 \ldots t_m$ (the stack), a list of literals $a_1 \ldots a_n$ (the remaining input) and a set of sort variables Φ (the sort variables over level 1 that are used in $\vec{t}$).

Algorithm 15.6.3 Define the function $\text{parse}(\Gamma) : \text{S}(\Gamma)^* \to \mathit{Set}(\text{Tree}(\Gamma))$ as

$$\text{parse}(\Gamma)(w) = \{t \mid (\epsilon \bullet w) \Rightarrow_\Gamma^* (t \bullet \epsilon)\}$$

where $\Rightarrow_\Gamma^*$ is the transitive closure of the one-step parse relation $\Rightarrow_\Gamma$ on parse configurations, which is defined by the rules

$$(\vec{t} \bullet a_1\ a_2 \ldots a_n)_\Phi \Rightarrow_\Gamma (\vec{t}\, a_1 \bullet a_2 \ldots a_n)_\Phi \tag{Shift}$$

$$\frac{x \in \text{V}(\tau'),\ \tau = \bar{\rho}(\tau')}{(\vec{t}\, x \bullet \vec{a})_\Phi \Rightarrow_\Gamma (\vec{t}\, \text{var}(x, \tau) \bullet \vec{a})_{\Phi \cup \text{vars}(\tau)}} \tag{Var}$$

$$\frac{p \in \text{P}(\mathcal{G}_0),\ \bar{\rho}(p) = \alpha \to \tau,\ |\alpha| = m,\ \text{mgu}(\alpha; \text{type}(t_1, \ldots, t_m)) = \sigma}{(\vec{t}\, t_1 \ldots t_m \bullet \vec{a})_\Phi \Rightarrow_\Gamma (\vec{t}\, \bar{\bar{\sigma}}(\text{app}(p, [t_1, \ldots, t_m])) \bullet \vec{a})_{\Phi \cup \text{vars}(\bar{\sigma}(p))}} \tag{Red}$$

where $\rho : \text{V}_1 \to \text{V}_1$ is a renaming of sort variables occurring in Φ such that $\rho(\Phi) \cap \Phi = \emptyset$. We identify configurations that are the same up to renaming of sort variables.

We now prove that the algorithm is a correct implementation of $\Pi(\Gamma)$ for finite chain two-level grammars. We first show that the trees produced by the parser are correct parse trees.

Lemma 15.6.4 (Sound) $\forall t \in \text{Tree}(\Gamma) : t \in \text{parse}(\Gamma)(w) \Rightarrow \text{yield}(t) = w$

Proof. We first prove that if $(\vec{t_1} \bullet \vec{a_1}) \Rightarrow_\Gamma (\vec{t_2} \bullet \vec{a_2})$, then $\text{yield}(\vec{t_1})\vec{a_1} = \text{yield}(\vec{t_2})\vec{a_2}$. For (Shift) and (Var) the property clearly holds. In (Red) we see $\text{yield}(\vec{t}\, t_1 \ldots t_m)\vec{a} = \text{yield}(\vec{t}\, \bar{\bar{\sigma}}(\text{app}(p, [t_1 \ldots t_m])))\vec{a}$ by definition of yield and by the fact that type substitutions do not affect the yield of a tree. But then also for $(\vec{t_1} \bullet \vec{a_1}) \Rightarrow_\Gamma^* (\vec{t_2} \bullet \vec{a_2})$ we have $\text{yield}(\vec{t_1})\vec{a_1} = \text{yield}(\vec{t_2})\vec{a_2}$. In particular, if $(\epsilon \bullet w) \Rightarrow_\Gamma^* (t \bullet \epsilon)$ we have that $w = \text{yield}(\epsilon)w = \text{yield}(t)\epsilon$. □

Next we show that the parser is *complete*, in the sense that any parse tree for the sentence can be derived by instantiating one of the parse trees produced by the parser.

Lemma 15.6.5 (Complete) $\forall t \in \mathrm{Tree}(\Gamma) : \mathrm{yield}(t) = w \Rightarrow \exists t' \geq t : t' \in \mathrm{parse}(\Gamma)(w)$

Proof. By induction on t: (1) if $t = L$ then $\mathrm{parse}(\Gamma)(L) \ni L \geq L$.
(2) if $t = \mathrm{var}(x, \tau)$, there is some $\tau' \geq \tau$ such that $x \in \mathrm{V}(\tau')$, but then $\mathrm{parse}(\Gamma)(x) \ni \mathrm{var}(x, \tau') \geq \mathrm{var}(x, \tau)$.
(3) If $t = \mathrm{app}(p, [t_1, \ldots, t_n])$ with $p = \tau_1 \ldots \tau_n \to \tau$ (a) By definition of yield we have $\mathrm{yield}(t) = \mathrm{yield}(t_1) \ldots \mathrm{yield}(t_n) = w_1 \ldots w_n$ with $\mathrm{yield}(t_i) = w_i$ for $1 \leq i \leq n$. (b) By induction we have $t'_i \geq t_i$ for $1 \leq i \leq n$—and thus $\tau'_i \geq \tau_i$ with $\tau'_i = \mathrm{type}(t'_i)$—such that $t'_i \in \mathrm{parse}(\Gamma)(w_i)$. (c) By (App) there is some $\tau''_1 \ldots \tau''_n \to \tau'' = p'' \in \mathrm{P}(\mathcal{G}_0)$ such that $p'' \geq p$ (variables of p'' and p disjunct). By (b) and (c) there is a substitution σ_0 such that $\bar{\sigma}_0(\tau''_1 \ldots \tau''_n) = \tau_1 \ldots \tau_n = \bar{\sigma}_0(\tau'_1 \ldots \tau'_n)$. Then there is also a most general unifier, say σ. Now take $t' = \bar{\bar{\sigma}}(\mathrm{app}(p'', [t'_1 \ldots t'_n])) \in \mathrm{Tree}(\Gamma)$. It is clear that $t' \geq t$ and that $(\epsilon \bullet w_1 \ldots w_n) \Rightarrow^*_\Gamma (t'_1 \ldots t'_n \bullet \epsilon) \Rightarrow_\Gamma (t' \bullet \epsilon)$ □

Next we show that the set of parse trees produced by the algorithm is *minimal* in the sense that it generates only the most general parse trees for a string.

Lemma 15.6.6 (Minimal) $\forall t, t' \in \mathrm{parse}(\Gamma)(w) : t \geq t' \vee t' \geq t \Rightarrow t \doteq t'$

Proof. Assume that $t, t' \in \mathrm{parse}(\Gamma)(w)$ and that $t \gtrdot t'$. Because both trees are in the set there must be sequences of configurations for their derivation. Because $t \gtrdot t'$, the trees have the same structure, i.e., the configuration sequences have the same number of reductions and shifts. But also because $t \gtrdot t'$, there must be some point at which the sequences diverge, i.e.,

$$(\epsilon \bullet w) \Rightarrow^*_\Gamma (\vec{t_1}\vec{t_2} \bullet \vec{a}) \Rightarrow_\Gamma (\vec{t_1}\ \bar{\bar{\sigma}}_1(\mathrm{app}(p_1, \vec{t_2})) \bullet \vec{a}) \Rightarrow^*_\Gamma (t \bullet \epsilon)$$

and

$$(\epsilon \bullet w) \Rightarrow^*_\Gamma (\vec{t_1}\vec{t_2} \bullet \vec{a}) \Rightarrow_\Gamma (\vec{t_1}\ \bar{\bar{\sigma}}_2(\mathrm{app}(p_2, \vec{t_2})) \bullet \vec{a}) \Rightarrow^*_\Gamma (t' \bullet \epsilon)$$

for t and t', respectively. Because $t \gtrdot t'$ we must have $\bar{\sigma}_1(p_1) \gtrdot \bar{\sigma}_2(p_2)$. Now we have either (1) $p_1 = p_2$ and $\sigma_1 \gtrdot \sigma_2$, but then σ_2 is not a most general unifier and hence $t' \notin \mathrm{parse}(\Gamma)(w)$ or (2) if $p_1 \gtrdot p_2$, then Γ does not satisfy the finite chain property because it has the redundant production p_2. □

Finally we prove that parse yields a finite set of parse trees, entailing that parse is effectively computable.

Lemma 15.6.7 (Finite) $|\mathrm{parse}(\Gamma)(w)| \in N$

Proof. (1) For each configuration and each production there is at most one reduction step (Red) because there is at most one most general unifier for α and $\tau_1 \ldots \tau_m$. For each configuration there is at most one (Shift) step and one (Var) step. Therefore, the graph of the relation $\Rightarrow_\Gamma$ is finitely branching.
(2) The length of configurations does not increase (no ϵ-productions). For any configuration $(\vec{t} \bullet \vec{a})$, $|\vec{a}|$ (Shift) steps can be done. A (Red) step with a production $\alpha \to \tau$ such that $|\alpha| > 1$ decreases the length of a configuration, therefore at

most $|\vec{t}|/2$ such reductions can be performed for a configuration $(\vec{t} \bullet \vec{a})$. By the finite chain property only finitely many chain reductions can be done, i.e., for each configuration $(\vec{t}\, t \bullet \vec{a})$ there is a maximal value n such that $(\vec{t}\, t \bullet \vec{a}) \Rightarrow^n_\Gamma (\vec{t}\, t' \bullet \vec{a})$. Therefore, the graph of the relation $\Rightarrow_\Gamma$ has no infinite paths.
(3) From any configuration $(\vec{t} \bullet \vec{a})$ only finitely many configurations are reachable. In particular, for any string w only finitely many configurations of the form $(t \bullet \epsilon)$ are reachable from $(\epsilon \bullet w)$. □

Finally, we see that Algorithm 15.6.3 is a correct implementation of a parser for finite chain two-level grammars.

Theorem 15.6.8 (Correct) *For any finite chain two-level grammar* Γ *and any string* $w \in \mathrm{Sl}(\Gamma)^*$, $\mathrm{parse}(\Gamma)(w)$ *is a minimal and finite set of parse trees, unique up to renaming of sort variables, that generates* $\Pi(\Gamma)(w)$.

Proof. By Lemma 15.6.4 and Lemma 15.6.5 all and exactly the trees in $\Pi(\Gamma)(w)$ can be derived from $\mathrm{parse}(\Gamma)(w)$. By Lemma 15.6.7 $\mathrm{parse}(\Gamma)(w)$ is finite and by Lemma 15.6.6 it is minimal. □

As a result the recognition problem for finite chain two-level grammars is decidable.

Corollary 15.6.9 (Decidable) *For a finite chain two-level grammar* Γ *it is decidable whether* $w \in \mathrm{L}(\Gamma)$ *and* $w \in \mathrm{L}(\Gamma)(\tau)$.

The relation $\Rightarrow_\Gamma$ defines a tree shaped search space. Only the types of trees in the configuration matter for the rest of the process. We would like to identify configurations $(\vec{t_1} \bullet \vec{a})$ and $(\vec{t_2} \bullet \vec{a})$ for which $|\vec{t_1}| = |\vec{t_2}|$ and $\mathrm{type}(\vec{t_1}) = \mathrm{type}(\vec{t_2})$. This would lead to a generalization of the graph structured stack and the parse forests of Tomita (1985) to parsing for two-level grammars.

15.7 Related Formalisms

In the same way that context-free grammars correspond to first-order signatures, two-level grammars correspond to two-level signatures. The type system of the functional programming language ML (Milner, 1978) can be considered as two-level signatures in which the expressions over level 1 are single-sorted expressions of sort `type`. This system was used to introduce parametric polymorphic functions. Two-level signatures are discussed in Poigné (1986), Möller (1987), Meinke (1992a). After a two-level signature is expanded, a, possibly infinite, one-level signature results that can again be used as the specification of the sort space of a level 0 signature. In this manner the extension of signatures to two-level signatures can be generalized to signatures with three and more levels. Hearn and Meinke (1994) introduce the three-level algebraic specification formalism Atlas, which is generalized by Hearn (1995) to a multi-level specification formalism. The complete and formal specification of the related multi-level specification formalism MLS is presented in Part III. MLS supports overloading of function symbols, which entails that a term can have infinitely many types, but only finitely many most general or principal types. This property is not respected by general two-level grammars as discussed in §15.5.

On the grammatical side, many variants of two-level grammars have been proposed in the literature for various purposes. Van Wijngaarden grammars (VWG) (van Wijngaarden *et al.*, 1976) were developed to express the syntax and semantics of Algol68. In VWGs strings, instead of trees, over level 1 are used as nonterminals (hypernotions) at level 0. This leads to the problem of grammatical unification—whether two sentential forms over a context-free grammar are unifiable by means of a substitution of nonterminals with strings—which Maluszynski (1984) shows to be undecidable. The transparent two-level grammars of Maluszynski (1984) are a restriction of VWGs such that grammatical unification comes down to term unification. Another restriction of VWGs are the Extended Affix Grammars (EAG) (Watt, 1977) that restrict the order in which the variables in nonterminals at level 0 can be instantiated.

The observation that two-level grammars are Turing equivalent sparked another development: two-level grammars as logic or functional programming languages. The (context-free) metagrammar (level 1) is used to define the syntax of language and semantic domains. The hypergrammar (level 0) is used to define the operations on the data. See for example Maluszynski (1984).

Definite Clause Grammars (DCG) introduced by Pereira and Warren (1980) are grammars embedded in Prolog programs. They are equivalent to two-level grammars with a fixed level 1 equivalent to the following grammar

```
module dcg
level 1
  sorts fun, term;
  syntax
    [a-z][A-Za-z0-9]*       -> fun;
    fun                     -> term;
    fun "(" {term ","}* ")" -> term;
  variables
    [A-Z][A-Za-z0-9]* -> term;
```

that defines an untyped domain of terms that can be used as grammar symbols in level 0. These terms are then typically used at level 0 in productions such as the following from a tiny natural language grammar:

```
np(N) vp(N)        -> s;
det(N) n(N) rel(N) -> np(N);
```

Parsing of DCGs—parsing as deduction (Pereira and Warren, 1983)—uses Prolog's built-in resolution strategy to answer queries like $w \in \mathrm{L}(\mathcal{G})(\mathtt{s})$. With the normal evaluation strategy of Prolog (SLD resolution) this comes down to top-down backtrack parsing. Problems with this strategy are that it cannot cope with left-recursion and that already computed answers are not reused. The tabulation strategy described in Warren (1992) partially overcomes these problems. One of the problems of the latter approach is that unification in Prolog is not many-sorted, disabling solutions like that with `regtype` in §15.4.

15.8 Conclusions

Algebraic specification with first-order signatures or context-free grammars enforce a choice between strongly typed structures with little genericity or generic operations over untyped structures. Polymorphism combines genericity with typedness, making it possible to develop libraries of specifications. In this chapter we have discussed how the integration of algebraic specification with user-definable syntax and polymorphism can be materialized. The extension with polymorphism of algebraic specification formalisms that use context-free grammars as signatures, e.g., OBJ or ASF+SDF, leads necessarily to formalisms with two-level grammars as signatures. Likewise, the extension with user-definable syntax of formalisms that have polymorphic signatures, including polymorphic functional and logic programming languages, leads to two-level grammars.

In two-level grammars level 1 defines the syntax of sort symbols used at level 0. Sort terms with variables are interpreted as sort schemata that can have many instantiations. Productions at level 0 with such sorts are production schemata, i.e., declarations of polymorphic functions with mix-fix syntax. Thus two-level grammars combine polymorphism with user-definable syntax, as we illustrated by means of a number of examples of polymorphic syntax definition in data type and programming language specifications.

Although the parsing problem for context-free grammars and the type-assignment problem for two-level signatures are decidable, the parsing problem for the integration of both formalisms is undecidable if no restrictions are considered. We defined an intuitive restriction of the class of two-level grammars that results in a class of two-level grammars for which the parsing problem is decidable and for which we defined a parsing algorithm that yields a minimal and finite set of most general parse trees for each string.

Part V

Epilogue

16
Concluding Remarks

In this thesis we have investigated techniques for improving the expressivity of syntax definitions for language prototyping. The main results are the design and implementation of the syntax definition formalism SDF2, the design and specification of the multi-level algebraic specification formalism MLS and theory about polymorphic syntax definition. There are many opportunities for further research. Here we mention a few.

16.1 Syntax

Implementation of SDF2 There are several possibilities for improvement of the implementation of SDF2. The parser generator that is currently implemented in ASF+SDF is not efficient enough and probably needs to be reimplemented in an imperative language that offers direct access to data structures. A more generic matter is the compact representation of parse tables. Due to the use of productions in the goto relation more transitions are computed than in a normal LR table. This is necessary to deal with priorities. However, for sorts that have no priorities and productions that have the same priority, the transitions should be shared. If the sharing is computed before computing the next state the performance of the parser generator might also be improved. Furthermore, the disambiguation with priorities as multi-set filters needs to be implemented.

Several improvements of the disambiguation are possible. Priority disambiguation could be refined such that the priority relation applies only to a selected number of arguments. Lexical disambiguation rules by means of follow restrictions and reject productions could be generated automatically from the grammar.

Applications of SDF2 The implementation of SDF2 has been tested to generate parsers for ASF+SDF specifications by upgrading the syntax part of a specification to SDF2. These parsers are used as front-ends for compiled ASF+SDF specifications. The next step is to couple semantics specifications directly to SDF2 definitions. In that manner the full expressiveness of SDF2 can be used. In principle, SDF2 is suitable for connection with other logics than equational logic as in ASF+SDF. Experiments could be made with frameworks such as functional programming, logic programming, and theorem provers to provide these languages with full user-definable syntax.

The implementation of SDF2 as described here is batch-oriented. A complete syntax definition is normalized and fed to the parser generator. To incorporate the SDF2 implementation in the new ASF+SDF Meta-Environment (Van den Brand *et al.*, 1997c), it might be necessary to reconsider the modular, lazy and incremental parser generation schema of Rekers (1992) to combine it with grammar normalization and renaming.

Filters In Chapter 4 a theoretical framework of disambiguation filters was presented. Given the infrastructure provided by the SDF2 implementation, experiments with disambiguation filters could be performed. Also the definition of lexical disambiguation by means of filters should be explored. Given such experimental filters, the investigation of optimized combinations of parsers and filters could be further pursued and generalized.

Parsers for Polymorphic Syntax Definitions In Chapter 15 an interpretative parsing algorithm for two-level grammars is defined. A more efficient implementation of parsing for two-level grammars could be achieved if an LR-like preprocessing of the grammar into a parse table could be derived. This should involve unification in the table lookup and storage of binding of types to type variables in item-sets.

Parser Generator Generators In Chapter 5 we derived an adaptation of the LR(0) parser generation algorithm by adapting its underlying parsing schema. The implementation of the derived schema was achieved by adapting an implementation of the parser generator. It would be desirable to generate this implementation from the schema automatically. This is actually not such a farfetched idea. Shieber *et al.* (1995) describe the interpretation of parsing schemata by means of logic programs that implement the control of a parser by maintaining a chart of all items derivable from a sentence. This chart interpretation is parameterized with a set of deduction rules for parsing, i.e., a parsing schema.

Now consider a chart parser with the schema for Earley's algorithm. In chart parsing the items are computed individually and dynamically. In Earley parsing items are bundled in sets but these are still computed dynamically. In LR parsing items are also bundled in sets, but the transitions between such item-sets is precomputed, i.e., computed only once for a grammar, inducing a considerable efficiency gain. All these algorithms are driven by the same parsing schema, but have a different control structure. There is a refinement relation between them as well. Earley is derived from chart parsing, by taking sets of items closed under prediction. Transitions occur from a set to a new set by shifting with some symbol. The initial item-set is the closure of the axiom rules of the schema. In LR these sets are precomputed in the same way. This procedure might be generalizable to arbitrary parsing schemata. Single premise rules are closure rules, rules with as one of the premises a token are shift rules, etc. In this way we can derive new parser generators by adapting a parsing schema and letting the parser generator generator derive an implementation from it.

Extensible Syntax An open problem is the description of the syntax of extensible languages. An extensible language is a language that can extend its own syntax. ASF+SF itself is an extensible language. The syntax of equations is parameterized with a grammar describing the syntax of terms. This is handled in the current Meta-Environment by separating the definition of syntax and equations such that the syntax definition can be parsed and analysed before the equations are parsed. It would be desirable to be able to mix syntax and equations. This would require the adaptation of the parser during parsing. The syntax of two-level grammars is even more extensible. The syntax of the rules in level 0 depends on the productions of level 1. Other work in this area includes Cardelli *et al.* (1994) who give a definition of a language with extensible syntax.

16.2 Type Systems

Module Systems The specification of SDF2 and MLS both contain a specification of module systems. It would be desirable to generalize these specifications to make the definition of the module system orthogonal to the details of the rest of the formalism.

Type equations The typechecker specified in Chapter 13 considers types as elements of a free term algebra. This means that normal syntactic unification can be used in type assignment. This is not sufficient if equations over type expressions are allowed. Such equations are useful for describing more powerful type systems, including general product types and type classes without further extending the framework. Other examples of the application of type equations in multi-level algebraic specifications to the modelling of type systems include the specification of type abbreviations, recursive types, record types, the polytypic functions of Jansson and Jeuring (1997), the type classes of Haskell and the constructor classes of Jones (1993).

Type assignment is undecidable in general for multi-level specifications with type equations, because it requires E-unification. In Visser (1996b) a preliminary study of methods for type assignment that are useful for a subset of multi-level specifications with type equations is presented.

Interaction between Levels In MLS and two-level grammars we have the problem of transferring values to the next level. Examples are separators in lists and field names in records. Currently this has to be done by introducing a sort at the type level containing all these values and declaring for each element that it generates the same element at the object level below it. A more elegant way should be possible.

16.3 Program and Specification Schemata

In this thesis we have applied several methods for reusing syntax definitions or specifications. The normalization function of SDF2 generates productions from the symbols used in a syntax definition, providing abbreviations for frequently occuring patterns. The module mechanism of SDF2 provides reuse of user-definable definitions. Renaming makes modules even more reuseable by freeing the user from the particular names chosen in the module that is reused. User-definable polymorphic types in MLS allow generic definition of data types and higher-order functions. Overloading and ambiguous grammars provide another method to keep syntax definitions concise.

A method not covered by all these forms of reuse is one directed at reuse by exploiting the genericity in the structure of data. An example application of such genericity is the formatter generator described in Van den Brand and Visser (1996). Given a context-free grammar, a specification of a formatter is generated. An extension of this technique to arbitrary transformation rules on terms is presented in Van den Brand *et al.* (1997b). In this approach the abstract derivation rules used to generate specifications and their implementation in a generator are intertwined. In Luttik and Visser (1997) the term traversal operator for rewriting strategies is defined using a schema that should be instantiated for all functions in the signature. It is desirable to drive the generation of generic functionality by means of such schemata.

One approach in this direction is the polytypic programming of Jansson and Jeuring (1997), where each regular data structure is associated with a data structure built from a few primitives such as product and recursion. Very general recursion functionals can be defined for those primitives structures and thus be implemented for all regular data types with very little effort, i.e., the conversion between the data type and its representation in terms of the primitives.

A similar result could also be achieved in a first-order setting as is illustrated by the preprocessor of Elan (Vittek, 1994) that is used to generate specifications given some extended signature. In fact the preprocessor statements correspond to specifications with a second-order quantor ranging over signature elements.

It would be interesting to consider a full integration of such second-order quantification in—the syntax definition formalism of—an algebraic specification formalism, making second-order quantification over sorts and functions available in a general approach to generation of derived syntax and equations. This could be used for example to make the regular expressions of SDF2 user-definable instead of built-in constructs or to generate traversal functions. Such specification schemata would provide a kind of reuse not currently available by means of the conventional abstraction facilities.

Part VI

Appendices

A

Auxiliary Modules for the Specification of SDF2

In this appendix we include several auxiliary modules used in the specification of SDF2.

A.1 Literals

module Literals
imports Layout
exports
  sorts Literal
  lexical syntax
    "\\"~□ → EscChar
    "\\"[01][0-7][0-7] → EscChar
    ~[\000-\040"\\] → L-Char
    [␣\t\n] → L-Char
    EscChar → L-Char
    "\\""L-Char∗"\\"" → Literal
  variables
    "*L*"[*0-9′*]∗ → Literal

A.2 ATerms

A.2.1 Constructors

module Grammar-Tree-Constructors
imports ATerms[7.5.1]
exports
  context-free syntax
    "empty-grammar" → AFun
    "conc-grammars" → AFun

module CC-Sdf-Tree-Constructors

imports Kernel-Sdf-Tree-Constructors[7.5.2]
exports
  context-free syntax
    "char-class" → AFun
    "range" → AFun

module Sorts-Sdf-Tree-Constructors
imports Kernel-Sdf-Tree-Constructors[7.5.2]
exports
  context-free syntax
    "sort" → AFun

module Literals-Sdf-Tree-Constructors
imports Kernel-Sdf-Tree-Constructors[7.5.2]
exports
  context-free syntax
    "lit" → AFun

module Regular-Sdf-Tree-Constructors
imports ATerms[7.5.1]
exports
  context-free syntax
    "empty" → AFun
    "seq" → AFun
    "opt" → AFun
    "iter" → AFun
    "iter-star" → AFun
    "iter-sep" → AFun
    "iter-star-sep" → AFun
    "iter-n" → AFun
    "iter-sep-n" → AFun
    "set" → AFun
    "pair" → AFun
    "func" → AFun
    "alt" → AFun
    "perm" → AFun

module Basic-Sdf-Tree-Constructors
imports Kernel-Sdf-Tree-Constructors[7.5.2]
exports
  context-free syntax
    "lexical-syntax" → AFun
    "context-free-syntax" → AFun
    "variables" → AFun
    "lexical-variables" → AFun
    "cf" → AFun
    "lex" → AFun
    "varsym" → AFun

"layout" $\rightarrow$ AFun

A.2.2 Encoding and Decoding

module Grammar-ATerms
imports Grammar-Tree-Constructors$^{A.2.1}$ Grammar-Syntax$^{7.2}$
exports
context-free syntax
aterm(Grammar) $\rightarrow$ ATerm
grammar(ATerm) $\rightarrow$ Grammar
equations
Encoding of grammars.

[1] $\text{aterm}(\emptyset) = \text{empty-grammar}$
[2] $\text{aterm}(\mathcal{G}_1\ \mathcal{G}_2) = \text{conc-grammars}(\text{aterm}(\mathcal{G}_1), \text{aterm}(\mathcal{G}_2))$

Decoding of grammars.

[3] $\text{grammar}(\text{empty-grammar}) = \emptyset$
[4] $\text{grammar}(\text{conc-grammars}(T_1,\ T_2)) = \text{grammar}(T_1)\ \text{grammar}(T_2)$

module CC-Sdf-ATerms
imports Kernel-Sdf-ATerms$^{7.5.4}$ CC-Sdf-Syntax$^{7.4.2}$
CC-Sdf-Tree-Constructors$^{A.2.1}$ Character-Arithmetic
exports
context-free syntax
atermlist(OptCharRanges) $\rightarrow$ ATermList
aterm(Character) $\rightarrow$ NatCon
ranges(ATermList) $\rightarrow$ CharRanges
range(ATerm) $\rightarrow$ CharRange
character(ATerm) $\rightarrow$ Character
equations
Encoding character classes.

[1] $\text{aterm}([cr^*]) = \text{char-class}(\text{atermlist}(cr^*))$
[2] $\text{atermlist}(cr^*) = [\ ]$ **when** $cr^* =$
[3] $\text{atermlist}(cr_1^+\ cr_2^+) = \text{atermlist}(cr_1^+) +\!\!+\ \text{atermlist}(cr_2^+)$
[4] $\text{atermlist}(cr) = [\text{aterm}(\mathsf{c})]$ **when** $cr = \mathsf{c}$
[5] $\text{atermlist}(\mathsf{c}_1 - \mathsf{c}_2) = [\text{range}(\text{aterm}(\mathsf{c}_1), \text{aterm}(\mathsf{c}_2))]$
[6] $\text{aterm}(\mathsf{c}) = \text{int}(\mathsf{c})$

Decoding character classes.

[7] $\text{symbol}(\text{char-class}([\ ])) = []$
[8] $\text{symbol}(\text{char-class}([Ts])) = [\text{ranges}([Ts])]$
[9] $\text{ranges}([T]) = \text{range}(T)$

[10] $\text{ranges}([T,\ Ts]) = \text{range}(T)\ \text{ranges}([Ts])$
[11] $\text{range}(n) = \text{character}(n)$
[12] $\text{range}(\text{range}(n_1,\ n_2)) = \text{character}(n_1) - \text{character}(n_2)$
[13] $\text{character}(n) = \text{char}(n)$
[14] $\text{symbol}(n) = [\text{char}(n)]$

module Sorts-Sdf-ATerms
imports Kernel-Sdf-ATerms[7.5.4] Sorts-Sdf-Tree-Constructors[A.2.1]
Sorts-Sdf-Syntax[7.4.1]
equations
Encoding and decoding sorts.

[1] $\text{aterm}(\text{sort}(c^+)) = \text{sort}(\text{literal}(\texttt{"""}\ c^+\ \texttt{"""}))$
[2] $\text{symbol}(\text{sort}(\text{literal}(\texttt{"""}\ c^+\ \texttt{"""}))) = \text{sort}(c^+)$

module Literals-Sdf-ATerms
imports Kernel-Sdf-ATerms[7.5.4] Literals-Sdf-Tree-Constructors[A.2.1]
Literals-Sdf-Syntax[7.4.3]
equations
Encoding and decoding literals.

[1] $\text{aterm}(L) = \text{lit}(L)$
[2] $\text{symbol}(\text{lit}(L)) = L$

module Priority-Sdf-ATerms
imports Kernel-Sdf-ATerms[7.5.4] Priority-Sdf-Syntax[8.1.1]
equations
Encoding attributes.

[1] $\text{aterm}(\text{left}) = \text{atr}(\texttt{"left"})$
[2] $\text{aterm}(\text{right}) = \text{atr}(\texttt{"right"})$
[3] $\text{aterm}(\text{bracket}) = \text{atr}(\texttt{"bracket"})$
[4] $\text{aterm}(\text{assoc}) = \text{atr}(\texttt{"assoc"})$
[5] $\text{aterm}(\text{non-assoc}) = \text{atr}(\texttt{"non-assoc"})$

Decoding attributes.

[6] $\text{attribute}(\text{atr}(\texttt{"left"})) = \text{left}$
[7] $\text{attribute}(\text{atr}(\texttt{"right"})) = \text{right}$
[8] $\text{attribute}(\text{atr}(\texttt{"bracket"})) = \text{bracket}$
[9] $\text{attribute}(\text{atr}(\texttt{"assoc"})) = \text{assoc}$
[10] $\text{attribute}(\text{atr}(\texttt{"non-assoc"})) = \text{non-assoc}$

module Regular-Sdf-ATerms
imports Regular-Sdf-Tree-Constructors$^{A.2.1}$ Kernel-Sdf-ATerms$^{7.5.4}$
Regular-Sdf-Syntax$^{8.2.1}$
equations
Encoding regular expressions.

[1] $\text{aterm}((\ )) = \text{empty}$
[2] $\text{aterm}((\mathcal{A}\ \alpha^+)) = \text{seq}(\text{atermlist}(\mathcal{A}\ \alpha^+))$
[3] $\text{aterm}(\mathcal{A}?) = \text{opt}(\text{aterm}(\mathcal{A}))$
[4] $\text{aterm}(\mathcal{A}+) = \text{iter}(\text{aterm}(\mathcal{A}))$
[5] $\text{aterm}(\mathcal{A}*) = \text{iter-star}(\text{aterm}(\mathcal{A}))$
[6] $\text{aterm}(\{\mathcal{A}\ \mathcal{B}\}+) = \text{iter-sep}(\text{aterm}(\mathcal{A}),\ \text{aterm}(\mathcal{B}))$
[7] $\text{aterm}(\{\mathcal{A}\ \mathcal{B}\}*) = \text{iter-star-sep}(\text{aterm}(\mathcal{A}),\ \text{aterm}(\mathcal{B}))$
[8] $\text{aterm}(\{\mathcal{A}\}\ n+) = \text{iter-n}(\text{aterm}(\mathcal{A}),\ \text{con}(n))$
[9] $\text{aterm}(\{\mathcal{A}\ \mathcal{B}\}\ n+) = \text{iter-sep-n}(\text{aterm}(\mathcal{A}),\ \text{aterm}(\mathcal{B}),\ \text{con}(n))$
[10] $\text{aterm}(\text{Set}[\mathcal{A}]) = \text{set}(\text{aterm}(\mathcal{A}))$
[11] $\text{aterm}(\mathcal{A}\ \#\ \mathcal{B}) = \text{pair}(\text{aterm}(\mathcal{A}),\ \text{aterm}(\mathcal{B}))$
[12] $\text{aterm}((\alpha \Rightarrow \mathcal{B})) = \text{func}(\text{atermlist}(\alpha),\ \text{aterm}(\mathcal{B}))$
[13] $\text{aterm}(\mathcal{A} \mid \mathcal{B}) = \text{alt}(\text{aterm}(\mathcal{A}),\ \text{aterm}(\mathcal{B}))$
[14] $\text{aterm}(\ll \alpha \gg) = \text{perm}(\text{atermlist}(\alpha))$

Decoding regular expressions.

[15] $\text{symbol}(\text{empty}) = (\ )$
[16] $\text{symbol}(\text{seq}(Tl)) = (\mathcal{A}\ \alpha^+)$ **when** $\text{symbols}(Tl) = \mathcal{A}\ \alpha^+$
[17] $\text{symbol}(\text{opt}(T_1)) = \text{symbol}(T_1)?$
[18] $\text{symbol}(\text{iter}(T_1)) = \text{symbol}(T_1)+$
[19] $\text{symbol}(\text{iter-star}(T_1)) = \text{symbol}(T_1)*$
[20] $\text{symbol}(\text{iter-sep}(T_1,\ T_2)) = \{\text{symbol}(T_1)\ \text{symbol}(T_2)\}+$
[21] $\text{symbol}(\text{iter-star-sep}(T_1,\ T_2)) = \{\text{symbol}(T_1)\ \text{symbol}(T_2)\}*$
[22] $\text{symbol}(\text{iter-n}(T_1,\ n)) = \{\text{symbol}(T_1)\}\ n+$
[23] $\text{symbol}(\text{iter-sep-n}(T_1,\ T_2,\ n)) = \{\text{symbol}(T_1)\ \text{symbol}(T_2)\}\ n+$
[24] $\text{symbol}(\text{set}(T_1)) = \text{Set}[\text{symbol}(T_1)]$
[25] $\text{symbol}(\text{pair}(T_1,\ T_2)) = \text{symbol}(T_1)\ \#\ \text{symbol}(T_2)$
[26] $\text{symbol}(\text{func}(Tl_1,\ T_2)) = (\text{symbols}(Tl_1) \Rightarrow \text{symbol}(T_2))$
[27] $\text{symbol}(\text{alt}(T_1,\ T_2)) = \text{symbol}(T_1) \mid \text{symbol}(T_2)$
[28] $\text{symbol}(\text{perm}(Tl)) = \ll \text{symbols}(Tl) \gg$

module Basic-Sdf-ATerms
imports Basic-Sdf-Tree-Constructors$^{A.2.1}$ Basic-Sdf-Syntax$^{8.3.1}$
Kernel-Sdf-ATerms$^{7.5.4}$
equations
Encoding grammars.

[1] $\text{aterm}(\text{lexical syntax } p^*) = \text{lexical-syntax}(\text{atermlist}(p^*))$
[2] $\text{aterm}(\text{context-free syntax } p^*) = \text{context-free-syntax}(\text{atermlist}(p^*))$

[3] $\text{aterm}(\text{variables } p^*) = \text{variables}(\text{atermlist}(p^*))$
[4] $\text{aterm}(\text{lexical variables } p^*) = \text{lexical-variables}(\text{atermlist}(p^*))$

Encoding symbols.

[5] $\text{aterm}(\langle \mathcal{A}\texttt{-LEX}\rangle) = \text{lex}(\text{aterm}(\mathcal{A}))$
[6] $\text{aterm}(\langle \mathcal{A}\texttt{-CF}\rangle) = \text{cf}(\text{aterm}(\mathcal{A}))$
[7] $\text{aterm}(\langle \mathcal{A}\texttt{-VAR}\rangle) = \text{varsym}(\text{aterm}(\mathcal{A}))$
[8] $\text{aterm}(\texttt{LAYOUT}) = \text{layout}$

Decoding grammars.

[9] $\text{grammar}(\text{lexical-syntax}(Tl)) = \text{lexical syntax productions}(Tl)$
[10] $\text{grammar}(\text{context-free-syntax}(Tl)) = \text{context-free syntax productions}(Tl)$
[11] $\text{grammar}(\text{variables}(Tl)) = \text{variables productions}(Tl)$
[12] $\text{grammar}(\text{lexical-variables}(Tl)) = \text{lexical variables productions}(Tl)$

Decoding symbols.

[13] $\text{symbol}(\text{lex}(T)) = \langle \text{symbol}(T)\texttt{-LEX}\rangle$
[14] $\text{symbol}(\text{cf}(T)) = \langle \text{symbol}(T)\texttt{-CF}\rangle$
[15] $\text{symbol}(\text{varsym}(T)) = \langle \text{symbol}(T)\texttt{-VAR}\rangle$
[16] $\text{symbol}(\text{layout}) = \texttt{LAYOUT}$

A.3 Renamings

module CC-Sdf-Renaming
imports Kernel-Sdf-Renaming$^{9.1.3}$ CC-Sdf-Syntax$^{7.4.2}$
equations
Renaming character classes.

[1] $[cc]\ \rho = cc$

module Literals-Sdf-Renaming
imports Kernel-Sdf-Renaming$^{9.1.3}$ Literals-Sdf-Syntax$^{7.4.3}$
hiddens
  variables
    "L" $\rightarrow$ Literal
equations
Renaming literals.

[1] $[L]\ \rho = L$

module Sorts-Sdf-Renaming
imports Sorts-Sdf-Syntax[7.4.1] Kernel-Sdf-Renaming[9.1.3]
equations
Renaming sorts.

[1] $$[\mathcal{S}]\ \rho = \mathcal{S}$$
[2] $$(\text{sorts}\ \alpha)\ \rho = \text{sorts}\ (\alpha\ )*\ \rho$$

module Priority-Sdf-Renaming
imports Priority-Sdf-Projection[8.1.2] Basic-Sdf-Renaming[A.3]
exports
context-free syntax
"(" Priorities ")" Renamings → Priorities
"(" Group ")$_{\mathrm{G}}$" Renamings → Group
equations
Renaming symbols and productions in priority declarations.

[1] $$(\text{priorities}\ pr^*)\ \rho = \text{priorities}\ (pr^*)\ \rho$$

Lists of priorities.

[2] $$(pr^*)\ \rho = \quad \textbf{when}\ \ pr^* =$$
[3] $$(pr_1^+, pr_2^+)\ \rho = (pr_1^+)\ \rho \mathbin{+\!\!+} (pr_2^+)\ \rho$$

Associativity and priority declarations.

[4] $$(g_1\ \ as\ g_2)\ \rho = (g_1\ )_{\mathrm{G}}\ \rho\ as\ (g_2\ )_{\mathrm{G}}\ \rho$$
[5] $$(g_1 > g_2)\ \rho = (g_1\ )_{\mathrm{G}}\ \rho > (g_2\ )_{\mathrm{G}}\ \rho$$
[6] $$(g_1 > g_2 > gg^+)\ \rho = (g_1\ )_{\mathrm{G}}\ \rho > g_2' > gg'^+ \quad \textbf{when}\ \ g_2' > gg'^+ = (g_2 > gg^+)\ \rho$$

Groups.

[7] $$(g\ )_{\mathrm{G}}\ \rho = p' \quad \textbf{when}\ \ g = p,\ \ p' = (p)\ \rho$$
[8] $$(\{p^*\}\ )_{\mathrm{G}}\ \rho = \{(p^*\ )*\ \rho\}$$
[9] $$(\{as : p^*\}\ )_{\mathrm{G}}\ \rho = \{as : (p^*\ )*\ \rho\}$$

module Regular-Sdf-Renaming
imports Kernel-Sdf-Renaming[9.1.3] Regular-Sdf-Syntax[8.2.1]
equations
Renaming symbols in regular expressions.

[1] $$[(\ )]\ \rho = (\ )$$
[2] $$[(\mathcal{A}\ \alpha^+)]\ \rho = (\mathcal{B}\ \beta^+) \quad \textbf{when}\ \ (\mathcal{A}\ \alpha^+\ )*\ \rho = \mathcal{B}\ \beta^+$$
[3] $$[\mathcal{A}?]\ \rho = (\mathcal{A})\ \rho?$$
[4] $$[\mathcal{A}+]\ \rho = (\mathcal{A})\ \rho+$$

[5] $[\mathcal{A}*]\ \rho = (\mathcal{A})\ \rho*$
[6] $[\{\mathcal{A}\ \mathcal{B}\}+]\ \rho = \{(\mathcal{A})\ \rho\ (\mathcal{B})\ \rho\}+$
[7] $[\{\mathcal{A}\ \mathcal{B}\}*]\ \rho = \{(\mathcal{A})\ \rho\ (\mathcal{B})\ \rho\}*$
[8] $[\{\mathcal{A}\}\ n+]\ \rho = \{(\mathcal{A})\ \rho\}\ n+$
[9] $[\{\mathcal{A}\ \mathcal{B}\}\ n+]\ \rho = \{(\mathcal{A})\ \rho\ (\mathcal{B})\ \rho\}\ n+$
[10] $[\mathrm{Set}[\mathcal{A}]]\ \rho = \mathrm{Set}[(\mathcal{A})\ \rho]$
[11] $[\mathcal{A}\ \#\ \mathcal{B}]\ \rho = (\mathcal{A})\ \rho\ \#\ (\mathcal{B})\ \rho$
[12] $[(\alpha \Rightarrow \mathcal{B})]\ \rho = ((\alpha\ )*\ \rho \Rightarrow (\mathcal{B})\ \rho)$
[13] $[\mathcal{A} \mid \mathcal{B}]\ \rho = (\mathcal{A})\ \rho \mid (\mathcal{B})\ \rho$

module Basic-Sdf-Renaming
imports Basic-Sdf-Normalization$^{8.3.2}$ Kernel-Sdf-Renaming$^{9.1.3}$
exports
context-free syntax
"<" Renamings "`-LEX`" ">" → Renamings
"<" Renamings "`-CF`" ">" → Renamings
equations
Renaming grammars.

[1] (context-free syntax p^*) ρ = context-free syntax $(p^*\ )*\ \rho$
[2] (lexical syntax p^*) ρ = lexical syntax $(p^*\ )*\ \rho$
[3] (variables p^*) ρ = variables $(p^*\ )*\ \rho$
[4] (lexical variables p^*) ρ = lexical variables $(p^*\ )*\ \rho$

Renaming symbols.

[5] $[\langle\mathcal{A}\texttt{-LEX}\rangle]\ \rho = \langle(\mathcal{A})\ \rho\texttt{-LEX}\rangle$
[6] $[\langle\mathcal{A}\texttt{-CF}\rangle]\ \rho = \langle(\mathcal{A})\ \rho\texttt{-CF}\rangle$
[7] $[\langle\mathcal{A}\texttt{-VAR}\rangle]\ \rho = \langle(\mathcal{A})\ \rho\texttt{-VAR}\rangle$
[8] $[\texttt{LAYOUT}]\ \rho = \texttt{LAYOUT}$

Applying $\langle\_\texttt{-LEX}\rangle$ to a renaming.

[9] $\langle[]\texttt{-LEX}\rangle = []$
[10] $\langle[\mathcal{A} \Rightarrow \mathcal{B}]\texttt{-LEX}\rangle = [\langle\mathcal{A}\texttt{-LEX}\rangle \Rightarrow \langle\mathcal{B}\texttt{-LEX}\rangle]$
[11] $\langle[p_1 \Rightarrow p_2]\texttt{-LEX}\rangle = [\langle p_1\texttt{-LEX}\rangle \Rightarrow \langle p_2\texttt{-LEX}\rangle]$
[12] $\langle[\rho_1^+\ \rho_2^+]\texttt{-LEX}\rangle = \langle[\rho_1^+]\texttt{-LEX}\rangle +\!\!+ \langle[\rho_2^+]\texttt{-LEX}\rangle$

Applying $\langle\_\texttt{-CF}\rangle$ to a renaming.

[13] $\langle[]\texttt{-CF}\rangle = []$
[14] $\langle[\mathcal{A} \Rightarrow \mathcal{B}]\texttt{-CF}\rangle = [\langle\mathcal{A}\texttt{-CF}\rangle \Rightarrow \langle\mathcal{B}\texttt{-CF}\rangle]$
[15] $\langle[p_1 \Rightarrow p_2]\texttt{-CF}\rangle = [\langle p_1\texttt{-CF}\rangle \Rightarrow \langle p_2\texttt{-CF}\rangle]$
[16] $\langle[\rho_1^+\ \rho_2^+]\texttt{-CF}\rangle = \langle[\rho_1^+]\texttt{-CF}\rangle +\!\!+ \langle[\rho_2^+]\texttt{-CF}\rangle$

module Restrictions-Sdf-Renaming
imports Restrictions-Sdf-Syntax[8.4.1] Kernel-Sdf-Renaming[9.1.3]
exports
context-free syntax
"(" Restrictions ")" Renamings → Restrictions
equations
Renaming restrictions.

$$[1] \quad (\text{restrictions } \mathit{restr}^*)\ \rho = \text{restrictions } (\mathit{restr}^*)\ \rho$$
$$[2] \quad (\mathit{restr}^*)\ \rho = \quad \textbf{when} \ \ \mathit{restr}^* =$$
$$[3] \quad (\alpha \not\to cc\ \mathit{restr}_1^*)\ \rho = (\alpha\ ){*}\ \rho \not\to cc\ \mathit{restr}_2^* \quad \textbf{when} \ \ \mathit{restr}_2^* = (\mathit{restr}_1^*)\ \rho$$

A.4 SDF2

module Sdf2-Projection
imports Kernel-Sdf-Projection[7.3.2] Sorts-Sdf-Projection[7.4.1]
Priority-Sdf-Projection[8.1.2] Renaming-Sdf-Projection[9.1.2]
Modular-Sdf-Projection[9.3.2] Alias-Sdf-Projection[9.2.2]
Restrictions-Sdf-Projection[8.4.2]

module Sdf2-Renaming
imports Sdf2-Syntax[10.1.1] Kernel-Sdf-Renaming[9.1.3] Priority-Sdf-Renaming[A.3]
Regular-Sdf-Renaming[A.3] Literals-Sdf-Renaming[A.3]
CC-Sdf-Renaming[A.3] Basic-Sdf-Renaming[A.3] Sorts-Sdf-Renaming[A.3]
Restrictions-Sdf-Renaming[A.3] Modular-Sdf-Renaming[9.3.4]
Alias-Sdf-Renaming[9.2.3]
equations

$$[1] \quad (\text{context-free priorities } pr^*)\ \rho = \text{context-free priorities } (pr^*)\ \rho$$
$$[2] \quad (\text{lexical priorities } pr^*)\ \rho = \text{lexical priorities } (pr^*)\ \rho$$

$$[3] \quad [\langle \texttt{Start} \rangle]\ \rho = \langle \texttt{Start} \rangle$$
$$[4] \quad [\langle \texttt{START} \rangle]\ \rho = \langle \texttt{START} \rangle$$

module Sdf2-Tree-Constructors
imports Kernel-Sdf-Tree-Constructors[7.5.2] Basic-Sdf-Tree-Constructors[A.2.1]
Modular-Sdf-Tree-Constructors Regular-Sdf-Tree-Constructors[A.2.1]
Priority-Sdf-Tree-Constructors CC-Sdf-Tree-Constructors[A.2.1]
Sorts-Sdf-Tree-Constructors[A.2.1] Literals-Sdf-Tree-Constructors[A.2.1]

module Sdf2-ATerms

imports Sdf2-Tree-Constructors$^{A.4}$ Sdf2-Syntax$^{10.1.1}$ Kernel-Sdf-ATerms$^{7.5.4}$
Basic-Sdf-ATerms$^{A.2.2}$ Modular-Sdf-ATerms Regular-Sdf-ATerms$^{A.2.2}$
Priority-Sdf-ATerms$^{A.2.2}$ CC-Sdf-ATerms$^{A.2.2}$ Sorts-Sdf-ATerms$^{A.2.2}$
Literals-Sdf-ATerms$^{A.2.2}$ Restrictions-Sdf-ATerms
equations

[1] aterm(⟨`START`⟩) = sort(`"<START>"`)
[2] aterm(⟨`Start`⟩) = sort(`"<Start>"`)

module Sdf2-Trees
imports Sdf2-ATerms$^{A.4}$ Sdf2-Syntax$^{10.1.1}$ Kernel-Sdf-Trees$^{7.5.5}$
Priority-Sdf-Trees$^{8.1.4}$ CC-Sdf-Trees$^{7.5.6}$ Renaming-Sdf-Trees$^{9.1.4}$

module Sdf2-Equality
imports Kernel-Sdf-Equality$^{7.5.8}$ Regular-Sdf-Equality Basic-Sdf-Equality$^{8.3.3}$

B

Auxiliary Modules for Multi-Level Specifications

This appendix contains the specification of several data types and opererations used in the specification of multi-level specifications. The first section contains a couple of standard library modules. The second section defines operations on terms such as substitution, matching, unification and renaming of variables.

B.1 Library Modules

B.1.1 Layout

module Layout
exports
  lexical syntax
    [␣\t\n] → LAYOUT
    "%%"~[\n]* → LAYOUT
    "−−"~[\n]* → LAYOUT

B.1.2 Booleans

module Booleans
imports Layout$^{B.1.1}$
exports
  sorts Bool
  context-free syntax
    "⊤" → Bool
    "⊥" → Bool
    "¬" Bool → Bool
    Bool "∧" Bool → Bool {**assoc**}
    Bool "∨" Bool → Bool {**assoc**}
    "(" Bool ")" → Bool {**bracket**}
  priorities
    "¬"Bool → Bool > Bool "∧"Bool → Bool > Bool "∨"Bool → Bool

variables
$[b][0\text{-}9']* \to$ Bool
equations

[1]	$\top \wedge b = b$	[2]	$\top \vee b = \top$	[3]	$\neg \top = \bot$
[4]	$\bot \wedge b = \bot$	[5]	$\bot \vee b = b$	[6]	$\neg \bot = \top$

B.1.3 Error Booleans

Boolean predicates are either true or false. In case of type checking this is not appropriate. In case the predicate does not hold a more refined value than false should be returned that explains the cause of the error. Error Booleans are a refinement of the normal Booleans with a true value $\top$ and a sort Error to represent the false values.

Errors The error $e_1; e_2$ indicates that both errors e_i occurred. The error $e_1 : e_2$ indicates that error e_1 occurred and that e_2 is an explanation of that error; as in

```
equation "(X :: L) ++ L' == X :: (L1 ++ L2)" not well-formed:
variables "L1; L2" of rhs do not occur in lhs
```

module Error-Booleans
imports Layout$^{B.1.1}$ Booleans$^{B.1.2}$
exports
sorts Error
context-free syntax
Error ";" Error → Error {**right**}
Error ":" Error → Error {**right**}
"if" Bool "then" Error "else" Error → Error
"(" Error ")" → Error {**bracket**}
priorities
"if" Bool "then" Error "else" Error → Error > Error ":" Error → Error > Error ";" Error → Error
equations

[1] $(e_1;\, e_2);\, e_3 = e_1;\, e_2;\, e_3$
[2] $(e_1 : e_2) : e_3 = e_1 : (e_2;\, e_3)$
[3] if $\top$ then e_1 else $e_2 = e_1$
[4] if $\bot$ then e_1 else $e_2 = e_2$

Error Booleans An error Boolean value is either $\top$ (correct, true) or an error. The place normally taken by the value false is here represented by the sort of errors. Since it is unclear which error should be indicated by the negation of

$\top$, we do not provide negation. The operations on EBool are $\wedge$, $\rightsquigarrow$ and $::$. The operator $\wedge$ is a symmetric conjunction that yields $\top$ if both arguments do and otherwise the conjunction of the errors. The operator $\rightsquigarrow$ is an assymetric conjunction that prefers the error in its first argument discarding the error in its second. This operator should be used to indicate a dependency between errors. If the well-formedness of a construct depends on the well-formedness of its subconstructs and some conditions, then one can express that the errors in the subconstructs are more important. Finally, the operator $::$ has $\top$ as right zero and as left unit. If both arguments are errors it yields the explanation of the first by the second.

exports
  sorts EBool
  context-free syntax
    "$\top$" $\rightarrow$ EBool
    Error $\rightarrow$ EBool
    EBool "::" EBool $\rightarrow$ EBool {**right**}
    EBool "$\rightsquigarrow$" EBool $\rightarrow$ EBool {**right**}
    EBool "$\wedge$" EBool $\rightarrow$ EBool {**right**}
    "if" Bool "then" EBool "else" EBool $\rightarrow$ EBool
    "(" EBool ")" $\rightarrow$ EBool {**bracket**}
  priorities
    "if"Bool "then"EBool "else"EBool $\rightarrow$ EBool >
    EBool "::"EBool $\rightarrow$ EBool > EBool "$\rightsquigarrow$"EBool $\rightarrow$ EBool >
    EBool "$\wedge$"EBool $\rightarrow$ EBool
  variables
    "e"[*0-9'*]* $\rightarrow$ Error
    "eb"[*0-9'*]* $\rightarrow$ EBool
equations

[5] $\top \wedge eb = eb$
[6] $eb \wedge \top = eb$
[7] $e_1 \wedge e_2 = e_1 ; e_2$
[8] $\top \rightsquigarrow eb = eb$
[9] $e \rightsquigarrow eb = e$
[10] $eb :: \top = \top$
[11] $\top :: eb = eb$
[12] $e_1 :: e_2 = e_1 : e_2$
[13] $\text{if } \top \text{ then } eb_1 \text{ else } eb_2 = eb_1$
[14] $\text{if } \bot \text{ then } eb_1 \text{ else } eb_2 = eb_2$

B.1.4 Naturals

```
module Naturals
imports Booleans^B.1.2
exports
  sorts Nat
  lexical syntax
    [0-9]+ → Nat
  context-free syntax
    succ(Nat)          → Nat
    pred(Nat)          → Nat
    Nat "+" Nat        → Nat   {left}
    max(Nat, Nat)      → Nat
    zero(Nat)          → Bool
    eq(Nat, Nat)       → Bool
  variables
    [mn][0-9′]*   → Nat
    "c*"[0-9′]*   → CHAR*
    "c+"[0-9′]*   → CHAR+
```

The usual equations for the natural numbers are not shown.

B.2 Term Utilities

In this section we define several data types and operations on terms.

B.2.1 Binary Operators

```
module Binary-Operators
imports Types^12.2.2 Terms^12.1.1
exports
  sorts BinOp
  lexical syntax
    ~[′␣\t\n%()\[\].]*
    ~[a-zA-Z0-9′␣\t\n%()\[\] <>,._]~[′␣\t\n%()\[\]]* → BinOp
  context-free syntax
    "(" BinOp ")"                → Fun
    "[" "]"                      → Fun
    "[" Term "]"                 → Term
    "⟨" "⟩"                      → Fun
    "⟨" Term "⟩"                 → Term
    Term BinOp Term              → Term   {non-assoc}
    Term "∘" Term "∘" Term       → Term   {non-assoc}
  priorities
    Term Term → Term > {non-assoc: Term BinOp Term → Term,
    Term "∘"Term "∘"Term → Term} > Term "×"Term → Term
```

variables
"$\oplus$"[*0-9'*]$*$ $\rightarrow$ BinOp
equations

[1] $t_1 \oplus t_2 = (\oplus)\ (t_1,\, t_2)$
[2] $t_1\ .t_2.\ t_3 = t_2\ (t_1,\, t_3)$
[3] $[t_1,\, t_2] = t_1\ ::\ [t_2]$
[4] $[t] = t :: [\ ]$ **otherwise**
[5] $\langle t_1,\, t_2\rangle = t_1\ \hat{}\ \langle t_2\rangle$
[6] $\langle t\rangle = t\ \hat{}\ \langle\ \rangle$ **otherwise**

B.2.2 Errors over Terms and Signatures

To provide errors that convey information related to terms and equations we define several error constructors. An example error is

```
function "(+)" not declared
```

module SPEC-Errors
imports Error-Booleans$^{B.1.3}$ OLS$^{12.2.4}$
exports
context-free syntax
"\"" Term "\"" not a well-formed sort declaration $\rightarrow$ Error
sort "\"" Term "\"" not declared $\rightarrow$ Error
"\"" Term "\"" not a well-formed sort $\rightarrow$ Error
sort "\"" Term "\"" matches no sort declaration $\rightarrow$ Error

function "\"" Term "\"" multiply declared $\rightarrow$ Error
variable "\"" Term "\"" multiply declared $\rightarrow$ Error
function "\"" Term "\"" not declared $\rightarrow$ Error
function "\"" Term "\""
with type "\"" Term "\"" not declared $\rightarrow$ Error
variable "\"" Term "\"" not declared $\rightarrow$ Error

term "\"" Term "\"" not well-formed $\rightarrow$ Error
pair "\"" Term "\"" not well-formed $\rightarrow$ Error
application "\"" Term "\"" not well-formed $\rightarrow$ Error
product "\"" Term "\"" not well-formed $\rightarrow$ Error
arrow "\"" Term "\"" not well-formed $\rightarrow$ Error
annotation of "\"" Term "\""
with "\"" Term "\"" not well-formed $\rightarrow$ Error

"\"" Term "\"" is not a function $\rightarrow$ Error
type of argument "\"" Term "\""
does not match type of domain "\"" Term "\"" $\rightarrow$ Error
type of result "\"" Term "\""

does not match type of codomain “\”” Term “\”” → Error

no declaration for function “\”” Term “\””
with type “\”” Term “\”” → Error

equation “\”” Eq “\”” not well-formed → Error
types do not match → Error
“variables” “\”” Terms “\”” of rhs not in lhs → Error

level “\”” Nat “\”” → Error
should be “\”” Term “\”” → Error

type “\”” Term “\”” of variable “\”” Term “\””
incompatible with declaration → Error
type “\”” Term “\”” of function “\”” Term “\””
incompatible with declaration → Error
type is “\”” Term “\”” → Error
types of variable “\”” Term “\”” incompatible
“\”” Term “\”” versus “\”” Term “\”” → Error

errors in “module” “\”” Term “\”” → Error

B.2.3 Term Functions

The sort (Term ⇒ Term) represents functions from terms to terms. The sort is defined in order to reuse several common higher-order operations such as function composition and mapping a function over a list. Furthermore, we define a conditional for terms, list membership, and term equality.

module Term-Functions
imports Terms[12.1.1] Booleans[B.1.2] Types[12.2.2]
exports
sorts (Term ⇒ Term)
context-free syntax
(Term ⇒ Term) “(” Term “)” → Term
“id” → (Term ⇒ Term)
(Term ⇒ Term) “∘” (Term ⇒ Term) → (Term ⇒ Term) {**assoc**}
“if” Bool “then” Term “else” Term → Term
(Term ⇒ Term) “∗” “(” Terms “)” → Terms
eq(Term, Term) → Bool
Term “∈” Terms → Bool
variables
“ρ”[*0-9′*]∗ → (Term ⇒ Term)
equations

[1] $\mathrm{id}(t) = t$

[2] $\rho_1 \circ \rho_2(t) = \rho_2(\rho_1(t))$

[3] $\mathrm{id} \circ \rho = \rho$
[4] $\rho \circ \mathrm{id} = \rho$
[5] $\rho{*}() =$
[6] $\rho{*}(t) = \rho(t)$
[7] $\rho{*}(t_1^+;\ t_2^+) = \rho{*}(t_1^+) +\!\!+ \rho{*}(t_2^+)$
[8] $\text{if } \top \text{ then } t_1 \text{ else } t_2 = t_1$
[9] $\text{if } \bot \text{ then } t_1 \text{ else } t_2 = t_2$
[10] $\mathrm{eq}(t,\ t) = \top$
[11] $\mathrm{eq}(t,\ t') = \bot$ **otherwise**
[12] $t \in\ = \bot$
[13] $t \in t' = \mathrm{eq}(t,\ t')$
[14] $t \in t_1^+;\ t_2^+ = t \in t_1^+ \vee t \in t_2^+$

B.2.4 Equation Functions

Map (Term ⇒ Term) functions over equations and lists of equations.

module Equation-Functions
imports Term-Functions$^{B.2.3}$ Equations$^{12.1.2}$
exports
  context-free syntax
    (Term ⇒ Term) "(" Eq ")" → Eq
    (Term ⇒ Term) "*e" "(" Eqs ")" → Eqs
    "if" Bool "then" Eqs "else" Eqs → Eqs
equations

[1] $\rho(t_1 \equiv t_2) = \rho(t_1) \equiv \rho(t_2)$
[2] $\rho{*}e(\varphi^*) =$ **when** $\varphi^* =$
[3] $\rho{*}e(\varphi;\ \varphi^*) = \rho(\varphi) +\!\!+ \rho{*}e(\varphi^*)$
[4] $\text{if } \top \text{ then } \mathcal{E}_1 \text{ else } \mathcal{E}_2 = \mathcal{E}_1$
[5] $\text{if } \bot \text{ then } \mathcal{E}_1 \text{ else } \mathcal{E}_2 = \mathcal{E}_2$

B.2.5 Term Sets

The function {_} creates a 'set' of terms from a list of terms by removing the duplicates from the list. The usual operations on sets are union ($\cup$), intersection ($\cap$), difference (/), emptiness ('empty'), membership ($\in$) and subset ($\subseteq$). The Cartesian product $\times$ yields the set of pairs of the elements of two sets.

module Term-Sets
imports Term-Functions$^{B.2.3}$ Terms$^{12.1.1}$ Booleans$^{B.1.2}$
exports
  sorts TermSet

context-free syntax

"{" Terms "}"	$\rightarrow$ TermSet	
TermSet "$\cup$" TermSet	$\rightarrow$ TermSet	{**left**}
TermSet "$\cap$" TermSet	$\rightarrow$ TermSet	{**left**}
TermSet "/" TermSet	$\rightarrow$ TermSet	{**left**}
TermSet "$\times$" TermSet	$\rightarrow$ TermSet	{**right**}
(Term $\Rightarrow$ Term) "$*$" "(" TermSet ")"	$\rightarrow$ TermSet	
"if" Bool "then" TermSet "else" TermSet	$\rightarrow$ TermSet	
trms(TermSet)	$\rightarrow$ Terms	
"(" TermSet ")"	$\rightarrow$ TermSet	{**bracket**}
empty(TermSet)	$\rightarrow$ Bool	
Term "$\in$" TermSet	$\rightarrow$ Bool	
TermSet "$\subseteq$" TermSet	$\rightarrow$ Bool	

priorities

TermSet "$\times$"TermSet $\rightarrow$ TermSet $>$ TermSet "/"TermSet $\rightarrow$ TermSet
$>$ TermSet "$\cap$"TermSet $\rightarrow$ TermSet $>$ TermSet "$\cup$"TermSet $\rightarrow$ TermSet
$>$ "if"Bool "then"TermSet "else"TermSet $\rightarrow$ TermSet

variables

"Φ"[*0-9'*]$*$ $\rightarrow$ TermSet

equations

[1] $\{t_1^*;\ t;\ t_2^*;\ t;\ t_3^*\} = \{t_1^*;\ t;\ t_2^*;\ t_3^*\}$

[2] $\{t_1^*\} \cup \{t_2^*\} = \{t_1^*;\ t_2^*\}$

[3] $\{\} \cap \Phi = \{\}$

[4] $\{t_1^+;\ t_2^+\} \cap \Phi = \{t_1^+\} \cap \Phi \cup \{t_2^+\} \cap \Phi$

[5] $\{t\} \cap \Phi = \text{if } t \in \Phi \text{ then } \{t\} \text{ else } \{\}$

[6] $\{\}\ /\ \Phi = \{\}$

[7] $\{t_1^+;\ t_2^+\}\ /\ \Phi = \{t_1^+\}\ /\ \Phi \cup \{t_2^+\}\ /\ \Phi$

[8] $\{t\}\ /\ \Phi = \text{if } t \in \Phi \text{ then } \{\} \text{ else } \{t\}$

[9] $\{t_1\} \times \{t_2\} = \{t_1,\ t_2\}$

[10] $\{\} \times \Phi = \{\}$

[11] $\{t_1^+;\ t_2^+\} \times \Phi = \{t_1^+\} \times \Phi \cup \{t_2^+\} \times \Phi$

[12] $\Phi \times \{\} = \{\}$

[13] $\Phi \times \{t_1^+;\ t_2^+\} = \Phi \times \{t_1^+\} \cup \Phi \times \{t_2^+\}$

[14] $\rho*(\{ts\}) = \{\rho*(ts)\}$

[15] $\text{if } \top \text{ then } \Phi_1 \text{ else } \Phi_2 = \Phi_1$

[16] $\text{if } \bot \text{ then } \Phi_1 \text{ else } \Phi_2 = \Phi_2$

[17] $\text{trms}(\{ts\}) = ts$

[18] $\text{empty}(\{\}) = \top$

[19] $\text{empty}(\{t^+\}) = \bot$

[20] $t \in \{ts\} = t \in ts$

[21] $\{\} \subseteq \Phi = \top$

[22] $\{t\} \subseteq \Phi = t \in \Phi$

[23] $\{t_1^+;\ t_2^+\} \subseteq \Phi = \{t_1^+\} \subseteq \Phi \wedge \{t_2^+\} \subseteq \Phi$

B.2.6 Variables

To extract the variables from a term a family of functions is defined. The functions differ in their treatment of variables and the type annotation operator :, but share their definition for the other operators. To prevent copying the same equations for the four functions, the function names are put in a sort. The generic part of the definition is expressed by means of a 'variable function name' *vs*. The functions are 'var' that yields the set of *all* variables in a term, 'tvars' that yields the set of all type variables, i.e., variables occurring in annotations, 'ovars' that yields all 'object variables', i.e., variables that are not in type annotations, and 'avars' that yields all object variables with their annotation.

module Variables
imports Term-Sets$^{B.2.5}$
exports
  sorts Vars
  context-free syntax

vars	$\to$ Vars
tvars	$\to$ Vars
avars	$\to$ Vars
ovars	$\to$ Vars
Vars "(" Term ")"	$\to$ TermSet
Vars "*(" Terms ")"	$\to$ TermSet
Vars "(" TermSet ")"	$\to$ TermSet

  variables
    "*vs*" $\to$ Vars
equations

[1] $vs(f) = \{\}$
[2] $vs(\text{nil}) = \{\}$
[3] $vs(\text{top}) = \{\}$
[4] $vs(t_1, t_2) = vs(t_1) \cup vs(t_2)$
[5] $vs(t_1\ t_2) = vs(t_1) \cup vs(t_2)$
[6] $vs(t_1 \times t_2) = vs(t_1) \cup vs(t_2)$
[7] $vs(t_1 \to t_2) = vs(t_1) \cup vs(t_2)$
[8] $vs(\{ts\}) = vs*(\ ts)$
[9] $vs*(\ ) = \{\}$
[10] $vs*(\ t) = vs(t)$
[11] $vs*(\ t_1^+;\ t_2^+) = vs*(\ t_1^+) \cup vs*(\ t_2^+)$

[12] $\text{vars}(x) = \{x\}$
[13] $\text{vars}(t : \tau) = \text{vars}(t) \cup \text{vars}(\tau)$
[14] $\text{ovars}(x) = \{x\}$

[15] $\mathrm{ovars}(t : \tau) = \mathrm{ovars}(t)$
[16] $\mathrm{tvars}(x) = \{\}$
[17] $\mathrm{tvars}(t : \tau) = \mathrm{tvars}(t) \cup \mathrm{vars}(\tau)$
[18] $\mathrm{avars}(x) = \{\}$
[19] $\mathrm{avars}(x : \tau) = \{x : \tau\}$
[20] $\mathrm{avars}(t : \tau) = \mathrm{avars}(t)$ **otherwise**

B.2.7 Substitution

A substitution is a mapping from variables to terms. When applied to a term all variables occurring in the domain of the substitution are replaced by their result in the substitution. A finite substitution maps only a finite number of variables to other terms than themselves. Finite substitutions are represented by a list of atomic substitutions of the form $x := t$, which express the mapping from variable x to term t. Note that [] is the empty substitution. The application $\sigma(t)$ of a substitution σ to a term t denotes t with each occurrence of a variable x in t replaced by $\sigma(x)$. The union (+) of two substitutions is simply the concatenation of their lists of atomic substitutions. If a conflict arises, i.e., both substitutions contain an assignment to the same variable, the assignment in the first substitution has priority over the second as a result of the definition of $\sigma(x)$ in equations [1,2,3].

module Substitution
imports Term-Functions$^{B.2.3}$ Terms$^{12.1.1}$ Types$^{12.2.2}$
exports
  sorts ASubst Subst
  context-free syntax

Var ":=" Term	$\rightarrow$ ASubst	
"[" ASubst∗ "]"	$\rightarrow$ Subst	
Subst	$\rightarrow$ (Term $\Rightarrow$ Term)	
"$\Downarrow$"((Term $\Rightarrow$ Term))	$\rightarrow$ Subst	
Subst "+" Subst	$\rightarrow$ Subst	{**assoc**}
"(" Subst ")"	$\rightarrow$ Subst	{**bracket**}

  variables

"*as*"[*0-9′*]∗	$\rightarrow$ ASubst
"*as*""∗"[*0-9′*]∗	$\rightarrow$ ASubst∗
"*as*""+"[*0-9′*]∗	$\rightarrow$ ASubst+
"σ"[*0-9′*]∗	$\rightarrow$ Subst

equations

[1] $[x := t\ as^*](x) = t$
[2] $[y := t\ as^*](x) = [as^*](x)$ **when** $\mathrm{eq}(x, y) = \perp$
[3] $[](x) = x$
[4] $\sigma(f) = f$

[5] $\sigma(\text{nil}) = \text{nil}$
[6] $\sigma(\text{top}) = \text{top}$
[7] $\sigma(t_1,\, t_2) = \sigma(t_1),\, \sigma(t_2)$
[8] $\sigma(t_1\ t_2) = \sigma(t_1)\ \sigma(t_2)$
[9] $\sigma(t \times \tau) = \sigma(t) \times \sigma(\tau)$
[10] $\sigma(t \to \tau) = \sigma(t) \to \sigma(\tau)$
[11] $\sigma(t : \tau) = \sigma(t) : \sigma(\tau)$
[12] $\Downarrow(\sigma) = \sigma$
[13] $\Downarrow(\text{id}) = []$
[14] $[as_1^*] + [as_2^*] = [as_1^*\ as_2^*]$
[15] $[] \circ \sigma = \sigma$
[16] $\sigma \circ [] = \sigma$
[17] $\sigma \circ [x := t\ as^*] = [x := \sigma(t)] + \sigma \circ [as^*]$

Failure Substitutions A failure substitution is a substitution or the value $\perp$ (fail), which denotes failure for partial functions producing substitutions like matching and unification. The operation $+_\perp$ is the strict extension of $+$ to failure substitutions. The operation is the consistent composition of two substitutions. Two substitutions are consistent if they coincide on the same variable or the variable is undefined in one or both substitutions.

sorts $\text{Subst}_\perp$
context-free syntax
Subst $\to \text{Subst}_\perp$
"$\perp$" $\to \text{Subst}_\perp$
$\text{Subst}_\perp$ "$+_\perp$" $\text{Subst}_\perp$ $\to \text{Subst}_\perp$ {**non-assoc**}
$\text{Subst}_\perp$ "$\oplus$" $\text{Subst}_\perp$ $\to \text{Subst}_\perp$ {**non-assoc**}
$\text{Subst}_\perp$ "$\circ_\perp$" $\text{Subst}_\perp$ $\to \text{Subst}_\perp$ {**non-assoc**}
"if" Bool "then" $\text{Subst}_\perp$ "else" $\text{Subst}_\perp$ $\to \text{Subst}_\perp$
"fail?"($\text{Subst}_\perp$) $\to$ Bool
"$\Downarrow_\perp$"($\text{Subst}_\perp$) $\to$ Subst
"(" $\text{Subst}_\perp$ ")" $\to \text{Subst}_\perp$ {**bracket**}
variables
"$\sigma_\perp$"[$'$]* $\to \text{Subst}_\perp$
priorities
{**non-assoc**: $\text{Subst}_\perp$ "$+_\perp$"$\text{Subst}_\perp \to \text{Subst}_\perp$, $\text{Subst}_\perp$ "$\oplus$"$\text{Subst}_\perp \to \text{Subst}_\perp$,
$\text{Subst}_\perp$ "$\circ_\perp$"$\text{Subst}_\perp \to \text{Subst}_\perp$} > "if" Bool "then"$\text{Subst}_\perp$ "else"$\text{Subst}_\perp \to \text{Subst}_\perp$

equations

[18] if $\top$ then $\sigma_\perp$ else $\sigma_\perp{}' = \sigma_\perp$
[19] if $\perp$ then $\sigma_\perp$ else $\sigma_\perp{}' = \sigma_\perp{}'$
[20] $\sigma_1 +_\perp \sigma_2 = \sigma_1 + \sigma_2$
[21] $\sigma_\perp +_\perp \perp = \perp$
[22] $\perp +_\perp \sigma_\perp = \perp$
[23] $\perp \oplus \sigma_\perp = \perp$

[24] $\sigma_\perp \oplus \perp = \perp$

[25] $[] \oplus \sigma_\perp = \sigma_\perp$

[26] $\sigma_\perp \oplus [] = \sigma_\perp$

[27] $[x := t\ as^*] \oplus \sigma = \text{if eq}(t', x) \vee \text{eq}(t', t)$ $\text{then } [x := t] +_\perp ([as^*] \oplus \sigma)$ $\text{else } \perp$ **when** $\sigma(x) = t'$

[28] $\perp \circ_\perp \sigma_\perp = \perp$

[29] $\sigma_\perp \circ_\perp \perp = \perp$

[30] $\sigma_1 \circ_\perp \sigma_2 = \Downarrow(\sigma_1 \circ \sigma_2)$

[31] $\text{fail?}(\sigma) = \perp$

[32] $\text{fail?}(\perp) = \top$

[33] $\Downarrow_\perp(\sigma) = \sigma$

B.2.8 Matching

A term t matches with a pattern term t', notation $t' := t$, if there exists a substitution σ such that $\sigma(t') = t$. If t matches t', t' is said to more general than t, which is expressed by means of the predicate $\geq$ as $t' \geq t$. If $t' \geq t$ we also say that t is an instance of t'. This relation gives a partial order on terms. A substitution σ is a *renaming* if $\sigma(t) \doteq t$ for any t.

module Matching
imports Substitution$^{B.2.7}$ Term-Sets$^{B.2.5}$
exports
context-free syntax
Terms ":=" Terms $\to$ Subst$_\perp$
Term "$\geq$" Term $\to$ Bool
Term "$>$" Term $\to$ Bool
Term "$\doteq$" Term $\to$ Bool
TermSet "$\geq$" Term $\to$ Bool
equations

[1] $x := t = [x := t]$

[2] $t := t = []$

[3] $t_1, t_2 := t_3, t_4 = t_1; t_2 := t_3; t_4$

[4] $t_1\ t_2 := t_3\ t_4 = t_1; t_2 := t_3; t_4$

[5] $t_1 \times t_2 := t_3 \times t_4 = t_1; t_2 := t_3; t_4$

[6] $t_1 \to t_2 := t_3 \to t_4 = t_1; t_2 := t_3; t_4$

[7] $t_1 : t_2 := t_3 : t_4 = t_1; t_2 := t_3; t_4$

[8] $:= = []$

[9] $t_1; t_1^+ := t_2; t_2^+ = t_1 := t_2 \oplus t_1^+ := t_2^+$

[10] $t := t' = \perp$ **otherwise**

[11] $t_1 \geq t_2 = \neg\ \text{fail?}(t_1 := t_2)$

[12] $t_1 > t_2 = t_1 \geq t_2 \wedge \neg\, t_2 \geq t_1$
[13] $t_1 \doteq t_2 = t_1 \geq t_1 \wedge t_2 \geq t_1$
[14] $\{\} \geq t = \bot$
[15] $\{t'\} \geq t = t' \geq t$
[16] $\{t_1^+;\, t_2^+\} \geq t = \{t_1^+\} \geq t \vee \{t_2^+\} \geq t$

B.2.9 Unification

Two terms t_1 and t_2 are unifiable if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$. The function 'mgu' yields the most general unifier σ for a set of equations $\mathcal{E}$, such that for each equation $t_1 \equiv t_2$ in $\mathcal{E}$, $\sigma(t_1) = \sigma(t_2)$. The definition is based on the algorithm by Martelli and Montanari (1982). Hendriks (1989) specifies in a similar manner the unification of types in ML. See also Jouannaud and Kirchner (1991) for a survey on unification.

module Unification
imports Variables$^{B.2.6}$ Substitution$^{B.2.7}$ Equation-Functions$^{B.2.4}$
exports
  context-free syntax
    mgu(Eqs) $\rightarrow$ Subst$_\bot$
    Term "$\overset{?}{=}$" Term $\rightarrow$ Bool
equations

[1] $\text{mgu}(t \equiv t) = [\,]$
[2] $\text{mgu}(x \equiv t) = [x := t]$ **when** $x \in \text{vars}(t) = \bot$
[3] $\text{mgu}(t \equiv x) = [x := t]$ **when** $x \in \text{vars}(t) = \bot$
[4] $\text{mgu}(t_1,\, t_2 \equiv t_3,\, t_4) = \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4)$
[5] $\text{mgu}(t_1\ t_2 \equiv t_3\ t_4) = \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4)$
[6] $\text{mgu}(t_1 \times t_2 \equiv t_3 \times t_4) = \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4)$
[7] $\text{mgu}(t_1 \rightarrow t_2 \equiv t_3 \rightarrow t_4) = \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4)$
[8] $\text{mgu}(t_1 : t_2 \equiv t_3 : t_4) = \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4)$
[9] $\text{mgu}() = [\,]$
[10] $\text{mgu}(\varphi_1^+;\, \varphi_2^+) = \text{mgu}(\Downarrow_\bot(\sigma_\bot) * e(\varphi_2^+)) \circ_\bot \sigma_\bot$
  when $\text{mgu}(\varphi_1^+) = \sigma_\bot$
[11] $\text{mgu}(\mathcal{E}) = \bot$ **otherwise**
[12] $t_1 \overset{?}{=} t_2 = \neg\, \text{fail?}(\text{mgu}(t_1 \equiv t_2))$

B.2.10 Renaming

It is sometimes necessary to rename variables in a term such that they are disjunct from the variables in another term. To this end several functions are defined to generate new variable names. The function get-fresh produces a fresh variable (not occurring in some set of variables). The function rn $\Phi_1[\Phi_2]$, with Φ_1 and Φ_2 sets of variables, yields a substitution σ_1 that renames the variables in Φ_1 such that none occurs in Φ_2, i.e., $\sigma_1(\Phi_1) \cap \Phi_2 = \emptyset$. The other 'rn' function renames the variables of a term with respect to (the variables of) another term.

module Renaming
imports Variables$^{B.2.6}$ Substitution$^{B.2.7}$
exports
  context-free syntax
    prime(Var) $\rightarrow$ Var
    deprime(Var) $\rightarrow$ Var
    base(Var) $\rightarrow$ Var
    get-fresh(Var, TermSet) $\rightarrow$ Term
    rn TermSet "[" TermSet "]" $\rightarrow$ Subst
    rn Term "[" Term "]" $\rightarrow$ Term
    add(Var, TermSet) $\rightarrow$ TermSet
hiddens
  variables
    "*c*+"[*0-9'*]* $\rightarrow$ CHAR+
equations

[1] $\text{prime}(\text{var}(c^+)) = \text{var}(c^+\ \texttt{"'"})$

[2] $\text{deprime}(\text{var}(c^+\ \texttt{"'"})) = \text{deprime}(\text{var}(c^+))$

[3] $\text{deprime}(x) = x$ **otherwise**

The function 'base' takes off all trailing digits and primes of a variable. The equations for the function 'base' are not shown.

[4] $\text{add}(\text{var}(c_1^+),\ \{\text{var}(c_2^+)\}) = \{\text{var}(c_1^+\ c_2^+)\}$

[5] $\text{add}(x,\ \{\}) = \{x\}$

[6] $\text{add}(x,\ \{t_1^+;\ t_2^+\}) = \text{add}(x,\ \{t_1^+\}) \cup \text{add}(x,\ \{t_2^+\})$

[7] $\text{add}(x,\ \{t\}) = \{x\}$ **otherwise**

[8] $\text{get-fresh}(x,\ \Phi) = \text{if } x \in \Phi \text{ then get-fresh}(\text{prime}(x),\ \Phi) \text{ else } x$

[9] $\text{rn } \{\}[\Phi] = []$

[10] $\text{rn } \{x;\ t^*\}[\Phi] = [x := y] + \text{rn } \{t^*\}[\{y\} \cup \Phi]$
when $\text{get-fresh}(\text{deprime}(x),\ \Phi) = y$

Rename a term with respect to the variables in another term.

[11] $\text{rn } t_1[t_2] = \text{rn } \Phi_1 \cap \Phi_2[\Phi_2](t_1)$
when $\text{vars}(t_1) = \Phi_1,\ \text{vars}(t_2) = \Phi_2$

C

Samenvatting

C.1 Algemeen

Computertalen Computertalen worden gebruikt voor het schrijven van computerprogramma's, maar ook voor het beschrijven van data en specificaties. Voortdurend worden nieuwe computertalen ontworpen. De opkomst van nieuwe technologie stelt andere eisen aan talen. Bijvoorbeeld, de recente groei en ontwikkeling van het internet heeft de doorbraak van de taal Java tot gevolg gehad. De ontwikkeling van talen voor speciale toepassingsgebieden vereenvoudigt de programmatuur voor die gebieden. Nieuwe inzichten op het gebied van programmeertaaltechnieken kunnen ook aanleiding zijn voor de ontwikkeling van een nieuwe taal. Kinnersley (1995) geeft een lijst van zo'n 2350 computertalen die ooit, dus sinds ongeveer 1940, ontwikkeld zijn. Aangezien deze lijst vermoedelijk slechts het topje van een ijsberg weergeeft, is het veilig om te stellen dat er iedere week een nieuw ontworpen taal bijkomt.

Het ontwikkelen van een nieuwe computertaal is veel werk. Door het snel ontwikkelen van een prototype van de taal, de rudimentaire vorm van de uiteindelijke taal waarin de belangrijkste eigenschappen naar voren komen, kunnen de ideeën die aan de taal ten grondslag liggen getoetst worden en zonodig bijgesteld. Ontwerpgereedschappen kunnen het ontwikkelen ondersteunen door routinewerk uit handen van de ontwerper te nemen. Dit soort gereedschappen zijn het onderwerp van dit proefschrift. We bespreken hier eerst een aantal basisbegrippen met betrekking tot het prototyperen van talen. Vervolgens geven we een samenvatting van de bijdrage die dit proefschrift levert.

Syntax en Semantiek Het ontwerp van een computertaal bestaat uit een beschrijving van de syntaxis en semantiek van de taal. De syntaxis (op z'n engels van af nu syntax) van een taal beschrijft de vorm van de zinnen waaruit de taal bestaat en kent aan die zinnen structuur toe. In het geval van een programmeertaal heet een zin een programma, in het geval van een specificatietaal een specificatie, etc. De semantiek beschrijft de betekenis van syntactisch correcte zinnen. Niet alle zinnen hebben altijd een semantiek, dat wil zeggen, een zinvolle betekenis.

Als voorbeeld volgt hier de beschrijving van de syntax en semantiek van de eenvoudige taal van rekenkundige expressies met optelling en product. De syntax van expressies wordt beschreven door de volgende regels: (1) Een variable is een expressie. (2) Een getal is een expressie. (3) Als x en y expressies zijn dan ook $x * y$ en $x + y$. Volgens deze regels is $x * y + 3$ een expressie. De semantiek van expressies definiëren we nu als de waarde van een expressie onder de toekenning van getallen aan variabelen. (1) De waarde van een variabele is die welke eraan is toegekend. (2) De waarde van een getal is dat getal zelf. (3) Als w_1 de waarde is van x en w_2 de waarde van y dan is het product van w_1 en w_2 de waarde van $x * y$ en de som van w_1 en w_2 de waarde van $x + y$.

Ambiguiteit De syntax van een taal beschrijft uit welke zinnen een taal bestaat en kent daarnaast aan die zinnen een structuur toe die vaak in de vorm van een boomdiagram voorgesteld wordt. Een zin is ambigu als de syntax er meer dan één structuur aan toekent. Bijvoorbeeld, volgens de syntax regels hierboven heeft $x * y + z$ twee mogelijke structuren: $x * (y + z)$ of $(x * y) + z$. Daarmee heeft de zin ook twee betekenissen: het product van x en de som van y en z, of de som van het product van x en y en z.

Disambiguatie is het oplossen van dergelijke ambiguiteiten. Een disambiguatieregel is een regel voor het oplossen ambiguiteiten. De gebruikelijke regel voor de disambiguatie van rekenkundige expressies wordt gegeven door de bekende regel 'meneer van Dalen wacht op antwoord'. Met dit ezelsbruggetje wordt uitgedrukt dat operatoren sterker binden als ze eerder voorkomen in het rijtje 'machtsverheffen, vermenigvulden, delen, optellen, aftrekken'. In het voorbeeld hierboven wordt dus de interpretatie $(x * y) + z$ gekozen volgens deze regel.

Syntax Definitie Formalismen Voor het ontwikkelen van een prototype van een taal die door een computer leesbaar is, worden de syntax en semantiek in de vorm van een formele definitie gegeven. Een syntax definitie formalisme is een formele taal waarin de regels van de syntax opgeschreven kunnen worden. Voor de definitie van de syntax van talen worden gewoonlijk context-vrije grammatica's gebruikt. Een context-vrije regel van de vorm $A_1 \ldots A_n \rightarrow A_0$ bepaalt dat uit een rijtje zinnen van type A_1 tot en met A_n een zin van type A_0 te maken valt. De volgende context-vrije grammatica beschrijft bijvoorbeeld de syntax van expressies zoals hierboven besproken.

```
Var         -> Exp
Num         -> Exp
Exp "*" Exp -> Exp
Exp "+" Exp -> Exp
```

Net zoals hierboven zijn deze regels ambigu. Om de gedefinieerde taal te disambigueren kunnen syntax definitie formalismen methoden verschaffen waarmee disambiguatieregels geformuleerd kunnen worden. De regel die stelt dat vermenigvuldiging sterker bindt dan optelling kan bijvoorbeeld uitgedrukt worden door de volgende prioriteitsregel.

```
Exp "*" Exp -> Exp > Exp "+" Exp -> Exp
```

Een syntax definitie formalisme kan veel meer faciliteiten bieden om te disambigueren of om definities korter te maken. Het geheel van dit soort faciliteiten bepaalt de uitdrukkingskracht en de bruikbaarheid van het formalisme.

Algebraïsche Specificaties Er zijn vele varianten van syntax definitie formalismen. In het algemeen kan een formalisme worden gekarakteriseerd door: de structuren (bomen) die uit een definitie kunnen worden afgeleid, de afleiding van een zin uit een structuur, en een ontleder die zinnen analyseert en er een structuur aan toekent. Deze abstracte benadering van syntax is ook van toepassing op de typesystemen van programmeertalen. Een signatuur beschrijft de geldige getypeerde expressies, uit getypeerde expressies kunnen ongetypeerde expressies worden afgeleid en een type checker analyseert ongetypeerde expressies en kent er een type aan toe. Op deze manier kunnen grammatica's beschouwd worden als signaturen voor algebraïsche specificaties.

Een algebra is een verzameling van gegevens met daarbij behorende operaties. Een algebraïsche specificatie is een beschrijving van een algebra. Bij het gebruik van grammatica's als signaturen zijn de programma's de gegevens van de algebra en hierop kunnen operaties gedefinieerd worden. Dit verband geeft een manier om ook de semantiek van een taal te beschrijven door middel van een operatie op de structuren van de taal. Zo'n operatie kan bijvoorbeeld een vertaling zijn naar een andere taal, of een interpretatie waarbij de waarde van een programma wordt uitgerekend zoals in het voorbeeld van expressies.

ASF+SDF is een algebraïsch specificatie formalisme waarin deze ideeën zijn uitgewerkt. ASF+SDF specificaties kunnen door een computer worden uitgevoerd, waardoor een taaldefinitie direct getest kan worden en daarmee getoetst aan de eisen die aan de taal gesteld worden.

C.2 Resultaten

In dit proefschrift worden methoden ontwikkeld waardoor de uitdrukkingskracht en bruikbaarheid van syntax definitie formalismen verbeterd worden. Het behandelt vier hoofonderwerpen: (1) Technieken voor ontleding en disambiguatie van context-vrije talen. (2) Ontwikkeling van een nieuw syntax definitie formalisme voor ontwerp van context-vrije talen. (3) Ontwerp van een meer-niveau algebraïsch specificatieformalisme. (4) Studie van polymorfe syntax definitie en ontleed problemen hiervan. Naast het introduceren van nieuwe technieken voor taalontwerp en ontwikkeling, bevat het proefschrift ook twee voorbeelden van de toepassing van technieken voor taalontwerp. De talen genoemd bij (2) en (3) zijn ontworpen met behulp van de algebraïsche specificatietaal ASF+SDF. De specificaties worden in dit proefschrift gebruikt om het ontwerp van de talen te presenteren. We bespreken deze onderwerpen en hun samenhang kort.

C.2.1 Ontleedtechnieken

Ontleden zonder Scanner Traditioneel bestaat een ontleder voor een programmeertaal uit twee onderdelen. Een scanner verdeelt de lijst van karakters waaruit een programma bestaat op in een lijst van lexicale tekens. Deze tekenlijst is vervolgens de invoer voor de eigenlijke ontleder die een boomstructuur aan het programma toekent. In Hoofdstuk 3 wordt een nieuwe ontleedmethode voorgesteld waarbij geen gebruik wordt gemaakt van een aparte scanner. De ontleder leest en analyseert direct de karakters van een programma.

De voordelen van ontleden zonder scanner ten opzichte van de traditionele methode zijn: (1) Er is geen implementatie van een scanner nodig, (2) Er kan met één formalisme voor de declaratie van de syntax volstaan worden. (3) Lexicale disambiguatie kan gebruik maken van de ontleedcontext. (4) De lexicale structuur blijft behouden. (5) De layout van de programmatekst blijft behouden in de boom. (6) Een expressievere lexicale syntax, bijvoorbeeld genest commentaar, wordt mogelijk.

De methode bestaat uit de volgende onderdelen: Normaliseren van de grammatica zorgt ervoor dat een expressief formalisme gebruikt kan worden terwijl de implementatie gebaseerd is op een eenvoudiger formalisme. Lexicale disambiguatie wordt uitgedrukt door middel van twee mechanismen. Zogenaamde 'volgbeperkingen' (follow restrictions) verbieden dat een constructie gevolgd kan worden door een bepaalde verzameling karakters. Hiermee kan uitgedrukt worden dat de langst mogelijke lezing van een lexicale categorie moet worden gekozen. Uitsluitingsregels (reject productions) beperken de verzameling bomen die voor een grammaticale categorie worden gegenereerd. Hiermee kan worden uitgedrukt dat gereserveerde woorden van de taal niet mogen worden gebruikt als variabelen. Het SLR(1) algoritme voor ontledergeneratie is aangepast om disambiguatie door middel van prioriteiten en volgbeperkingen uit te drukken. Gegeneraliseerd LR ontleden wordt gebruikt om beslissingsproblemen dynamisch op te lossen. Het GLR algoritme is aangepast om context-vrije grammatica's met uitsluitingsregels te behandelen.

Het blijkt dat deze uitsluitingsregels een grammaticaformalisme opleveren met een grotere uitdrukkingskracht dan context-vrije grammatica's. Dit wordt aangetoond door middel van een aantal voorbeelden. Het ontleedprobleem van het formalisme is echter wel beslisbaar.

Disambiguatiefilters Een disambiguatiefilter is een operatie die uit een verzameling ontleedbomen een deelverzameling selecteert. Dit biedt een zeer algemeen raamwerk waarbinnen disambiguatiemethoden bestudeerd kunnen worden. Het geeft een beschrijving van disambiguatie die onafhankelijk is van ontleedalgoritmen en geeft derhalve een meer inzichtelijke semantiek aan disambiguatiemethoden. Diverse eigenschappen van filters worden bestudeerd en een aantal disambiguatiemethoden wordt als filter gedefinieerd.

Optimalisatie van Ontlederschema's door Disambiguatiefilters Filters geven een zeer algemene methode om disambiguatie te beschrijven. Een filter kan geïmplementeerd worden door eerst een algemene ontleder te gebruiken die alle mogelijke bomen voor een ambigue zin oplevert en vervolgens deze verzameling te filteren, dat wil zeggen een deelverzameling van de mogelijke bomen te selecteren. Voor bepaalde toepassingen kan dit echter te duur zijn omdat het aantal mogelijke ontleedbomen exponentieel toeneemt met de lengte van de zin. Voorbeelden hiervan zijn ambigue binaire expressies en lexicale ambiguiteiten. Bij dit soort methoden willen we dan liever een implementatie waarbij disambiguatie in een zo vroeg mogelijk stadium plaatsvindt, indien mogelijk al bij de constructie van de ontleder. In Hoofdstuk 5 wordt onderzocht hoe uit de samenstelling van een ontleder en een disambiguatiefilter een geoptimaliseerde ontleder kan worden afgeleid. Dit wordt gedaan door een speciaal geval te onderzoeken, namelijk disambiguatie door middel van prioriteiten. Hiervoor wordt een algoritme voor ontledergeneratie afgeleid waardoor prioriteiten in de ontleedtabel verwerkt worden. Deze afleiding wordt gedaan door een abstracte beschrijving door middel van een ontleedschema van Earley's ontleedalgoritme te transformeren naar een ontleedschema dat rekening houdt met prioriteiten.

C.2.2 Een Familie van Syntax Definitie Formalismen

SDF is een syntax definitie formalisme voor de specificatie van lexicale en context-vrije syntax van programmeertalen. Het ontwerp van het formalisme is een monolithisch geheel waardoor het moeilijk is om het te implementeren en uit te breiden. In Deel II wordt SDF herontworpen als modulaire en uitbreidbare familie van syntax definitie formalismen. Iedere eigenschap van het formalisme wordt gespecificeerd als een orthogonale uitbreiding van het kernformalisme. De betekenis van veel eigenschappen wordt uitgedrukt door middel van een normalisatiefunctie die de constructies afbeeldt op constructies in het kernformalisme. Na vertaling blijven syntaxdefinities over die gebruik maken van context-vrije grammatica's met karakterklassen, prioriteiten, volgbeperkingen, en uitsluitingsregels. Eigenschappen die op deze manier behandeld worden, zijn: literals, reguliere expressies, lexicale en context-vrije syntax, hernoemingen, aliasen en modules. De samenstelling van deze eigenschappen vormt het syntax definitie formalisme SDF2.

C.2.3 Meer-Niveau Algebraïsche Specificaties

In ASF+SDF worden context-vrije grammatica's gebruikt om de structuur van data te definieren. Context-vrije grammatica's komen overeen met eerste-orde meersoortige algebraïsche signaturen. Hierbij heeft de typestructuur een beperkte vorm en kunnen alleen typeconstanten worden gedefinieerd. Dit heeft rigide structuren tot gevolg waarbij veel overeenkomsten tussen datastructuren niet tot uitdrukking komen. Veel aspecten van datastructuren zijn echter onafhankelijk van de data die ze opslaan. Beschouw bijvoorbeeld lijsten. Bij het bepalen van de lengte van een lijst is het onbelangrijk of de elementen van de lijst getallen zijn of waarheidswaarden. Door te abstraheren van de soort van data in een datatype kan de definitie van zo'n datatype makkelijker hergebruikt worden.

Om een flexibeler typestructuur te bereiken wordt in Deel III een studie gemaakt van meer-niveau algebraïsche specificaties. Hierbij zijn de types zelf definieerbaar als een algebraïsch datatype, die worden beschreven door middel van een algebraïsche signatuur. Bij twee-niveau specificaties zijn de types in de signatuur van niveau 0 termen over de signatuur van niveau 1. Bij meer-niveau specificaties wordt deze constructie veralgemeniseerd tot willekeurig veel niveaus, waarbij steeds geldt dat een type gebruikt op niveau n een term is over de signatuur op niveau $n + 1$.

Een groot deel van Deel III bestaat uit een formele specificatie in ASF+SDF van het meer-niveau algebraïsche specificatie formalisme MLS. De specificatie beschrijft de syntax, de semantiek en het typesysteem van het formalisme. De specificatie van het typesysteem is executeerbaar als termherschrijfsysteem en vormt een prototype programmeeromgeving voor het formalisme.

Er wordt een groot aantal voorbeelden van specificaties in MLS gegeven die laten zien hoe polymorfe datatypen gespecificeerd kunnen worden. In de eerste plaats worden een aantal standaard voorbeelden uit de wereld van de functionele programmeertalen besproken zoals lijsten, product en disjuncte vereniging. Daarna volgen voorbeelden die in het normale Hindley/Milner systeem niet uitdrukbaar zijn zoals de gestratificeerde stapel en tuples. Met behulp van typevergelijkingen kunnen geavanceerde typeconstructies gespecificeerd worden. Een voorbeeld is de typering van de functie `zip` die een tuple van lijsten aan elkaar ritst tot een lijst van tuples. Andere voorbeelden zijn typedefinities, tuple functor en tuple compositie, recursieve types en typeklassen. Deze voorbeelden laten zien dat meer-niveau algebraïsche specificaties een eenvoudig en uniform raamwerk vormen voor de specificatie van type structuren.

C.2.4 Polymorfe Syntax Definitie

In het MLS formalisme worden signaturen met prefix en infix functies gebruikt. Ook voor context-vrije grammatica's geldt dat ze rigide zijn en niet kunnen abstraheren van de concrete inhoud van een structuur. Het is wenselijk om de grammaticale regels voor een bepaalde structuur te kunnen hergebruiken door de soort van de elementen van de structuur in te vullen. De volgende regels geven bijvoorbeeld op een generieke manier aan hoe lijsten opgebouwd worden.

```
A      -> A+
A+ A+ -> A+
```

Hierbij is `A` de parameter van de regels. Door deze parameter op verschillende manieren in te vullen kan de structuur van lijsten hergebruikt worden. Ook operaties op lijsten kunnen dan op de generieke structuur gedefinieerd worden en hergebruikt voor iedere instantiatie.

Door het idee van grammatica's als signaturen te combineren met het idee van meer-niveau specificaties krijgen we een formalisme voor polymorfe syntax definitie. In een twee-niveau grammatica zijn de grammaticasymbolen van niveau 0 ontleedbomen over de grammatica op niveau 1. In Hoofdstuk 15 wordt dit idee uitgewerkt.

Een probleem van deze combinatie is dat, hoewel het ontleedprobleem van context-vrije grammatica's beslisbaar is en ook het typetoekenningsprobleem van meer-niveau specificaties beslisbaar is, het ontleedprobleem van twee-niveau grammatica's onbeslisbaar is. Hiervoor wordt een oplossing gegeven door de beperking tot twee-niveau grammatica's met eindige ketens. Voor deze klasse van grammatica's is het ontleedprobleem wel beslisbaar, terwijl de beperking de toepassing voor polymorfe syntax definitie niet schaadt.

D

Bibliography

Technical reports from the Programming Research Group of the University of Amsterdam can be obtained from `http://www.wins.uva.nl/research/prog/reports/reports.html` Technical reports from CWI can be found at `http://www.cwi.nl/cwi/publications/reports/reports.html`.

Aasa, A. (1991). Precedences in specifications and implementations of programming languages. In J. Maluzynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag.

Aasa, A. (1992). *User Defined Syntax*. Ph.D. thesis, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden.

Aho, A. V., Johnson, S. C., and Ullman, J. D. (1975). Deterministic parsing of ambiguous grammars. *Communications of the ACM*, **18**(8), 441–452.

Anderson, T., Eve, J., and Horning, J. (1973). Efficient LR(1) parsers. *Acta Informatica*, **2**(1), 12–39.

Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132, Paris. UNESCO.

Bahlke, R. and Snelting, G. (1986). The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, **8**(4), 547–576.

Bailes, P. A. and Chorvat, T. (1993). Facet grammars: Towards static semantic analysis by context-free parsing. *Computer Languages*, **18**(4), 251–271.

Bergstra, J. and Sellink, M. (1996). Sequential data algebra primitives. Technical Report P9602, University of Amsterdam, Programming Research Group.

Bergstra, J. A., Heering, J., and Klint, P., editors (1989a). *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley.

Bergstra, J. A., Heering, J., and Klint, P. (1989b). The algebraic specification formalism ASF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley. Chapter 1.

Bergstra, J. A., Heering, J., and Klint, P. (1990). Module algebra. *Journal of the ACM*, **37**(2), 335–372.

Bidoit, M., Gaudel, M.-C., and Mauboussin, A. (1989). How to make algebraic specifications more understandable: An experiment with the PLUSS specification language. *Science of Computer Programming*, **12**, 1–38.

Billot, S. and Lang, B. (1989). The structure of shared forests in ambiguous parsing. In *Proceedings of the Twenty-Seventh Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Bird, R. S. (1987). An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag.

Bird, R. S. (1989). Algebraic identities for program calculation. *The Computer Journal*, **32**(2), 122–126.

Blikle, A. (1989). Denotational engineering. *Science of Computer Programming*, **12**(3), 207–253.

Van den Brand, M. and Visser, E. (1994). From Box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam.

Van den Brand, M. G. J. (1992). *Pregmatic, A Generator for Incremental Programming Environments*. Ph.D. thesis, Katholieke Universiteit Nijmegen.

Van den Brand, M. G. J. and Visser, E. (1996). Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, **5**(1), 1–41.

Van den Brand, M. G. J., van Deursen, A., Dinesh, T. B., Kamperman, J. F. T., and Visser, E., editors (1995). *Proc. ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*. Technical Report P9504, Programming Research Group, University of Amsterdam.

Van den Brand, M. G. J., Klint, P., Olivier, P., and Visser, E. (1997a). ATerms: Representing structured data for exchange between heterogeneous tools. Technical report, Programming Research Group, University of Amsterdam.

Van den Brand, M. G. J., Sellink, M. P. A., and Verhoef, C. (1997b). Generation of components for software renovation factories from context-free grammars. Technical Report P9705, Programming Research Group, University of Amsterdam.

Van den Brand, M. G. J., Kuipers, T., Moonen, L., and Olivier, P. (1997c). Implementation of a prototype for the new ASF+SDF meta-environment. In A. Sellink, editor, *Second International Conference on the Theory and Practice of Algebraic Specification (ASF+SDF'97)*, Amsterdam, The Netherlands. Programming Research Group, University of Amsterdam.

Bratko, I. (1990). *Prolog. Programming for Artificial Intelligence*. Addison-Wesley, second edition.

Broy, M., Facchi, C., Grosu, R., Hettler, R., Hussmann, H., Nazareth, D., Regensburger, F., Slotosch, O., and Stølen, K. (1993). The requirement and design specification language SPECTRUM. An informal introduction. Version 1.0. Technical Report TUM-I9311 and TUM-I9312, Technische Universität München, München, Germany.

Cameron, R. D. (1993). Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, **2**(1-4), 85–94.

Cardelli, L. (1993). Typeful programming. SRC Research Report 45, May 24, 1989. Revised January 1, 1993, Digital Systems Research Center, Palo Alto, California.

Cardelli, L. (1997). Type systems. In J. B. Tucker, editor, *The Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, **17**(4), 471–522.

Cardelli, L., Matthes, F., and Abadi, M. (1994). Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, Palo Alto, California.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, **2**, 113–124.

Cordy, J. R. and Carmichael, I. H. (1993). *The TXL Programming Language. Syntax and Informal Semantics. Version 7*. Software Technology Laboratory, Department of Computer and Information Science, Queen's University at Kingston, Kinston, Canada, 7 edition.

Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM.

DeRemer, F. L. (1971). Simple LR(k) grammars. *Communications of the ACM*, **14**, 453–460.

Van Deursen, A. (1996). The static semantics of pascal. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*, chapter 2, pages 31–52. World Scientific, Singapore.

Van Deursen, A., Heering, J., and Klint, P., editors (1996). *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore.

Deussen, P. (1975). A decidability criterion for van Wijngaarden grammars. *Acta Informatica*, **5**, 353–375.

Dinesh, T. B. (1995). Injection misdemeanors. In M. G. J. v. d. Brand *et al.*, editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 255–270. Technical Report P9504, Programming Research Group, University of Amsterdam.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, **13**(2), 94–102.

Earley, J. (1975). Ambiguity and precedence in syntax description. *Acta Informatica*, **4**(1), 183–192.

Van Eijck, J. (1997). Email, july 9.

Floyd, R. W. (1962). Syntactic analysis and operator precedence. *Communications of the ACM*, **5**(10), 316–333.

Futatsugi, K., Goguen, J., Jouannaud, J.-P., and Meseguer, J. (1985). Principles of OBJ2. In B. Reid, editor, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM.

Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. (1977). Initial algebra semantics and continuous algebras. *Journal of the ACM*, **24**(1), 68–95.

Gordon, M., Milner, R., Morris, L., Newey, M., and Wadsworth, C. (1978). A meta language for interactive proof in LCF. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 119–130. ACM.

Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M., and Waite, W. M. (1992). Eli: A complete, flexible compiler construction system. *Communications of the ACM*, **35**, 121–131.

Groenink, A. V. (1997). *Surface without Structure*. Ph.D. thesis, University of Utrecht.

Grosch, J. (1990). Lalr - a generator for efficient parsers. *Software-Practice & Experience*, **20**, 1115–1135.

Hatcher, W. S. and Rus, T. (1976). Context-free algebras. *Journal of Cybernetics*, **6**, 65–76.

Hearn, B. M. (1995). *The Design and Implementation of Typed Languages for Algebraic Specification*. Ph.D. thesis, University of Wales, Swansea.

Hearn, B. M. and Meinke, K. (1994). ATLAS: A typed language for algebraic specification. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Proceedings of the First International Workshop on Higher-Order Algebra, Logic and Term Rewriting (HOA '93)*, volume 816 of *lecture Notes in Computer Science*, pages 146–168, Berlin. Springer-Verlag.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, **24**(11), 43–75.

Hendriks, P. R. H. (1989). Typechecking Mini-ML. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 299–337. The ACM Press in co-operation with Addison-Wesley. Chapter 7.

Hendriks, P. R. H. (1991). *Implementation of Modular Algebraic Specifications*. Ph.D. thesis, University of Amsterdam.

Higgins, P. J. (1963). Algebras with a scheme of operators. *Mathematische Nachrichten*, **27**, 115–132.

Hillebrand, J. and Korver, H. (1995). A well-formedness checker for μCRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes (ACP '95)*, pages 81–119. Eindhoven University of Technology, Computing Science Report 95/14.

Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions American Mathematical Society*, **146**, 29–60.

Hudak, P., Peyton Jones, S., and Wadler, P., editors (1992). *Report on the Programming Language Haskell. A Non-strict, Purely Functional Language. (Version 1.2)*, volume 27. ACM SIGPLAN Notices.

Jansson, P. and Jeuring, J. (1997). Polyp - a polytypic programming language extension. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France. ACM.

Jensen, K. and Wirth, N. (1975). *PASCAL User Manual and Report*, volume 18 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, second edition edition.

Johnson, S. C. (1975). YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J.

Jones, M. P. (1992). A theory of qualified types. In B. Krieg-Brückner, editor, *4th European Symposium on Programming (ESOP'92)*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag.

Jones, M. P. (1993). A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark. (Appears in extended form in the Journal of Functional Programming, 1995.).

Jones, M. P. (1995). A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, **5**, 1–35.

Jouannaud, J. P. and Kirchner, C. (1991). Solving equations in abstract algebras: A rule-based survey of unification. In J. L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in Honour of Alan Robinson*, chapter 8, pages 257–321. M.I.T. Press, Cambridge (MA).

Jouannaud, J.-P. and Lescanne, P. (1982). On multiset orderings. *Information Processing Letters*, **15**(2), 57–63.

Kahn, G., Lang, B., Mélèse, B., and Morcos, E. (1983). METAL: A formalism to specify formalisms. *Science of Computer Programming*, **3**, 151–188.

Kamperman, J. (1992). A try at improving the second disambiguation phase in SDF. technical note.

Kamperman, J. F. T. (1994). GEL, a graph exchange language. Technical Report CS-R9440, CWI, Amsterdam.

Kinnersley, B. (1995). The language list – version 2.4. Collected information on about 2350 computer languages, past and present. Available from http://wuarchive.wustl.edu/doc/misc/lang-list.txt.

Klint, P. (1988). Definitie van prioriteiten in SDF. Unpublished technical note (in dutch).

Klint, P. (1993). A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, **2**(2), 176–201.

Klint, P., editor (1995). *The ASF+SDF Meta-environment User's Guide*. Centrum voor Wiskunde en Informatica (CWI), Amsterdam. Version June 4. Available as ftp://ftp.cwi.nl/pub/gipe/reports/SysManual.ps.Z.

Klint, P. and Visser, E. (1994). Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

Knuth, D. E. (1965). On the translation of languages from left to right. *Information and Control*, **8**, 607–639.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, **2**(2), 127–145. Correction in: Mathematical Systems Theory 5(1), pp. 95-96, Springer-Verlag, 1971.

Koster, C. H. A. (1971). Affix grammars. In J. E. L. Peck, editor, *Algol-68 Implementation*. North-Holland, Amsterdam.

LaLonde, W. R. and Rivieres, J. d. (1981). Handling operator precedence in arithmetic expressions with tree transformations. *ACM Transactions on Programming Languages and Systems*, **3**(1), 83–103.

Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, **9**(3), 157–166.

Lang, B. (1974). Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag.

Lee, J. A. N. (1972). The formal definition of the BASIC language. *Computer Journal*, **15**(1), 37–41.

Lesk, M. E. and Schmidt, E. (1986). *LEX — A lexical analyzer generator*. Bell Laboratories. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

Loeckx, J., Ehrich, H.-D., and Wolf, M. (1996). *Specification of abstract data types*. J. Wiley & Sons and B.G.Teubner Publishers.

Luttik, B. and Visser, E. (1997). Specification of rewriting strategies. In A. Sellink, editor, *Second International Conference on the Theory and Practice of Algebraic Specification (ASF+SDF'97)*, Amsterdam, The Netherlands. Programming Research Group, University of Amsterdam.

Maluszynski, J. (1984). Towards a programming language based on the notion of two-level grammar. *Theoretical Computer Science*, **28**, 13–43.

Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, **4**, 258–282.

McCrosky, C. and Sailor, K. (1993). A synthesis of type-checking and parsing. *Computer Languages*, **18**(4), 241–250.

Meinke, K. (1992a). Equational specification of abstract types and combinators. In E. Boerger, G. Jaeger, H. K. Buening, and M. M. Richter, editors, *Computer Science Logic - CSL'91*, volume 626 of *Lecture Notes in Computer Science*, pages 257–271, Berlin. Springer-Verlag.

Meinke, K. (1992b). Universal algebra in higher types. *Theoretical Computer Science*, **100**, 385–417.

Meinke, K. (1993). Algebraic semantics of rewriting terms and types. In J. Remy and M. Rusinowitch, editors, *Proc. Third Int. Workshop on Conditional Term Rewriting Systems*, volume 656 of *Lecture Notes in Computer Science*, pages 1–20, Berlin. Springer-Verlag.

Meinke, K. and Tucker, J. V. (1992). Universal algebra. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science. Volume I: Mathematical Structures*, pages 189–141. Oxford University Press, Oxford.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**(3), 348–375.

Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts.

Mitchell, J. (1990). Type theories in programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 366–458. Elsevier Science Publishers.

Möller, B. (1987). Algebraic specification with higher-order operators. In L. Meertens, editor, *Program Specification and Transformation*, pages 367–398. Elsevier Science Publishers B.V. (North-Holland).

Mosses, P. D. (1992). *Action Semantics*. Cambridge University Press.

Mosses, P. D. (1993). The use of sorts in algebraic specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification (WADT '91)*, volume 655 of *Lecture Notes in Computer Science*, pages 66–92. Springer-Verlag.

Naur, P. *et al.* (1960). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, **3**(5), 299–314.

Nazareth, D. (1995). *A Polymorphic Sort System for Axiomatic Specification Languages*. Ph.D. thesis, Technische Universität München. Technical Report TUM-I9515.

Nipkow, T. and Prehofer, C. (1995). Type reconstruction for type classes. *Journal of Functional Programming*, **5**(2), 201–224.

Oude Luttighuis, P. and Sikkel, K. (1992). Attribute evaluation during generalized parsing. Memoranda Informatica 92-85, Universiteit Twente, Faculteit der Informatica.

Parr, T. J. and Quong, R. W. (1994). Adding semantic and syntactic predicates to LL(k): pred-LL(k). In P. A. Fritzson, editor, *Compiler Construction, 5th International Conference, CC'94*, volume 786 of *LNCS*, pages 263–277, Edinburgh, U.K. Springer-Verlag.

Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, **13**, 231–278.

Pereira, F. C. N. and Warren, D. H. D. (1983). Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Poigné, A. (1986). On specifications, theories, and models with higher types. *Information and Control*, **68**, 1–46.

Rekers, J. (1992). *Parser Generation for Interactive Environments*. Ph.D. thesis, University of Amsterdam. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

Reps, T. and Teitelbaum, T. (1989). *The Synthesizer Generator: a System for Constructing Language-Based Editors*. Springer-Verlag.

Richards, H. (1984). An overview of ARC SASL. *SIGPLAN Notices*, **19**(10).

Rus, T. (1972). ΣS-Algebra of a formal language. *Bulletin Mathematique de la Societe de Science, Bucharest*, **15**(63-2), 227–235.

Rus, T. and Jones, J. S. (1995). Multi-layered pipeline parsing from multi-axiom grammars. In A. Nijholt, G. Scollo, and R. Steetskamp, editors, *Algebraic Methods in Language Processing AMiLP'95*, volume 10 of *Twente Workshops in Language Technology*, pages 65–81, Enschede, The Netherlands. Twente University of Technology.

Salomon, D. J. and Cormack, G. V. (1989). Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, **24**(7), 170–178.

Salomon, D. J. and Cormack, G. V. (1995). The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canada.

Sellink, A., editor (1997). *Proceedings of the Second International Workshop on Theory and Practice of Algebraic Specification (ASF+SDF'97)*. Programming Research Group, University of Amsterdam.

Share, M. (1988). Resolving ambiguities in the parsing of translation grammars. *ACM SIGPLAN Notices*, **23**, 103–109.

Shieber, S. M., Schabes, Y., and Pereira, F. C. N. (1995). Principles and implementation of deductive parsing. *The Journal of Logic Programming*, pages 3–36.

Sikkel, K. (1993). *Parsing Schemata.* Ph.D. thesis, Universiteit Twente, Enschede.

Sikkel, K. (1994). How to compare the structure of parsing algorithms. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 21–39, Milano, Italy. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

Sikkel, K. (1997). *Parsing Schemata. A Framework for Specification and Analysis of Parsing Algorithms*, volume XVI of *Texts in Theoretical Computer Science. An EATCS Series*. Springer-Verlag, Berlin / Heidelberg / New York.

Sintzoff, M. (1967). Existence of a van Wijngaarden syntax for every recursively enumerable set. *Annales de la Société Scientifique de Bruxelles*, **81**(II), 115–118.

Tanaka, H., Tokunga, T., and Aizawa, M. (1996). Integration of morphological and syntactical analysis based on GLR parsing. In H. C. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technology*, pages 325–342. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Thorup, M. (1992). Ambiguity for incremental parsing and evaluation. Technical Report PRG-TR-24-92, Program Research Group, Oxford University, Oxford, U.K.

Thorup, M. (1994a). Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, **16**(3), 1024–1050.

Thorup, M. (1994b). Disambiguating grammars by exclusion of sub-parse trees. Technical Report 94/11, Dept. of Computer Science, University of Copenhagen, Denmark.

Tomita, M. (1985). *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

Turner, D. A. (1985). Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France. Springer-Verlag.

Veldhuijzen van Zanten, G. E. (1988a). *An attributed-LALR-parser generator for syntactically ambiguous grammars*. Master's thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands.

Veldhuijzen van Zanten, G. E. (1988b). SABLE: A parser generator for ambiguous grammars. Memoranda Informatica INF 88-62, Department of Computer Science, University of Twente, Enschede, The Netherlands.

Vigna, S. (1995). Specifying IMP(G) using ASF+SDF: A case study. In M. G. J. van den Brand, A. v. Deursen, T. B. Dinesh, J. F. T. Kamperman, and E. Visser, editors, *Proc. ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 65–88. Technical Report P9504, Programming Research Group, University of Amsterdam.

Vigna, S. (1996). *Distributive Computability*. Ph.D. thesis, Università degli Studi di Milano e di Torino.

Visser, E. (1993). *Combinatory Algebraic Specification & Compilation of List Matching*. Master's thesis, Department of Computer Science, University of Amsterdam, Amsterdam.

Visser, E. (1994a). *ASF+SDF to LaTeX. User Manual*. Programming Research Group, University of Amsterdam.

Visser, E. (1994b). Writing course notes with ASF+SDF to LaTeX. In T. B. Dinesh and S. M. Üsküdarlı, editors, *Using the ASF+SDF environment for teaching computer science*, chapter 6. Collection submitted to NSF workshop on teaching formal methods.

Visser, E. (1995a). A case study in optimizing parsing schemata by disambiguation filters. In S. Fischer and M. Trautwein, editors, *Proceedings Accolade95*, pages 153–167, Amsterdam. The Dutch Graduate School in Logic.

Visser, E. (1995b). A family of syntax definition formalisms. In M. G. J. van den Brand *et al.*, editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam.

Visser, E. (1995c). Polymorphic syntax definition (extended abstract). In A. Nijholt, G. Scollo, and R. Steetskamp, editors, *Algebraic Methods in Language Processing AMiLP'95*, volume 10 of *Twente Workshops in Language Technology*, pages 43–54, Enschede, The Netherlands. Twente University of Technology.

Visser, E. (1996a). Multi-level specifications. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*, pages 105–196. World Scientific, Singapore.

Visser, E. (1996b). Solving type equations in multi-level specifications (preliminary version). Technical Report P9606, Programming Research Group, University of Amsterdam, Amsterdam.

Visser, E. (1997a). A case study in optimizing parsing schemata by disambiguation filters. In A. Nijholt, editor, *International Workshop on Parsing Technology IWPT'97*, Boston, USA. (To appear).

Visser, E. (1997b). Character classes. Technical Report P9708, Programming Research Group, University of Amsterdam.

Visser, E. (1997c). Executable specification of programming languages. (Draft).

Visser, E. (1997d). A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam.

Visser, E. (1997e). Polymorphic syntax definition. *Theoretical Computer Science*. (To appear).

Visser, E. (1997f). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

Vittek, M. (1994). *ELAN: Un cadre logique pour le prototypage de language de programmation avec contraintes*. Ph.D. thesis, Université Henri Poincaré – Nancy I, Nancy, France.

Voisin, F. (1986). CIGALE: a tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, **7**, 61–86.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *16'th Symposium on Principles of Programming Languages (POPL'89)*, pages 60–67, Austin, Texas. ACM Press.

Wagner, T. A. and Graham, S. L. (1997). Incremental analysis for real programming languages. *SIGPLAN Notices*, **32**(5), 31–43. Proc. of the 1997 ACM SIGPLAN Conferene on Programming Language Design and Implementation (PLDI).

Walters, H. and Kamperman, J. (1996). Epic 1.0 (unconditional), an equational programming language. Technical Report CS-R9604, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

Warren, D. S. (1992). Memoing for Logic Programs. *Communications of the ACM*, **35**(3), 94–111.

Watt, D. A. (1977). The parsing problem for affix grammars. *Acta Informatica*, **8**(1), 1–20.

Wechler, W. (1992). *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag.

Wharton, R. (1976). Resolution of ambiguity in parsing. *Acta Informatica*, **6**(4), 387–395.

van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., and Fisker, R. G., editors (1976). *Revised Report on the Algorithmic Language Algol 68*. Springer-Verlag, Berlin Heidelberg New York.

Williams, M. H. (1982). A flexible notation for syntactic definitions. *ACM Transactions on Programming Languages and Systems*, **4**(1), 113–119.

Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM*, **20**(11), 822–823.