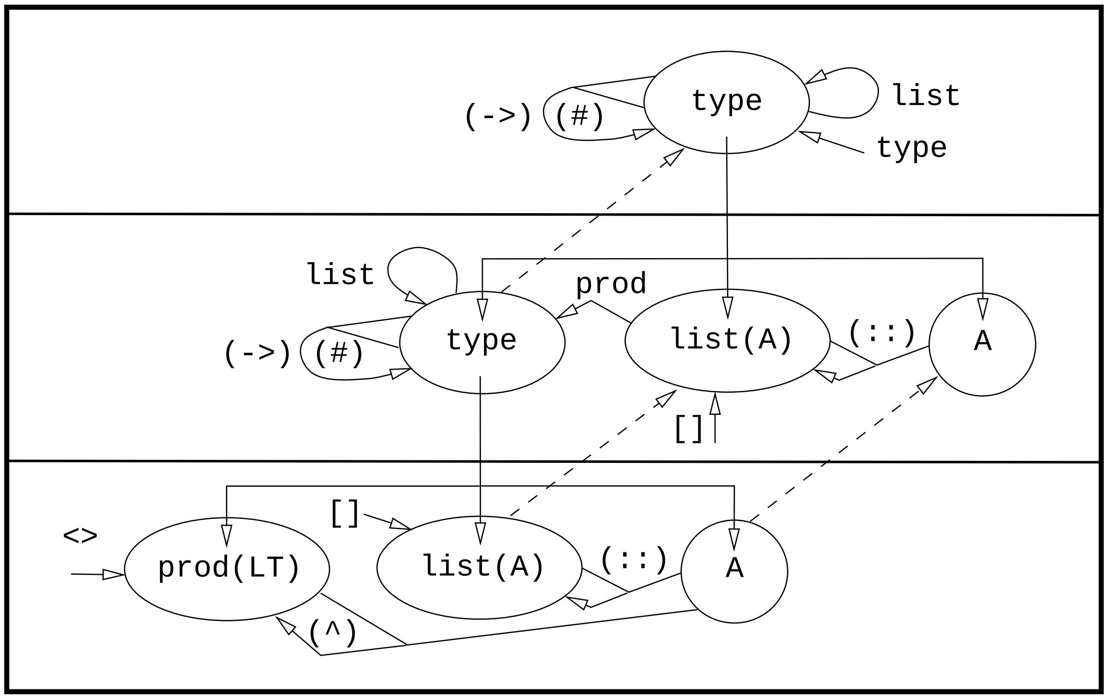
University of Amsterdam

Programming Research Group



Multi-Level Specifications

E. Visser

Report P9604

April 1996

University of Amsterdam

Department of Logic and Computer Science

Programming Research Group

Multi-level specifications

E. Visser

Report P9604 April 1996

E. Visser

Programming Research Group
Department of Logic and Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7590
e-mail: visser@fwi.uva.nl

This technical report is a preprint of Chapter 5 of A. van Deursen, J. Heering and P. Klint (editors) *Language Prototyping. An Algebraic Specification Approach.* AMAST Series in Computing, World Scientific Publishing Inc., 1996

Acknowledgements Discussions with and/or comments on drafts of this chapter by Jan Bergstra, Arie van Deursen, Dinesh, Pieter Hartel, Jan Heering, Paul Klint, Karl Meinke, Jon Mountjoy and Teo Rus considerably improved the formalism, the specification, the presentation and my spirit. Arie van Deursen is acknowledged for asking me to write this chapter. Leon Moonen created a tool for automatically constructing the import graph for ASF+SDF specifications, which was of great help in restructuring the specification.

The research was conducted with support from the Dutch Organization for Scientific Research (NWO) under grant 612-317-420: Incremental parser generation and context-dependent disambiguation, a multi-disciplinary perspective.

Contents

Multi-Level Specifications

Eelco Visser

Abstract This chapter introduces 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 in level n are defined in 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 this chapter 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, this chapter illustrates the use of ASF+SDF for the design and prototyping of sophisticated specification formalisms.

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.

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 ($-0 = 0$).

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 $s_1 \times \cdots \times s_n \to s_0$ 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.

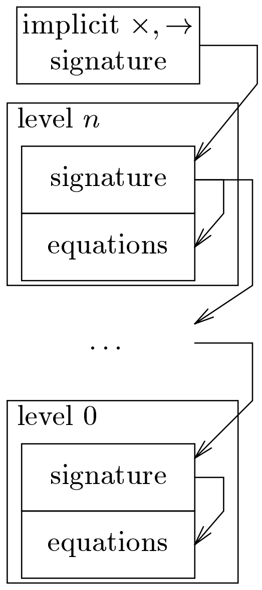
Types of Types: The language of types built from type constants and type constructors is itself an algebraic language with a signature. In many-sorted algebraic signatures the only type constructors are $\times$ and $\rightarrow$ and the language of types is restricted to types of the form $c_1 \times \cdots \times c_n \rightarrow c_0$, 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 `type` $\times \cdots \times$ `type` $\rightarrow$ `type`. For instance, `list` is a type constructor that takes a type and constructs a type, i.e., it is declared as `list : type` $\rightarrow$ `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 Section 8 for some references to surveys of type systems.

1.3 Multi-Level Specifications

In this chapter we present the formalism MLS, a *modular, applicative, multi-level, equational specification formalism with overloading.* Figure 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 the diagram next to this paragraph. 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

```
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; list(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
    sorts A; tree(A);
    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 1: Two-level specification of list and tree data types.

of the terms declared as sort. These ideas are illustrated in Figure 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 `#` 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 declarations `A` and `list(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 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 Section 5 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 $\tau_1 \rightarrow \tau_2$ 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[1] over terms express the semantics of terms. Equational logic can be used for reasoning about terms, whereas term rewriting can be used to decide equations for appropriate systems of equations or to compute the result

[1]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.

of defined functions.

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 Section 5 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 in this chapter does not take into account equations over types. This 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.

1.4 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 1.

One-Level Monomorphic Algebraic Languages The algebraic specification formalisms OBJ (Futatsugi *et al.*, 1985), Pluss (Bidoit *et al.*, 1989) and ASF+SDF (see Chapter 1 of van Deursen *et al.* (1996)) have monomorphic many-sorted first-order signatures as type system. The sort space consists of terms of the form $c_1 \times \cdots \times c_n \to c_0$, 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 in this chapter support neither subsorting nor syntaxless functions.

One-Level Monomorphic Applicative Languages The one-level applicative specification language OLS, defined in Sections 3 and 4, generalizes the sort space of monomorphic algebraic languages to the closure under $\times$ and $\to$ 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

add. features	algebraic	# levels	applicative	add. 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 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.

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 Section 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 distinction 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 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 $id : \forall\alpha.\alpha \rightarrow \alpha$, 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 $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, 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.

1.5 Outline

The rest of this chapter presents 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 Section 2 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 Section 3 this untyped language is extended to a one-level language, after introducing the notions of types and signatures.

In Section 4 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 Section 5 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 MLS.

In Section 7 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.

The appendices of this chapter define a number of tools that are used in the specification. In Appendix A several 'standard' library modules like Layout and Booleans are defined. In Appendix B several utilities on terms such as sets of terms, substitution, matching and unification are defined.

2 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.

2.1 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.

```
imports Layout^A.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, Appendix B.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 1 the expression `size(T) + size(T')` is equivalent to `(+)(size(T), size(T'))` and `X :: L` is equivalent to `(::)(X, L)`. Furthermore, Appendix B.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 `<`$t_1, \ldots, t_n$`>` 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 that are out of the scope of this chapter, and the subject of ongoing research (see (Visser, 1995b) and Section 8).

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

[l-conc] $t^*_1 + t^*_2 = t^*_1; t^*_2$

2.2 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.

imports Binary-Operators$^{B.1}$ Terms$^{2.1}$

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

```
0 + X           == X;
s(X) + Y        == s(X + Y);
map(G)([])      == [];
map(G)(X :: L) == G(X) :: map(G)(L)
```

Figure 2: Untyped equational specification of addition on successor naturals and map over cons lists.

equations

[eqs-conc] $\varphi_1^* +\!\!+\ \varphi_2^* = \varphi_1^*; \varphi_2^*$

An example specification is shown in Figure 2. 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.

2.3 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 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 [el-sub] uses the notation $\sigma(t)$ for the application to a term t of a substitution σ that maps variables to terms. (See Appendix B.7 for the definition of substitution.)

imports Equations$^{2.2}$ Substitution$^{B.7}$ Booleans$^{A.2}$
exports
  context-free syntax
    Eqs "$\vdash$" Eq $\rightarrow$ Bool

equations

[el-refl] $$\mathcal{E} \vdash t \equiv t \; = \; \top$$

[el-sym] $$\frac{\mathcal{E} \vdash t_2 \equiv t_1 = \top}{\mathcal{E} \vdash t_1 \equiv t_2 \; = \; \top}$$

[el-trans] $$\frac{\mathcal{E} \vdash t_1 \equiv t_2 = \top, \; \mathcal{E} \vdash t_2 \equiv t_3 = \top}{\mathcal{E} \vdash t_1 \equiv t_3 \; = \; \top}$$

[el-ax] $$\varphi_1^*; \, t_1 \equiv t_2; \, \varphi_2^* \vdash t_1 \equiv t_2 \; = \; \top$$

[el-sub] $$\frac{\mathcal{E} \vdash t_1 \equiv t_2 = \top}{\mathcal{E} \vdash \sigma(t_1) \equiv \sigma(t_2) \; = \; \top}$$

[el-app] $$\frac{\mathcal{E} \vdash t_1 \equiv t_3 = \top, \; \mathcal{E} \vdash t_2 \equiv t_4 = \top}{\mathcal{E} \vdash t_1 \; t_2 \equiv t_3 \; t_4 \; = \; \top}$$

[el-pr] $$\frac{\mathcal{E} \vdash t_1 \equiv t_3 = \top, \; \mathcal{E} \vdash t_2 \equiv t_4 = \top}{\mathcal{E} \vdash t_1, t_2 \equiv t_3, t_4 \; = \; \top}$$

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.

2.4 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 lhs 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 rhs is evaluated. In equation [redex] the list of equations is searched (by means of list matching, see Section 1.4.2 of van Deursen *et al.* (1996)) for an equation $t_1 \equiv t_2$ such that the lhs 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 rhs of the equation. If no matching equation is found, 'step' just returns its argument (equation [nf]). 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.[2]

imports Matching$^{B.8}$ Equations$^{2.2}$
exports
  context-free syntax

eval "(" Eqs ")" "⟦" Term "⟧"	→ Term
eval "(" Eqs ")" "⟦" Term "⟧" "_" Subst	→ Term
step "(" Eqs ")" "⟦" Term "⟧"	→ Term

equations

[eval-trm] $\text{eval}(\mathcal{E})[\![t]\!] = \text{eval}(\mathcal{E})[\![t]\!]_{[]}$

[eval-var] $\text{eval}(\mathcal{E})[\![x]\!]_\sigma = \sigma(x)$

[eval-fun] $\text{eval}(\mathcal{E})[\![f]\!]_\sigma = \text{step}(\mathcal{E})[\![f]\!]$

[eval-app] $\text{eval}(\mathcal{E})[\![t_1\ t_2]\!]_\sigma = \text{step}(\mathcal{E})[\![\text{eval}(\mathcal{E})[\![t_1]\!]_\sigma\ \text{eval}(\mathcal{E})[\![t_2]\!]_\sigma]\!]$

[eval-pr] $\text{eval}(\mathcal{E})[\![t_1, t_2]\!]_\sigma = \text{step}(\mathcal{E})[\![\text{eval}(\mathcal{E})[\![t_1]\!]_\sigma, \text{eval}(\mathcal{E})[\![t_2]\!]_\sigma]\!]$

[redex] $\text{step}(\mathcal{E})[\![t]\!] = \text{eval}(\mathcal{E})[\![t_2]\!]_\sigma$
  when $\mathcal{E} = \varphi_1^*; t_1 \equiv t_2; \varphi_2^*,\ t_1 := t = \sigma$

[nf] $\text{step}(\mathcal{E})[\![t]\!] = t$ **otherwise**

The following proposition states that evaluation is sound with respect to derivable equality.

Proposition 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

[2] 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.

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.

3 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 2 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.

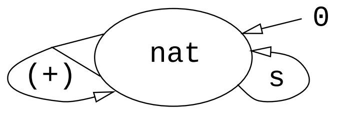
3.1 An Example

Before giving the syntax of type terms, signatures and specifications we discuss a simple example of a one-level specification. Figure 3(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 3(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 `(+)`

```
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 3: Specification of successor naturals with addition (a) and corresponding signature diagram (b).

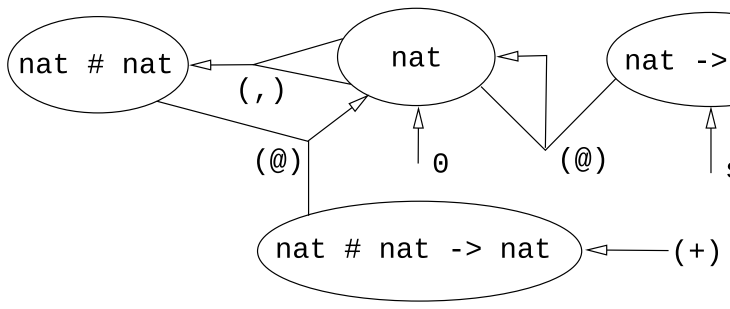


Figure 4: Signature diagram of natural numbers in which function and product types and the corresponding application and pairing functions are depicted explicitly. The functions `s` and `(+)` are constants of functional types.

takes two natural numbers and produces a new one, which is depicted by the forked arrow.

As we will see, the term structure of one-level specifications is actually applicative. This entails that besides `nat`, there are two sorts `nat -> nat` and `nat # nat -> nat`, i.e., they are sets of terms. The signature diagram in Figure 4 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.

3.2 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 Sections 5 and 7 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.

imports Terms$^{2.1}$
exports
  context-free syntax
    "nil" $\rightarrow$ Term
    "top" $\rightarrow$ Term
    Term "$\times$" Term $\rightarrow$ Term {**right**}
    Term "$\rightarrow$" Term $\rightarrow$ Term {**right**}
    Term ":" Term $\rightarrow$ Term {**right**}
  priorities
    Term Term $\rightarrow$ Term $>$ Term "$\times$"Term $\rightarrow$ Term $>$ Term "$\rightarrow$"Term $\rightarrow$ Term
    $>$ Term ":"Term $\rightarrow$ Term $>$ Term ","Term $\rightarrow$ Term
  variables
    "τ"[*0-9'*]$*$ $\rightarrow$ Term
    "τ" "$*$"[*0-9'*]$*$ $\rightarrow$ {Term ";"}$*$
    "τ" "+"[*0-9'*]$*$ $\rightarrow$ {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, read

```
list : type -> type           as  list : (type -> type)
list A -> nat                 as  (list A) -> nat
nat # nat -> nat              as  (nat # nat) -> nat
list : type -> type A : type  as  list : ((type -> (type A)) : type)
                       and not as  (list : (type -> type)) (A : type)
```

3.3 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 TermToTerm is the sort of functions from terms to terms that is defined in Appendix B.3. The basic operation of this sort is the application of a function to a term yielding a term, i.e., TermToTerm(Term) → Term. This approach makes it possible to generically define a function that applies a TermToTerm function to all terms in a list of terms.

imports Term-Functions$^{B.3}$ Terms$^{2.1}$ Types$^{3.2}$ Binary-Operators$^{B.1}$
exports
  context-free syntax
    spine → TermToTerm
    fspine → TermToTerm
    term → TermToTerm
    type → TermToTerm
    dom → TermToTerm
    cod → TermToTerm
    fun → TermToTerm
    arg → TermToTerm
    bterm → TermToTerm
    bapp → TermToTerm
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)`.

[sp-ann] $\mathrm{spine}(t : \tau) = \mathrm{spine}(t)$
[sp-fun] $\mathrm{spine}(f) = f$
[sp-var] $\mathrm{spine}(x) = x$
[sp-top] $\mathrm{spine}(\mathrm{nil}) = \mathrm{nil}$
[sp-top] $\mathrm{spine}(\mathrm{top}) = \mathrm{top}$
[sp-pr] $\mathrm{spine}(t_1, t_2) = \mathrm{spine}(t_1), \mathrm{spine}(t_2)$
[sp-app] $\mathrm{spine}(t_1\ t_2) = \mathrm{spine}(t_1)\ \mathrm{spine}(t_2)$
[sp-prd] $\mathrm{spine}(t_1 \times t_2) = \mathrm{spine}(t_1) \times \mathrm{spine}(t_2)$
[sp-arr] $\mathrm{spine}(t_1 \to t_2) = \mathrm{spine}(t_1) \to \mathrm{spine}(t_2)$

The function 'fspine' is the same as 'spine' except that it does not remove the annotation from a function symbol.

[fsp-fun] $\mathrm{fspine}(f : \tau) = f : \tau$
[fsp-ann] $\mathrm{fspine}(t : \tau) = \mathrm{fspine}(t)$ **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.

[trm-ann] $\mathrm{term}(t : \tau) = t$
[tp-ann] $\mathrm{type}(t : \tau) = \tau$

We see that for any term t of the form $t' : \tau$, $\mathrm{term}(t) : \mathrm{type}(t) = t$. 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.

[trm] $\mathrm{term}(t) = t$ **otherwise**
[type] $\mathrm{type}(t) = \mathrm{top}$ **otherwise**
[top-ann] $t : \mathrm{top} = t$

Now we have for arbitrary terms

[term-type] $\mathrm{term}(t) : \mathrm{type}(t) = t$

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

[dom-arr] $\mathrm{dom}(t_1 \to t_2) = t_1$ [arg-app] $\mathrm{arg}(t_1\ t_2) = t_2$

[dom]	$\text{dom}(t)$	$=$	nil	**otherwise**	[arg]	$\text{arg}(t)$	$=$	nil	**otherwise**
[cod-arr]	$\text{cod}(t_1 \to t_2)$	$=$	t_2		[fun-app]	$\text{fun}(t_1\ t_2)$	$=$	t_1	
[cod]	$\text{cod}(t)$	$=$	t	**otherwise**	[fun]	$\text{fun}(t)$	$=$	t	**otherwise**
[nil-cod]	$\text{nil} \to t$	$=$	t		[fun-nil]	$t\ \text{nil}$	$=$	t	

We have

[arg-res] $\quad \text{dom}(t) \to \text{cod}(t) \ = \ t \qquad$ [arg-res] $\quad \text{fun}(t)\ \text{arg}(t) \ = \ t$

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.

[bterm1] $\quad \text{bterm}(t) = \text{term}(\text{dom}(\text{term}(t))) \to \text{cod}(\text{term}(t))$

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

[bapp1] $\quad \text{bapp}(t) = t_1\ (t_2, t_3)$
$\qquad$ **when** $\text{term}(t) = t_1'\ t_2',\ \text{term}(t_1') = t_1,\ \text{term}(t_2') = t_2, t_3$
[bapp2] $\quad \text{bapp}(t) = t$ **otherwise**

For example,

```
bapp(((+) : nat # nat -> nat)((0 : nat, 0 : nat) : nat # nat) : nat)
= (+)((0 : nat), (0 : nat))
```

3.4 Syntax of One-Level Specifications (OLS)

A signature is constructed from sort, function and variable 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.

imports Terms$^{2.1}$ Types$^{3.2}$ Binary-Operators$^{B.1}$

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.

[f-decl] $$d_1^*; f_1^+, f_2^+ : \tau; d_2^* = d_1^*; f_1^+ : \tau; f_2^+ : \tau; d_2^*$$
[v-decl] $$d_1^*; x_1^+, x_2^+ : \tau; d_2^* = d_1^*; x_1^+ : \tau; x_2^+ : \tau; d_2^*$$
[decls-conc] $$d_1^* \mathbin{+\!\!+} d_2^* = d_1^*; d_2^*$$

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 → Sig
    “functions” Decls → Sig
    “variables” Decls → Sig
    → Sig
    Sig “;” Sig → Sig {**right**}
    “(” Sig “)” → Sig {**bracket**}
    “S” (Sig) → Terms
    “F” (Sig) → Decls
    “V” (Sig) → Decls
  variables

"Σ"[*0-9′*]∗ → Sig

equations

Equations [Sig-es], [Sig-ef] and [Sig-ev] express that atomic signatures with empty declaration lists are equivalent to empty signatures.

[Sgel]	$;\Sigma$	=	Σ	[Sges]	sorts	=	
[Sger]	$\Sigma;$	=	Σ	[Sgef]	functions	=	
[Sgas]	$(\Sigma_1;\Sigma_2);\Sigma_3$	=	$\Sigma_1;\Sigma_2;\Sigma_3$	[Sgev]	variables	=	
[S1]	S(sorts *ts*)	=	*ts*	[S4]	S()	=	
[S2]	S(functions *ds*)	=		[S5]	$\mathrm{S}(\Sigma_1;\Sigma_2)$	=	$\mathrm{S}(\Sigma_1) +\!\!+ \mathrm{S}(\Sigma_2)$
[S3]	S(variables *ds*)	=					
[F1]	F(sorts *ts*)	=		[F4]	F()	=	
[F2]	F(functions *ds*)	=	*ds*	[F5]	$\mathrm{F}(\Sigma_1;\Sigma_2)$	=	$\mathrm{F}(\Sigma_1) +\!\!+ \mathrm{F}(\Sigma_2)$
[F3]	F(variables *ds*)	=					
[V1]	V(sorts *ts*)	=		[V4]	V()	=	
[V2]	V(functions *ds*)	=		[V5]	$\mathrm{V}(\Sigma_1;\Sigma_2)$	=	$\mathrm{V}(\Sigma_1) +\!\!+ \mathrm{V}(\Sigma_2)$
[V3]	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$^{2.2}$
exports
  sorts Spec
  context-free syntax

"signature" Sig	→	Spec	
"equations" Eqs	→	Spec	
	→	Spec	
Spec ";" Spec	→	Spec	{**right**}
"(" Spec ")"	→	Spec	{**bracket**}
"Sg"(Spec)	→	Sig	
"E"(Spec)	→	Eqs	

  variables
    "$\mathcal{S}$"[*0-9′*]∗ → Spec

equations

[Spel]	$;\mathcal{S}$	=	$\mathcal{S}$	[Spes]	signature	=	
[Sper]	$\mathcal{S};$	=	$\mathcal{S}$	[Spee]	equations	=	
[Spas]	$(\mathcal{S}_1;\mathcal{S}_2);\mathcal{S}_3$	=	$\mathcal{S}_1;\mathcal{S}_2;\mathcal{S}_3$				
[Sg1]	Sg(signature Σ)	=	Σ	[E1]	E(signature Σ)	=	
[Sg2]	Sg(equations $\mathcal{E}$)	=		[E2]	E(equations $\mathcal{E}$)	=	$\mathcal{E}$
[Sg3]	Sg()	=		[E3]	E()	=	

[Sg4] $\mathrm{Sg}(\mathcal{S}_1; \mathcal{S}_2) = \mathrm{Sg}(\mathcal{S}_1); \mathrm{Sg}(\mathcal{S}_2)$ [E4] $\mathrm{E}(\mathcal{S}_1; \mathcal{S}_2) = \mathrm{E}(\mathcal{S}_1) +\!\!+ \mathrm{E}(\mathcal{S}_2)$

We can extend the TermToTerm 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 $\mathcal{S}$.

3.5 Specification Semantics

The semantics of specifications is defined by means of an extension of equational logic to terms with type annotations.

3.5.1 Typed Equational Logic

Equation [tel-ax] 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 (Section 2.3), except for the congruence rule for annotated terms [tel-ann]. Only terms with the same annotation can be equated if they are equal without annotation. Compare this to the congruence rules for application [el-app] and pairing [el-pr] in Section 2.3, 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 (Section 6.4) where equations over types play a role.

imports OLS$^{3.4}$ Substitution$^{B.7}$
exports
  context-free syntax
    Spec "⊢" Eq → Bool
equations

[tel-ax] $$\frac{\mathrm{E}(\mathcal{S}) = \varphi_1^*;\, t_1 \equiv t_2;\, \varphi_2^*}{\mathcal{S} \vdash t_1 \equiv t_2 \;=\; \top}$$

[tel-ann] $$\frac{\mathcal{S} \vdash t_1 \equiv t_2 = \top}{\mathcal{S} \vdash t_1 : \tau \equiv t_2 : \tau \;=\; \top}$$

The standard rules for reflexivity, symmetry, transitivity, substitution and congruence for the other binary operators are not shown.

Proposition 2 *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 $t_1 \equiv t_2 \in \mathcal{E}$,* $\mathrm{type}(t_1) = \mathrm{type}(t_2)$ *then $\mathcal{E} \vdash t \equiv t'$ implies* $\mathrm{type}(t) = \mathrm{type}(t')$.

Proof: by induction on derivations. The property clearly holds for [tel-ax], [tel-refl] and [tel-ann] and equality of types is preserved by symmetry, transitivity, substitution and congruence. □

3.5.2 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 [eval-ann], because the equations of a specification apply only to object terms and not to types.

imports $\mathrm{OLS}^{3.4}$ $\mathrm{Matching}^{B.8}$
exports
context-free syntax

```
eval "(" Spec ")" "⟦" Term "⟧"               → Term
eval "(" Spec ")" "⟦" Term "⟧" "_" Subst     → Term
step "(" Spec ")" "⟦" Term "⟧"               → Term
```

equations

$$\begin{array}{lrl}
\text{[eval-trm]} & \mathrm{eval}(\mathcal{S})[\![t]\!] & = \mathrm{eval}(\mathcal{S})[\![t]\!]_{[]} \\
\text{[eval-ann]} & \mathrm{eval}(\mathcal{S})[\![t : \tau]\!]_{\sigma} & = \mathrm{step}(\mathcal{S})[\![\mathrm{eval}(\mathcal{S})[\![t]\!]_{\sigma} : \sigma(\tau)]\!] \\
\text{[redex]} & \mathrm{step}(\mathcal{S})[\![t]\!] & = \mathrm{eval}(\mathcal{S})[\![t_2]\!]_{\sigma} \\
 & & \quad \textbf{when } \mathrm{E}(\mathcal{S}) = \varphi_1^*;\, t_1 \equiv t_2;\, \varphi_2^*,\ t_1 := t = \sigma \\
\text{[nf]} & \mathrm{step}(\mathcal{S})[\![t]\!] & = t \quad \textbf{otherwise}
\end{array}$$

The evaluation rules for the other operators are straightforward following Section 2.4. Note that recursive applications of eval to the other new operators product and arrow have to be added.

4 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 Section 7 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.

In Section 4.2 (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 Section 4.3 (Module OLS-NWF).

Since fully annotated specifications are difficult to read and write, programmers are 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 Section 4.4 (Module OLS-TA).

Finally, the typechecker defined in Section 4.5 (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.

First we define projection functions to find the type of a function or variable in a signature.

4.1 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.

imports OLS$^{3.4}$
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.

[p1b] $\pi_f(d^*) = \text{nil}$ **when** $d^* =$

[p2b] $\pi_f(f : \tau;\ d^*) = \tau$

[p4b] $\pi_f(d;\ d^*) = \pi_f(d^*)$ **otherwise**

The projection of a variable from a list of declarations is defined similarly.

[p1a] $\pi_x(d^*) = \text{nil}$ **when** $d^* =$

[p2a] $\pi_x(x : \tau;\ d^*) = \tau$

[p4a] $\pi_x(d;\ d^*) = \pi_x(d^*)$ **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.

[pf] $\pi_f(\Sigma) = \pi_f(\text{F}(\Sigma))$

[px] $\pi_x(\Sigma) = \pi_x(\text{V}(\Sigma))$

```
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 5: A fully annotated one-level specification. This is an annotation of the specification in Figure 3.

4.2 Well-formedness (OLS-WF)

Well-formedness judgements on terms characterize the well-formed, fully annotated terms over a signature, i.e., given a signature Σ the set $\mathrm{T}_{fa}(\Sigma)$ defined as

$$\mathrm{T}_{fa}(\Sigma) = \{t \mid t \in \mathrm{Term} \wedge \Sigma \vdash_{\mathrm{term}} t\}$$

is the set of fully annotated terms t that satisfy the well-formedness judgement $\Sigma \vdash_{\mathrm{term}} t$. 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 $\mathrm{T}(\Sigma)$ of plain terms over Σ defined as

$$\mathrm{T}(\Sigma) = \{\mathrm{spine}(t) \mid t \in \mathrm{Term} \wedge \Sigma \vdash_{\mathrm{term}} t\}$$

In this section we define well-formedness of fully annotated terms. In Section 4.4 we define a type assignment function that produces a fully annotated term for a plain term. Figure 5 shows a fully annotated specification.

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.

imports OLS$^{3.4}$ Projection$^{4.1}$ Variables$^{B.6}$ Error-Booleans$^{A.3}$ Term-Analysis$^{3.3}$

exports

context-free syntax

"$\vdash_{\text{spec}}$" Spec	$\rightarrow$ EBool
"$\vdash_{\text{sig}}$" Sig	$\rightarrow$ EBool
"$\vdash_{\text{sorts}}$" Terms	$\rightarrow$ EBool
Sig "$\vdash_{\text{decls}}$" Decls	$\rightarrow$ EBool
Sig "$\vdash_{\text{sort}}$" Term	$\rightarrow$ EBool
Sig "$\vdash_{\text{term}}$" Term	$\rightarrow$ EBool
Sig "$\vdash_{\text{eqs}}$" Eqs	$\rightarrow$ EBool

The well-formedness of a fully annotated specification $\mathcal{S}$ is defined by means of the judgement $\vdash_{\text{spec}} \mathcal{S}$. It is defined in terms of several other judgements of the form $\Sigma \vdash_{\text{r}} r$, 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 Appendix A.3) with a constant $\top$ ('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 $\wedge$ yields $\top$ in case both arguments are $\top$ and yields the addition of the errors otherwise. The asymmetric conjunction $\leadsto$ yields $\top$ if both arguments are $\top$ 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.

[wf-spec] $\quad \vdash_{\text{spec}} \mathcal{S} = \vdash_{\text{sig}} \Sigma \leadsto \Sigma \vdash_{\text{eqs}} \text{E}(\mathcal{S}) \quad$ **when** $\text{Sg}(\mathcal{S}) = \Sigma$

Signatures A signature is well-formed if all its sort, function and variable declarations are well-formed.

[wf-srt] $\quad \vdash_{\text{sig}} \Sigma = \vdash_{\text{sorts}} \text{S}(\Sigma) \wedge \Sigma \vdash_{\text{decls}} \text{F}(\Sigma) \wedge \Sigma \vdash_{\text{decls}} \text{V}(\Sigma)$

The terms declared as sorts in the sorts section should be constant terms, i.e., function symbols.

[wf-sort] $\quad \vdash_{\text{sorts}} f = \top$

[wf-sort] $\vdash_{\text{sorts}} \ = \top$

[wf-sort-prd] $\vdash_{\text{sorts}} \tau_1^+ ; \tau_2^+ = \vdash_{\text{sorts}} \tau_1^+ \wedge \vdash_{\text{sorts}} \tau_2^+$

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.

[wf-d-fun] $\Sigma \vdash_{\text{decls}} f : \tau = \Sigma \vdash_{\text{sort}} \tau \quad \textbf{when} \quad \pi_f(\Sigma) = \tau$

[wf-d-var] $\Sigma \vdash_{\text{decls}} x : \tau = \Sigma \vdash_{\text{sort}} \tau \quad \textbf{when} \quad \pi_x(\Sigma) = \tau$

[wf-d-cnc] $\Sigma \vdash_{\text{decls}} \ = \top$

[wf-d-cnc] $\Sigma \vdash_{\text{decls}} d_1^+ ; d_2^+ = \Sigma \vdash_{\text{decls}} d_1^+ \wedge \Sigma \vdash_{\text{decls}} d_2^+$

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 [wf-sort-fun].

[wf-sort-fun] $\Sigma \vdash_{\text{sort}} f = \top \quad \textbf{when} \quad f \in \mathrm{S}(\Sigma) = \top$

[wf-sort-prd] $\Sigma \vdash_{\text{sort}} t_1 \times t_2 = \Sigma \vdash_{\text{sort}} t_1 \wedge \Sigma \vdash_{\text{sort}} t_2$

[wf-sort-arr] $\Sigma \vdash_{\text{sort}} t_1 \rightarrow t_2 = \Sigma \vdash_{\text{sort}} t_1 \wedge \Sigma \vdash_{\text{sort}} t_2$

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.

[wf-var] $$\frac{\pi_x(\Sigma) = \tau, \ \Sigma \vdash_{\text{sort}} \tau = \top}{\Sigma \vdash_{\text{term}} x : \tau \ = \ \top}$$

[wf-fun] $$\frac{\pi_f(\Sigma) = \tau, \ \Sigma \vdash_{\text{sort}} \tau = \top}{\Sigma \vdash_{\text{term}} f : \tau \ = \ \top}$$

[wf-pr] $$\frac{\mathrm{type}(t_1) \times \mathrm{type}(t_2) = \tau}{\Sigma \vdash_{\text{term}} (t_1, t_2) : \tau \ = \ \Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2}$$

[wf-app] $$\frac{\mathrm{type}(t_1) = \mathrm{type}(t_2) \rightarrow \tau}{\Sigma \vdash_{\text{term}} t_1 \ t_2 : \tau \ = \ \Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2}$$

There is no need to check the well-formedness of the types of applications and pairs, because equations [wf-app] and [wf-pr] preserve well-formedness of type annotations. In equation [wf-pr]: if $\mathrm{type}(t_i)$ are well-formed, then their product is also well-formed. In equation [wf-app]: 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.

[wf-eqn]
$$\frac{\mathrm{type}(t_1) = \mathrm{type}(t_2),\ \mathrm{vars}(t_2) \subseteq \mathrm{vars}(t_1) = \top}{\Sigma \vdash_{\mathrm{eqs}} t_1 \equiv t_2 \ = \ \Sigma \vdash_{\mathrm{term}} t_1 \wedge \Sigma \vdash_{\mathrm{term}} t_2}$$

[wf-eqns-empty]
$$\Sigma \vdash_{\mathrm{eqs}} \ = \ \top$$

[wf-eqns-conc]
$$\Sigma \vdash_{\mathrm{eqs}} \varphi_1^+ ; \varphi_2^+ \ = \ \Sigma \vdash_{\mathrm{eqs}} \varphi_1^+ \wedge \Sigma \vdash_{\mathrm{eqs}} \varphi_2^+$$

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 3 (type soundness) *Well-formed specifications preserve types, i.e., if* $\vdash_{\mathrm{spec}} \mathcal{S}$ *and* $\mathrm{Sg}(\mathcal{S}) \vdash_{\mathrm{term}} t_i$ *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 [wf-eqn]) and by Proposition 2 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 1). □

The condition $\mathrm{Sg}(\mathcal{S}) \vdash_{\mathrm{term}} t_i$ 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 4 *If* $\vdash_{\mathrm{spec}} \mathcal{S}'$*,* $\mathcal{S} = \mathrm{spine}(\mathcal{S}')$*,* $t_1 \neq t_2$ *and* $\mathcal{S} \vdash t_1 \equiv t_2$*, then there are* t_1' *and* t_2' *such that* $\mathrm{spine}(t_1') = t_1$*,* $\mathrm{spine}(t_2') = t_2$*,* $\mathcal{S}' \vdash_{\mathrm{term}} t'_{\{1,2\}}$ *and* $\mathrm{type}(t_1') = \mathrm{type}(t_2')$*.*

```
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 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 3(a).

4.3 Non-wellformedness (OLS-NWF)

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 6 shows the errors for an incorrect version of equations of the natural numbers specification from Section 3. 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

$$\frac{C_1(q,\Sigma),\ldots,C_m(q,\Sigma)}{\Sigma \vdash_{\mathrm{q}} q(r_1,\ldots,r_n) = \Sigma \vdash_{\mathrm{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\mathrm{r}_n} r_n}$$

the error case will be of the form

$$\frac{\neg C_1(q,\Sigma) \vee \cdots \vee \neg C_m(q,\Sigma)}{\Sigma \vdash_{\mathrm{q}} q(r_1,\ldots,r_n) = \Sigma \vdash_{\mathrm{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\mathrm{r}_n} r_n \rightsquigarrow \texttt{"}q\texttt{" not well-formed}}$$

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, which is expressed by the use of the asymmetric conjunction $\rightsquigarrow$. 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

$$\frac{C_1(q,\Sigma),\ldots,C_{i-1}(q,\Sigma),\neg C_i(q,\Sigma)}{\Sigma \vdash_{\mathrm{q}} q(r_1,\ldots,r_n) = \Sigma \vdash_{\mathrm{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\mathrm{r}_n} r_n \rightsquigarrow \texttt{"}q\texttt{" does not have property } C_i}$$

Instead of negating the conditions we can use default equations to deal with the remaining cases.

$$\Sigma \vdash_{\mathrm{q}} q(r_1, \ldots, r_n) \;=\; \Sigma \vdash_{\mathrm{r}_1} r_1 \wedge \ldots \wedge \Sigma \vdash_{\mathrm{r}_n} r_n \rightsquigarrow \texttt{"}q\texttt{" not well-formed} \quad \textbf{otherwise}$$

imports OLS-WF$^{4.2}$ SPEC-Errors$^{B.2}$
equations

Declarations No terms other than constants can be declared as sorts.

[wf-sorts'] $\vdash_{\mathrm{sorts}} \tau =$ " τ " not a well-formed sort declaration **otherwise**

[wf-d-fun'] $\Sigma \vdash_{\mathrm{decls}} f\colon \tau =$ function " f " multiply declared **otherwise**
[wf-d-var'] $\Sigma \vdash_{\mathrm{decls}} x : \tau =$ 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.

[wf-sort-fun'] $\Sigma \vdash_{\mathrm{sort}} f =$ sort " f " not declared **when** $f \in \mathrm{S}(\Sigma) = \bot$
[wf-sort"] $\Sigma \vdash_{\mathrm{sort}} t =$ " t " not a well-formed sort **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.

[wf-fun'] $\Sigma \vdash_{\mathrm{term}} f\colon \tau =$ function " f " not declared **when** $\pi_f(\Sigma) = \mathrm{nil}$
[wf-fun"] $\Sigma \vdash_{\mathrm{term}} f\colon \tau = \Sigma \vdash_{\mathrm{sort}} \tau$ **otherwise**
[wf-var'] $\Sigma \vdash_{\mathrm{term}} x : \tau =$ variable " x " not declared **when** $\pi_x(\Sigma) = \mathrm{nil}$
[wf-var"] $\Sigma \vdash_{\mathrm{term}} x : \tau = \Sigma \vdash_{\mathrm{sort}} \tau$ **otherwise**

Pair and Application

[wf-pr'] $\Sigma \vdash_{\mathrm{term}} (t_1, t_2) : \tau \;=\; (\Sigma \vdash_{\mathrm{term}} t_1 \wedge \Sigma \vdash_{\mathrm{term}} t_2)$
$\rightsquigarrow$ pair " spine(t_1, t_2) " not well-formed
otherwise

```
[wf-app'] $\Sigma \vdash_{\mathrm{term}} t_1\ t_2 : \tau$
      $= (\Sigma \vdash_{\mathrm{term}} t_1 \wedge \Sigma \vdash_{\mathrm{term}} t_2)$
         $\rightsquigarrow$ 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.

$$[\text{wf-term}]\ \frac{\text{type}(t) = \text{top}}{\Sigma \vdash_{\text{term}} t \ = \ \text{term " } t \text{ " not well-formed}}$$

$$\begin{array}{ll}[\text{wf-ann}] & \Sigma \vdash_{\text{term}} (t : \tau_1) : \tau_2 \\ & = \ \Sigma \vdash_{\text{term}} t : \tau_1 \\ & \quad \leadsto \text{annotation of " spine}(t) \text{ " with " } \tau_2 \text{ " not well-formed} \\ & \quad\quad : \text{should be " spine}(\tau_1) \text{ "}\end{array}$$

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.

$$\begin{array}{ll}[\text{wf-term-prd}] & \Sigma \vdash_{\text{term}} t_1 \times t_2 : \tau = \text{term " spine}(t_1 \times t_2) \text{ " not well-formed} \\ [\text{wf-term-arr}] & \Sigma \vdash_{\text{term}} t_1 \to t_2 : \tau = \text{term " spine}(t_1 \to t_2) \text{ " not well-formed} \\ [\text{wf-term-top}] & \Sigma \vdash_{\text{term}} \text{top} : \tau = \text{term " top " not well-formed} \\ [\text{wf-term-nil}] & \Sigma \vdash_{\text{term}} \text{nil} : \tau = \text{term " nil " not well-formed}\end{array}$$

Equations

$$\begin{array}{ll}[\text{wf-eqn'}] & \Sigma \vdash_{\text{eqs}} t_1 \equiv t_2 \\ & = \ (\Sigma \vdash_{\text{term}} t_1 \wedge \Sigma \vdash_{\text{term}} t_2) \\ & \quad \leadsto \text{equation " spine}(t_1) \equiv \text{spine}(t_2) \text{ " not well-formed} \\ & \quad\quad :: \text{if } \neg\ \text{eq}(\text{type}(t_1), \text{type}(t_2)) \\ & \quad\quad\quad \text{then types do not match} \\ & \quad\quad\quad \text{else variables " trms}(\text{vars}(t_2)\ /\ \text{vars}(t_1)) \text{ " of rhs not in lhs} \\ & \textbf{otherwise}\end{array}$$

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 5 *For any term t and signature Σ, $\Sigma \vdash_{\text{term}} t \in \{\top\} \cup \text{Error}$.*

Proof: by induction on terms. □

4.4 Type Assignment (OLS-TA)

Figure 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.

imports OLS$^{3.4}$ Projection$^{4.1}$ Term-Analysis$^{3.3}$
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.

[wsp]
$$\frac{\Sigma = \mathrm{Sg}(\mathcal{S})}{\mathrm{Wsp}[\![\mathcal{S}]\!] \;=\; \text{signature } \Sigma\text{; equations } \mathrm{We}(\Sigma)[\![\mathrm{E}(\mathcal{S})]\!]}$$

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.

[wt-var]
$$\frac{\pi_x(\Sigma) = \tau}{\mathrm{Wt}(\Sigma)[\![x]\!] \;=\; x : \tau}$$

[wt-fun]
$$\frac{\pi_f(\Sigma) = \tau}{\mathrm{Wt}(\Sigma)[\![f]\!] \;=\; f : \tau}$$

[wt-pr]
$$\frac{\mathrm{Wt}(\Sigma)[\![t_1]\!] = t_3,\ \mathrm{Wt}(\Sigma)[\![t_2]\!] = t_4}{\mathrm{Wt}(\Sigma)[\![t_1, t_2]\!] \;=\; (t_3, t_4) : \mathrm{type}(t_3) \times \mathrm{type}(t_4)}$$

[wt-app]
$$\frac{\mathrm{Wt}(\Sigma)[\![t_1]\!] = t_3,\ \mathrm{Wt}(\Sigma)[\![t_2]\!] = t_4}{\mathrm{Wt}(\Sigma)[\![t_1\ t_2]\!] \;=\; t_3\ t_4 : \mathrm{cod}(\mathrm{type}(t_3))}$$

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.

[wt-ann]
$$\frac{\mathrm{Wt}(\Sigma)[\![t]\!] = t'}{\mathrm{Wt}(\Sigma)[\![t : \tau]\!] \;=\; \text{if } \mathrm{eq}(\mathrm{type}(t'), \tau) \text{ then } t' \text{ else } t' : \tau}$$

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 [wf-ann] in Section 4.3. Observe that equation [wt-ann] 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', '×' and '→' are assigned the error type 'nil', since these constructors cannot occur in well-formed terms.

[wt-nil]	$\mathrm{Wt}(\Sigma)[\![\mathrm{nil}]\!] = \mathrm{nil} : \mathrm{nil}$
[wt-top]	$\mathrm{Wt}(\Sigma)[\![\mathrm{top}]\!] = \mathrm{top} : \mathrm{nil}$
[wt-prd]	$\mathrm{Wt}(\Sigma)[\![t_1 \times t_2]\!] = \mathrm{Wt}(\Sigma)[\![t_1]\!] \times \mathrm{Wt}(\Sigma)[\![t_2]\!] : \mathrm{nil}$
[wt-arr]	$\mathrm{Wt}(\Sigma)[\![t_1 \rightarrow t_2]\!] = \mathrm{Wt}(\Sigma)[\![t_1]\!] \rightarrow \mathrm{Wt}(\Sigma)[\![t_2]\!] : \mathrm{nil}$

Equations Both sides of an equation are assigned types.

[we-eqn-1]	$\mathrm{We}(\Sigma)[\![t_1 \equiv t_2]\!] = \mathrm{Wt}(\Sigma)[\![t_1]\!] \equiv \mathrm{Wt}(\Sigma)[\![t_2]\!]$
[we-eqns-0]	$\mathrm{We}(\Sigma)[\![]\!] =$
[we-eqns-n]	$\mathrm{We}(\Sigma)[\![\varphi_1^+; \varphi_2^+]\!] = \mathrm{We}(\Sigma)[\![\varphi_1^+]\!] +\!\!+ \mathrm{We}(\Sigma)[\![\varphi_2^+]\!]$

In Section 4.2 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 6 (correctness of 'Wt') *The 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 [wt-var] until [wt-app] assign types to terms as required by [wf-var] until [wf-app] in Section 4.2.) □

4.5 Typechecking (OLS-TC)

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 (Section 4.2) and type assignment functions (Section 4.4).

imports OLS-NWF$^{4.3}$ OLS-TA$^{4.4}$
exports
  context-free syntax
    tc "(" Sig ")" "[\![" Term "]\!]" → EBool

tc "(" Sig ")" "⟦" Eqs "⟧" → EBool
tc "⟦" Spec "⟧" → EBool

equations

[tc-terms] $\mathrm{tc}(\Sigma)[\![t]\!] = \vdash_{\mathrm{sig}} \Sigma \rightsquigarrow \Sigma \vdash_{\mathrm{term}} \mathrm{Wt}(\Sigma)[\![t]\!]$
[tc-eqns] $\mathrm{tc}(\Sigma)[\![\mathcal{E}]\!] = \vdash_{\mathrm{sig}} \Sigma \rightsquigarrow \Sigma \vdash_{\mathrm{eqs}} \mathrm{We}(\Sigma)[\![\mathcal{E}]\!]$
[tc-spec] $\mathrm{tc}[\![\mathcal{S}]\!] = \vdash_{\mathrm{spec}} \mathrm{Wsp}[\![\mathcal{S}]\!]$

Now we have seen the complete specification of a typechecker for a monomorphic applicative language. In the next two sections we will repeat this exercise for a multi-level polymorphic specification language.

5 Multi-Level Specifications

In the one-level framework of sections 3 and 4 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 7 illustrates the concepts of one-level, two-level and multi-level specifications. The arrow from a signature means that the terms over that signature are used at the target of the arrow.

In the next two sections we define the extension of one-level specifications to multi-level specifications. In this section 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). It is not necessary to read all examples to continue with the rest of the chapter. Indeed for the under-

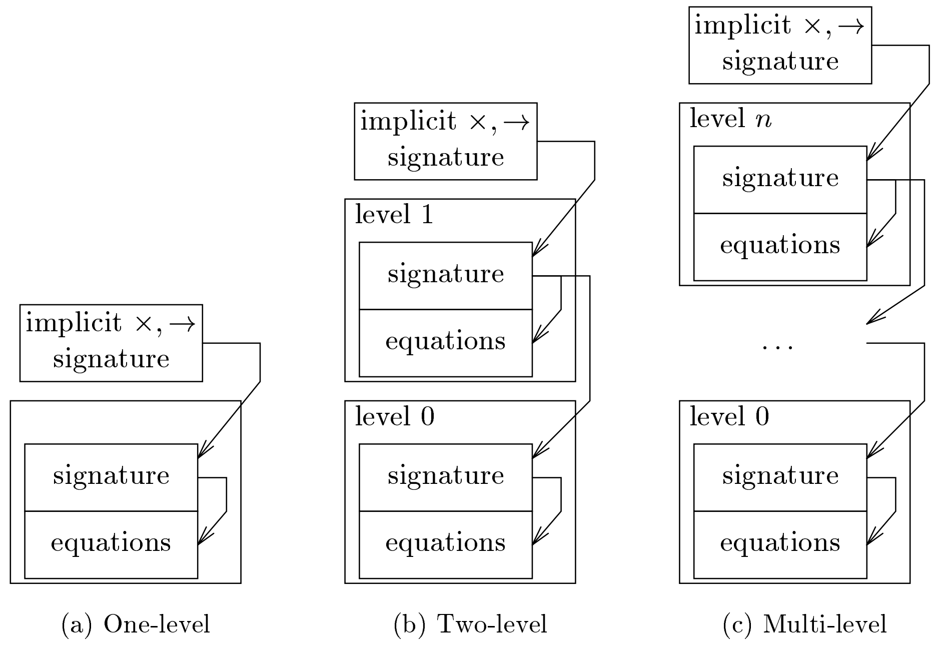


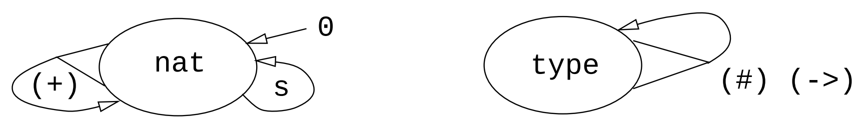
Figure 7: Specifications with one, two and multiple levels of signatures.

standing of some of the later examples it might be a good idea to first continue with Section 6, where the syntax of multi-level specifications is defined.

5.1 Natural Numbers

The running example of sections 3 and 4, 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 (→) and (×). 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 8(a).

```
module nat
level 0
  signature
    sorts nat;
    functions
      0   : nat;
      s   : nat -> nat;
      (+) : nat # nat -> nat;
    variables
```



(a) nat (b) type

Figure 8: Signature diagrams of modules `nat` and `type`

```
      I, J : nat;
  equations
    0 + I    == I;
    s(I) + J == s(I + J);
```

5.2 Signature of Types

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 8(b).

```
module type
level 1
  signature
    sorts type;
    functions
      (#), (->)  : type # type -> type;
    variables
       A, B, C : type;
```

5.3 Functions

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`.

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.

5.4 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 is illustrated by the diagram in Figure 9. This provides the polymorphic functionality defined for arbitrary types to natural numbers.

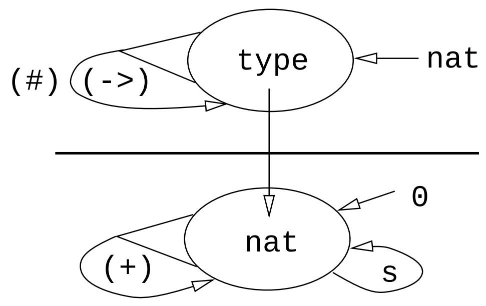


Figure 9: Signature diagram of module `nat-typed`.

```
module nat_typed
imports function, nat;
level 1
  signature
    functions
      nat : type;
```

5.5 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)`.[3] 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 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 dupli-

[3]Recall that `T1 .T2. T3` can be written as an abbreviation of `T2(T1,T2)`.

cation, projection and composition. Finally, the function `pair` is the curried version of the built-in pairing operator `(,)`.

```
module product
imports function;
level 0
  signature
    sorts A # B;
    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);
```

5.6 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 10 illustrates the structure of the algebra. Due to polymorphism, the number of sorts of a specification becomes infinite. Therefore,

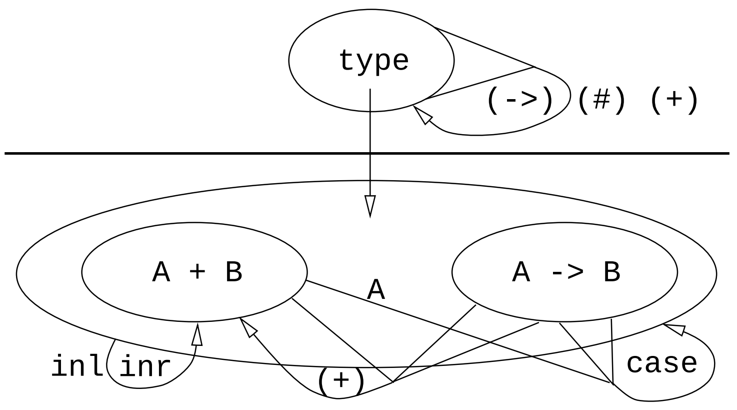


Figure 10: Signature diagram of module `sum`.

signature diagrams do not provide an accurate description 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
    sorts A + B;
    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);
```

5.7 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.[4]

```
module list
imports product, nat_typed;
level 1
  signature
    functions
      list : type -> type;
level 0
  signature
    sorts list(A);
    functions
      []   : list(A);
      (::) : A # list(A) -> list(A);
      (*)  : (A -> B) # list(A) -> list(B);
      (/)  : (A # B -> B) # B -> list(A) -> B;
      (\)  : (A # B -> A) # A -> list(B) -> A;
      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;
```

[4]Note that a variable declaration like `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.

```
(G \ Z)(X :: L)       == (G \ (Z .G. X))(L);

size                  == (s . exl) \ 0;
(++)                  == s(((/) . pair(::)) . exr)(exl);

zip([], L)            == [];
zip(L, [])            == [];
zip(X :: L, X' :: L') == (X, X') :: zip(L, L');
```

5.8 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 11 gives an overview of the signature in module `stratified-stack`.

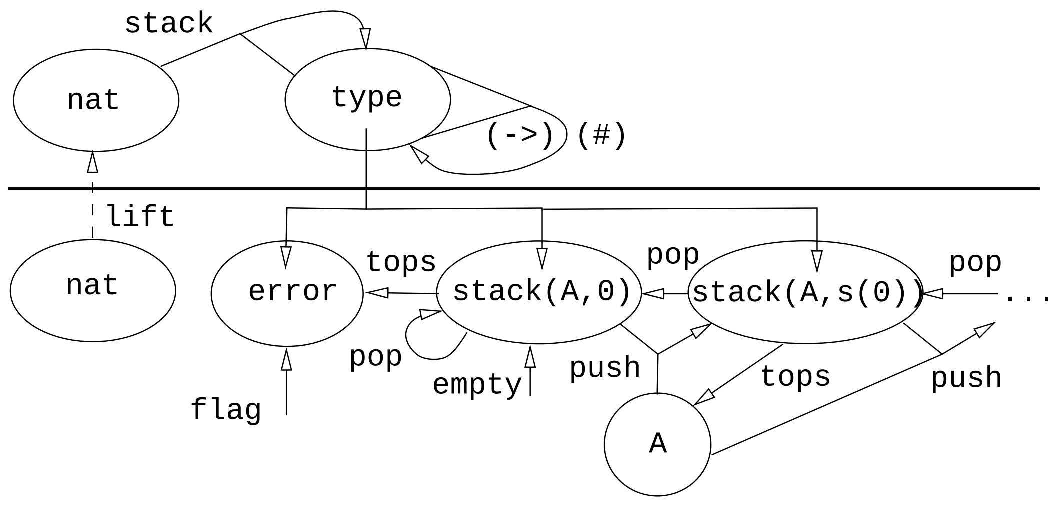


Figure 11: Signature diagram of module `stratified-stack`.

```
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;
```

5.9 Kinds

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 $\{\texttt{type}, (\rightarrow), (\times)\}$ 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;
```

5.10 Generalized Product

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

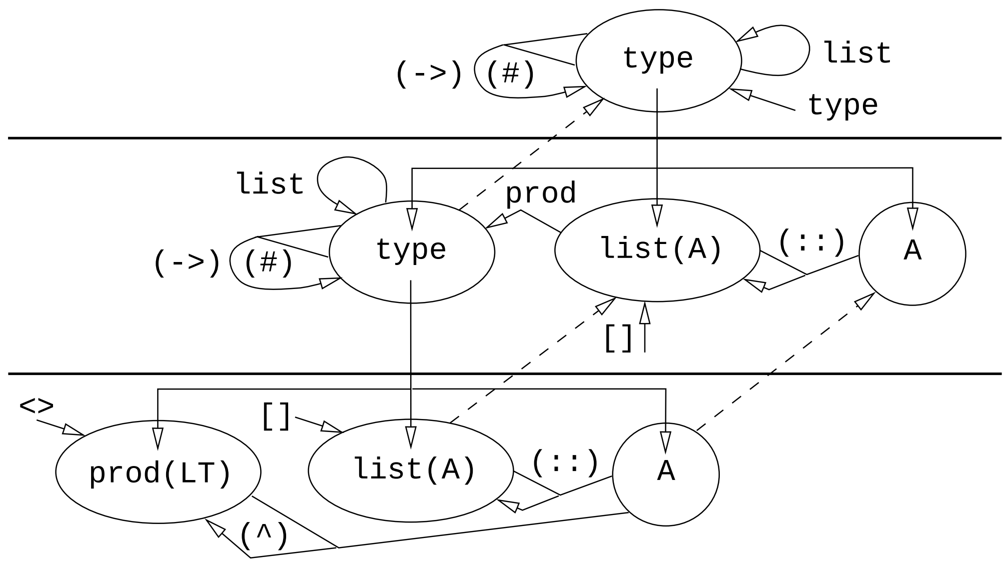


Figure 12: Signature diagram of module `generalized-product` with lifted `list` and `type`.

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 Section 2.1 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.

```
module generalized-product
imports type-type;
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);
    variables
      P : prod(LT);
  equations
    exl(X ^ P) == X;
    exr(X ^ P) == P;
```

In Hindley/Milner type systems it is not possible to construct the type of stratified stacks nor the type of generalized products, because only one sort (`type`) can be used at the level of types. The next paragraph shows an example of a specification that goes even further by using operations at the level of types. This can be expressed in the MLS formalism, but is not supported by the MLS typechecker defined Section 7.

5.11 Generalized Zip

The function `zip` as defined in module `list` above takes a pair of lists into a list of pairs by pairing the heads of both lists until one of the lists is empty. Variants of zip can also be constructed for triples of lists, quadruples of lists and so on. Unzip is the inverse of `zip` that maps a list of products to a product of lists. The following generalized definition of `zip` takes a generalized product of lists into a list of products by tupling the heads of the lists. For instance, if the argument of `zip` has type `prod([list(nat), list(bool), list(list(bool))])`, its result has type `list(prod([nat, bool, list(bool)]))`. The declaration of `zip` has to relate the contents of the `list` types in the domain to the types in the codomain. In the declaration below this is achieved by declaring the domain as `prod(list * LT)`. The type constructor `list` is mapped using operator `(*)` (see module `list`) over the list of types `LT`. This means that the argument of `zip` should be a product with all its arguments of the form `list(A)`. For the example above, we can see that according to the equations for `(*)` (in module `list`) `prod([list(nat), list(bool), list(list(bool))])` is equal to `prod(list * [nat, bool, list(bool)])`. This type can be unified syntactically with the domain type of `zip` (take the substitution [`LT` := `[nat, bool, list(bool)]`], from which the type of the codomain `list(prod([nat, bool, list(bool)]))` follows.

In order to reflect this in the type assignment for specifications, $\mathcal{E}$-unification has to be used. 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 `(*)`. This 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 Section 7.5 does not consider equations over types.

```
module generalized-zip
imports generalized-product, list;
level 0
  signature
    functions
      zip   : prod(list * LT) -> list(prod(LT));
      unzip : list(prod(LT)) -> prod(list * LT);
    variables
      L, M, N : list(A);
  equations
    -- empty tuple
    zip(<>) == [];

    -- singleton tuple
    zip(<[]>)     == [];
    zip(<X :: L>) == <X> :: zip(<L>);

    -- pairs of lists
    zip(<[], L>)          == [];
    zip(<L, []>)          == [];
    zip(<X :: L, Y :: M>) == <X, Y> :: zip(<L, M>);

    -- tuples with at least three lists
    zip(L ^ (M ^ (N ^ P))) == zip(L ^ zip(M ^ (N ^ P)));

    unzip([])       == <>;
    unzip(<X> :: L) == <X :: (exl * L)>;
    unzip((X ^ (Y ^ P)) :: L)  == (X :: (exl * L))
                                  ^ unzip((Y ^ P) :: (exr * L));
```

5.12 Type Classes

Another example of a specification that uses equations over types, is the following module that 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 discussion of MLS examples and proceed to formalize the MLS language in the remaining sections.

6 Syntax of Multi-Level Specifications

In this section we define the syntax of multi-level specifications, extend these with a module mechanism and define the semantics of multi-level specifications.

6.1 Syntax (MLS)

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.

imports OLS[3.4] Naturals[A.4]
exports
  sorts MLS
  context-free syntax

```
                       → MLS
    "level" Nat Spec   → MLS
    MLS ";" MLS        → MLS  {left}
    "(" MLS ")"        → MLS  {bracket}
```

  priorities
    Sig ";"Sig → Sig > MLS ";"MLS → MLS
  variables
    "Γ"[*0-9′*]∗ → 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 (→) and (×) are introduced with the same notation as used to make other infix operators into prefix functions.

exports
  context-free syntax

```
    "(→)" → Fun
    "(×)" → Fun
```

Specification Projections As for the OLS case we define several projection functions to decomposing a specification. 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 ")" → Spec
    max(MLS)                → Nat
    lift(MLS)               → MLS
    drop(MLS)               → MLS
    up(MLS)                 → MLS
    top-sig                 → MLS
    decl(Terms, Term)       → Decls
```

equations

[ms-assoc]	$\Gamma_1; (\Gamma_2; \Gamma_3) \;=\; \Gamma_1; \Gamma_2; \Gamma_3$	[ms-lu]	$; \Gamma \;=\; \Gamma$
[ms-el]	$\text{level } n \;=$	[ms-ru]	$\Gamma; \;=\; \Gamma$

The projection π_n gives the n-th level of a specification.

[p1] $\pi_n() =$

[p2] $\pi_n(\text{level } n\, \mathcal{S}) = \mathcal{S}$

[p3] $\pi_n(\text{level } m\, \mathcal{S}) = \quad \textbf{when} \;\; \text{eq}(n, m) = \bot$

[p4] $\pi_n(\Gamma_1; \Gamma_2) = \pi_n(\Gamma_1); \pi_n(\Gamma_2)$

The function 'max' gives the index of the highest level of a specification. Note that 'max' is also the maximum function on natural numbers.

[max0] $\max() = 0$

[max1] $\max(\text{level } n\, \mathcal{S}) = n$

[max2] $\max(\Gamma_1; \Gamma_2) = \max(\max(\Gamma_1), \max(\Gamma_2))$

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.

[lift1] $\text{lift}() =$

[lift2] $\text{lift}(\text{level } n\, \mathcal{S}) = \text{level succ}(n)\; \mathcal{S}$

[lift3] $\text{lift}(\Gamma_1; \Gamma_2) = \text{lift}(\Gamma_1); \text{lift}(\Gamma_2)$

The function 'drop' lowers all levels by one level and drops the lowest level.

[drop1] $\text{drop}() =$

[drop2] $\text{drop}(\text{level } 0\, \mathcal{S}) =$

[drop3] $\text{drop}(\text{level } n\, \mathcal{S}) = \text{level pred}(n)\; \mathcal{S} \quad \textbf{when} \;\; \text{zero}(n) = \bot$

[drop4] $\text{drop}(\Gamma_1; \Gamma_2) = \text{drop}(\Gamma_1); \text{drop}(\Gamma_2)$

For π, 'lift' and 'drop' we have (modulo associativity and commutativity of ';')

[lift-drop] $\text{lift}(\text{drop}(\Gamma)); \text{level } 0\; \pi_0(\Gamma) = \Gamma$

A multi-level specification is actually 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.

[topsig] $\text{top-sig} \;=\; \text{level } 0 \text{ signature functions } (\times), (\to) : \text{top} \times \text{top} \to \text{top}$

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 $\mathrm{up}(\mathrm{up}(\Gamma)) = \mathrm{up}(\Gamma)$.

[up1] $\mathrm{up}(\Gamma) = \text{top-sig; level 0 signature functions } \mathrm{decl}(\mathrm{S}(\mathrm{Sg}(\pi_0(\Gamma))), \mathrm{top})$
$\quad$ **when** $\mathrm{zero}(\max(\Gamma)) = \top$

[up2] $\mathrm{up}(\Gamma) = \mathrm{drop}(\Gamma)$ **when** $\mathrm{zero}(\max(\Gamma)) = \bot$

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.

[decl1] $\mathrm{decl}(, \tau) =$

[decl2] $\mathrm{decl}(f, \tau) = f : \tau$

[decl3] $\mathrm{decl}(t_1^+; t_2^+, \tau) = \mathrm{decl}(t_1^+, \tau) +\!\!+ \mathrm{decl}(t_2^+, \tau)$

[decl4] $\mathrm{decl}(t, \tau) =$ **otherwise**

6.2 Normalization (MLS-Norm)

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 specifications 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.

imports MLS$^{6.1}$
exports
  context-free syntax
    norm(MLS) → MLS
    norm(Spec) → Spec
equations

[n-1] $\mathrm{norm}(\Gamma) = \text{level 0 } \mathrm{norm}(\pi_0(\Gamma))$ **when** $\max(\Gamma) = 0$

[n-2] $\mathrm{norm}(\Gamma) = \mathrm{lift}(\mathrm{norm}(\mathrm{drop}(\Gamma))); \text{ level 0 } \mathrm{norm}(\pi_0(\Gamma))$
$\quad$ **when** $\mathrm{zero}(\max(\Gamma)) = \bot$

[n-3] $$\begin{aligned}\mathrm{norm}(\mathcal{S}) = {} & \text{signature } \Sigma'; \text{equations } \mathrm{E}(\mathcal{S}) \\ & \textbf{when} \quad \mathrm{Sg}(\mathcal{S}) = \Sigma, \\ & \quad \Sigma' = \text{sorts } \mathrm{S}(\Sigma); \\ & \qquad \text{functions } \mathrm{F}(\Sigma); \\ & \qquad \text{variables } \mathrm{V}(\Sigma);\end{aligned}$$

[n-4] $$d_1^*;\ d;\ d_2^*;\ d;\ d_3^* = d_1^*;\ d;\ d_2^*;\ d_3^*$$

[n-5] $$\text{sorts } t_1^*;\ t;\ t_2^*;\ t;\ t_3^* = \text{sorts } t_1^*;\ t;\ t_2^*;\ t_3^*$$

[n-6] $$\text{equations } \varphi_1^*;\ \varphi;\ \varphi_2^*;\ \varphi;\ \varphi_3^* = \text{equations } \varphi_1^*;\ \varphi;\ \varphi_2^*;\ \varphi_3^*$$

6.3 Modular Multi-Level Specifications (MMLS)

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 Section 5. 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.

imports MLS$^{6.1}$ MLS-Norm$^{6.2}$
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

[mod-cnc] $$M_1^* +\!\!+\ M_2^* = M_1^*\ M_2^*$$

A list of imports denotes the concatenation of the imported specifications.

[imp] $$\text{imports } f_1^+, f_2^+ = \text{imports } f_1^+; \text{imports } f_2^+$$

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.

[p1] $\pi_f() =$

[p1] $\pi_f(\text{module}\, f\, \Gamma;) = \Gamma$

[p3] $\pi_f(\text{module}\, f'\ \Gamma;) = \quad \textbf{when}\ \ \text{eq}(f, f') = \bot$

[p4] $\pi_f(M_1^+\ M_2^+) = \pi_f(M_1^+); \pi_f(M_2^+)$

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$^{B.5}$
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.

[flat-main] $\text{flat}(M^*) = \text{flat}(M^*)[\![M^*]\!]$

[flat-mods1] $\text{flat}(M^*)[\![]\!] =$

[flat-mods2] $\text{flat}(M^*)[\![M_1^+\ M_2^+]\!] = \text{flat}(M^*)[\![M_1^+]\!] +\!\!+ \text{flat}(M^*)[\![M_2^+]\!]$

[flat-mods3] $\text{flat}(M^*)[\![\text{module}\, f\, \Gamma;]\!] = \text{module}\, f\, \text{norm}(\text{flat}(M^*, \{\})[\![\Gamma]\!]);$

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 [flat-imp2]).

[flat-imp1]
$$\frac{f \in \Phi = \bot}{\text{flat}(M^*, \Phi)[\![\text{imports}\, f]\!] \ = \ \text{flat}(M^*, \Phi \cup \{f\})[\![\pi_f(M^*)]\!]}$$

[flat-imp2]
$$\frac{f \in \Phi = \top}{\text{flat}(M^*, \Phi)[\![\text{imports}\, f]\!] \ =}$$

Imports inside other constructs are replaced by distributing 'flat' over all operators except 'imports'.

[flat-ml0] $\text{flat}(M^*, \Phi)[\![\Gamma]\!] = \quad \textbf{when}\ \ \Gamma =$

[flat-ml1]	$\text{flat}(M^*, \Phi)[\![\Gamma_1; \Gamma_2]\!] = \text{flat}(M^*, \Phi)[\![\Gamma_1]\!]; \text{flat}(M^*, \Phi)[\![\Gamma_2]\!]$
[flat-ml2]	$\text{flat}(M^*, \Phi)[\![\text{level } n\, \mathcal{S}]\!] = \text{level } n \text{ flat}(M^*, \Phi)[\![\mathcal{S}]\!]$
[flat-ml3]	$\text{flat}(M^*, \Phi)[\![\pi_n(\Gamma)]\!] = \pi_n(\text{flat}(M^*, \Phi)[\![\Gamma]\!])$
[flat-ml4]	$\text{flat}(M^*, \Phi)[\![\text{lift}(\Gamma)]\!] = \text{lift}(\text{flat}(M^*, \Phi)[\![\Gamma]\!])$
[flat-ml5]	$\text{flat}(M^*, \Phi)[\![\text{drop}(\Gamma)]\!] = \text{drop}(\text{flat}(M^*, \Phi)[\![\Gamma]\!])$
[flat-ml6]	$\text{flat}(M^*, \Phi)[\![\text{up}(\Gamma)]\!] = \text{up}(\text{flat}(M^*, \Phi)[\![\Gamma]\!])$
[flat-ml7]	$\text{flat}(M^*, \Phi)[\![\max(\Gamma)]\!] = \max(\text{flat}(M^*, \Phi)[\![\Gamma]\!])$

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.

6.4 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 [mlel-ann] 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.

imports MLS$^{6.1}$ Substitution$^{B.7}$
exports
  context-free syntax
    MLS "⊢" Eq → Bool
equations

[mlel-ax]
$$\frac{\text{E}(\pi_0(\Gamma)) = \varphi_1^*; t_1 \equiv t_2; \varphi_2^*}{\Gamma \vdash t_1 \equiv t_2 \ = \ \top}$$

[mlel-ann]
$$\frac{\Gamma \vdash t_1 \equiv t_2 = \top,\ \text{up}(\Gamma) \vdash \tau_1 \equiv \tau_2 = \top}{\Gamma \vdash t_1 : \tau_1 \equiv t_2 : \tau_2 \ = \ \top}$$

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 [mlel-ann] have to be syntactically equal. In that case multi-level equational logic reduces to the typed equational logic of Section 3.5.1 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 Section 3.5.2. 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.

7 Typechecking Multi-Level Specifications

In this section 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 Section 7.2. Rules for the complementary cases produce error messages for non-wellformed constructs in Section 7.3. Type assignment functions, defined in Sections 7.4 and 7.5, produce a fully annotated specification for a plain specification an example of which is shown in Figure 13. Finally, the typechecker is defined in Section 7.7 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.

7.1 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 Section 4.1 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.

imports MLS$^{6.1}$ Renaming$^{B.10}$ Term-Sets$^{B.5}$
exports
  context-free syntax
    "π" "_" TermSet "(" Decls ")" → TermSet
    "π" "_" Var "(" MLS ")" → TermSet
    "π" "_" Fun "(" MLS ")" → TermSet
equations

The projection function π_t finds the types of a set of functions or variables in a list of declarations.

[p1] $$\pi_\Phi() = \{\}$$
[p2] $$\pi_\Phi(f : \tau) = \text{if } f \in \Phi \text{ then } \{\tau\} \text{ else } \{\}$$

[p2] $\pi_\Phi(x : \tau) = \text{if } x \in \Phi \text{ then } \{\tau\} \text{ else } \{\}$

[p3] $\pi_\Phi(d_1^+;\, d_2^+) = \pi_\Phi(d_1^+) \cup \pi_\Phi(d_2^+)$

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.

[pf] $\pi_f(\Gamma) = \pi_{\{f\}}(\mathrm{F}(\mathrm{Sg}(\pi_0(\Gamma))))$

[px] $\pi_x(\Gamma) = \pi_{\{x;\ \mathrm{base}(x)\}}(\mathrm{V}(\mathrm{Sg}(\pi_0(\Gamma))))$

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 Appendix B.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.

7.2 Well-Formedness (MLS-WF)

As in the one-level case in Section 4.2, 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 13.

imports MLS$^{6.1}$ MLS-Projection$^{7.1}$ Error-Booleans$^{A.3}$ SPEC-Errors$^{B.2}$
MLS-TA-Aux$^{7.4}$ Matching$^{B.8}$ Term-Analysis$^{3.3}$

exports

context-free syntax

"$\vdash_{\mathrm{mls}}$" MLS	$\rightarrow$ EBool
MLS "$\vdash_{\mathrm{spec}}$" Spec	$\rightarrow$ EBool
MLS "$\vdash_{\mathrm{sig}}$" Sig	$\rightarrow$ EBool
MLS "$\vdash_{\mathrm{sorts}}$" Terms	$\rightarrow$ EBool
MLS "$\vdash_{\mathrm{decls}}$" Decls	$\rightarrow$ EBool
MLS "$\vdash_{\mathrm{sort}}$" Term	$\rightarrow$ EBool
MLS "$\vdash_{\mathrm{trm}}$" Term ":" Term	$\rightarrow$ EBool
MLS "$\vdash_{\mathrm{term}}$" Term	$\rightarrow$ EBool
MLS "$\vdash_{\mathrm{eqs}}$" Eqs	$\rightarrow$ 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.

[wf-spec-1] $\vdash_{\mathrm{mls}} \Gamma = \Gamma \vdash_{\mathrm{spec}} \pi_0(\Gamma)$ **when** $\max(\Gamma) = 0$

[wf-spec-1] $\vdash_{\mathrm{mls}} \Gamma = \vdash_{\mathrm{mls}} \mathrm{up}(\Gamma) \rightsquigarrow \Gamma \vdash_{\mathrm{spec}} \pi_0(\Gamma)$
when $\mathrm{zero}(\max(\Gamma)) = \bot$

```
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 13: Example of a fully annotated two-level specification. Observe that the types at level 0 are fully annotated terms over level 1.

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 [wf-spec] gives precedence to signature errors over equation errors.

[wf-spec] $$\Gamma \vdash_{\text{spec}} \mathcal{S} = \Gamma \vdash_{\text{sig}} \text{Sg}(\mathcal{S}) \rightsquigarrow \Gamma \vdash_{\text{eqs}} \text{E}(\mathcal{S})$$

A signature is well-formed if the sorts section contains well-formed sort declarations and if the function and variable declarations are well-formed.

[wf-sig] $$\Gamma \vdash_{\text{sig}} \Sigma = \Gamma \vdash_{\text{sorts}} \text{S}(\Sigma) \rightsquigarrow \Gamma \vdash_{\text{decls}} \text{F}(\Sigma) \wedge \Gamma \vdash_{\text{decls}} \text{V}(\Sigma)$$

The terms in a sort declaration at level n should be well-formed terms over level $n+1$.

[wf-sorts1] $$\frac{\text{up}(\Gamma) \vdash_{\text{term}} \tau = \top}{\Gamma \vdash_{\text{sorts}} \tau \ = \ \top}$$

[wf-sorts2] $$\Gamma \vdash_{\text{sorts}} \tau_1^+; \tau_2^+ \ = \ \Gamma \vdash_{\text{sorts}} \tau_1^+ \wedge \Gamma \vdash_{\text{sorts}} \tau_2^+$$

[wf-sorts3] $\Gamma \vdash_{\text{sorts}} \; = \; \top$

A function or variable declaration is well-formed if its type is a well-formed sort.

[wf-decls-vd] $\Gamma \vdash_{\text{decls}} x : \tau = \Gamma \vdash_{\text{sort}} \tau$
[wf-decls-fd] $\Gamma \vdash_{\text{decls}} f : \tau = \Gamma \vdash_{\text{sort}} \tau$
[wf-decls-empty] $\Gamma \vdash_{\text{decls}} = \top$
[wf-decls-conc] $\Gamma \vdash_{\text{decls}} d_1^+ ; d_2^+ = \Gamma \vdash_{\text{decls}} d_1^+ \wedge \Gamma \vdash_{\text{decls}} d_2^+$

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.

[wf-sort] $$\frac{\text{zero}(\max(\Gamma)) = \bot, \;\; \{\mathrm{S}(\mathrm{Sg}(\pi_0(\Gamma)))\} \geq t = \top}{\Gamma \vdash_{\text{sort}} t \; = \; \text{up}(\Gamma) \vdash_{\text{term}} t}$$

The predicate $\Phi \gtrdot t$ (Appendix B.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.

[wf-sort] $$\frac{\text{zero}(\max(\Gamma)) = \top}{\Gamma \vdash_{\text{sort}} t \; = \; \text{up}(\Gamma) \vdash_{\text{term}} t}$$

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 $\vdash_{\text{term.}}$ 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.

[wf-term] $\Gamma \vdash_{\text{term}} t = \Gamma \vdash_{\text{trm}} \text{term}(t) : \text{type}(t)$

The term 'top' has type 'top'. Since 'top' can not be declared as a function, this is the only possible type it can have.

[wf-top] $\Gamma \vdash_{\text{trm}} \text{top} : \text{top} \; = \; \top$

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.

[wf-fun] $$\frac{\pi_f(\Gamma) \geq \tau = \top}{\Gamma \vdash_{\text{trm}} f : \tau \;=\; \top}$$

[wf-var] $$\frac{\pi_x(\Gamma) = \Phi,\ \Phi \geq \tau \vee \text{empty}(\Phi) = \top}{\Gamma \vdash_{\text{trm}} x : \tau \;=\; \Gamma \vdash_{\text{sort}} \tau}$$

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.

[wf-pr] $$\frac{\text{term}(\tau) = \text{type}(t_1) \times \text{type}(t_2)}{\Gamma \vdash_{\text{trm}} t_1, t_2 : \tau \;=\; \Gamma \vdash_{\text{term}} t_1 \wedge \Gamma \vdash_{\text{term}} t_2}$$

[wf-app] $$\frac{\text{term}(\text{type}(t_1)) = \text{type}(t_2) \to \tau}{\Gamma \vdash_{\text{trm}} t_1\ t_2 : \tau \;=\; \Gamma \vdash_{\text{term}} t_1 \wedge \Gamma \vdash_{\text{term}} t_2}$$

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 $(\to)$ should be. For this purpose, the function 'bterm' (Section 3.3) is used to strip the annotation from the the declared types.

[wf-prd] $$\frac{\text{bterm}{*}(\pi_{(\times)}(\Gamma)) \geq \text{type}(t_1) \times \text{type}(t_2) \to \tau = \top}{\Gamma \vdash_{\text{trm}} t_1 \times t_2 : \tau \;=\; \Gamma \vdash_{\text{term}} t_1 \wedge \Gamma \vdash_{\text{term}} t_2}$$

[wf-arr] $$\frac{\text{bterm}{*}(\pi_{(\to)}(\Gamma)) \geq \text{type}(t_1) \times \text{type}(t_2) \to \tau = \top}{\Gamma \vdash_{\text{trm}} t_1 \to t_2 : \tau \;=\; \Gamma \vdash_{\text{term}} t_1 \wedge \Gamma \vdash_{\text{term}} t_2}$$

Equations An equation is well-formed if both sides have the same type, the variables of the rhs are contained in the variables of the lhs and all occurrences of a variable on both sides have the same type.

[wf-eqn] $$\frac{\begin{array}{c}\text{type}(t_1) = \text{type}(t_2),\ \text{vars}(t_2) \subseteq \text{vars}(t_1) = \top,\\ \text{var-types}(\text{avars}(t_1, t_2)) = []\end{array}}{\Gamma \vdash_{\text{eqs}} t_1 \equiv t_2 \;=\; \Gamma \vdash_{\text{term}} t_1 \wedge \Gamma \vdash_{\text{term}} t_2}$$

[wf-eqns-empty] $\Gamma \vdash_{\text{eqs}} \; = \; \top$

[wf-eqns-conc] $\Gamma \vdash_{\text{eqs}} \varphi_1^+; \varphi_2^+ \; = \; \Gamma \vdash_{\text{eqs}} \varphi_1^+ \wedge \Gamma \vdash_{\text{eqs}} \varphi_2^+$

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 7 (type soundness) *Well-formed specifications preserve types, i.e, let* Γ *be a fully annotated multi-level specification such that declarations in* Γ *use only free type constructors, if* $\vdash_{\text{mls}} \Gamma$ *and* $\Gamma \vdash_{\text{term}} t_i$ *then* $\pi_0(\Gamma) \vdash t_1 \equiv t_2$ *implies* $\text{type}(t_1) = \text{type}(t_2)$.

Proof: Since Γ is well-formed, all equations in $\pi_0(\Gamma)$ have equal types in the lhs and rhs and typed equational logic is type preserving for equations with that property (Proposition 2). □

The following proposition relates equalities over plain terms to equalities over fully annotated terms.

Proposition 8 *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* $\vdash_{\text{mls}} \Gamma$*, then* $\Gamma \vdash t_1 \equiv t_2$ *implies* $\text{spine}(\Gamma) \vdash \text{spine}(t_1) \equiv \text{spine}(t_2)$

In Section 7.6 we discuss the requirements for the reverse implication; when does equality in the plain specification preserve types?

7.3 Non-wellformedness (MLS-NWF)

The generation of error messages for the non-wellformed cases is very similar to Section 4.3, therefore only the case of a non-wellformed application is presented.

imports MLS-WF$^{7.2}$ SPEC-Errors$^{B.2}$

equations

```
[wf-app'] $\Gamma \vdash_{\text{trm}} t_1\ t_2 : \tau$
          $= (\Gamma \vdash_{\text{term}} t_1 \wedge \Gamma \vdash_{\text{term}} t_2)$
            $\rightsquigarrow$ application " spine($t_1\ t_2$) " not well-formed
              :: if $\neg$ A $\to$ 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($\tau$)
                       " does not match type of codomain " cod(term(type($t_1$))) "
          otherwise
```

7.4 Preliminaries for Type Assignment (MLS-TA-Aux)

In the next section we will define the type assignment functions for the multi-level case. First, we define several auxiliary functions that will make the definition of type assignment easier. The two major complications 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 Appendix B.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 Appendix B.5).

imports Renaming$^{B.10}$

exports
  context-free syntax
    TermSet "$\bowtie$" TermSet $\rightarrow$ TermSet {**non-assoc**}
equations

$$\text{[join]} \quad \frac{\text{vars}(\Phi_2) = \Phi_2',\ \text{rn tvars}(\Phi_1) \cap \Phi_2'[\text{vars}(\Phi_1) \cup \Phi_2']*(\Phi_1) = \Phi_3,\quad \text{vars}(\Phi_3) = \Phi_3',\ \text{rn tvars}(\Phi_2) \cap \Phi_3'[\Phi_2' \cup \Phi_3']*(\Phi_2) = \Phi_4}{\Phi_1 \bowtie \Phi_2 \ = \ \Phi_3 \times \Phi_4}$$

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)[5], 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. Furthermore, from the non-wellformed term the well-formedness judgements can find out the cause of the error.

imports Term-Analysis$^{3.3}$ Term-Functions$^{B.3}$
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
    "G"$[0\text{-}9']*$ $\rightarrow$ (Term $\Rightarrow$ Bool $\times$ TermSet)
equations

[mf0] $G*(\Phi) = G*(\bot, \Phi)$

[mf1] $G*(b, \{\}) = \{\}$

[mf2] $G*(b, \{t\}) = \text{if } b \wedge \neg\, b' \text{ then } \{\} \text{ else } \Phi$ **when** $G(t) = \langle b', \Phi\rangle$

[mf3] $G*(b, \{t; t^+\}) = (\text{if } b' \text{ then } \Phi \text{ else } \{\}) \cup G*(b \vee b', \{t^+\})$
when $G(t) = \langle b', \Phi\rangle$

[5]Note that we instructed ToLaTeX to typeset the sort identifier Term2BoolXTermSet as (Term $\Rightarrow$ Bool $\times$ TermSet)

For functions of sort (Term $\Rightarrow$ Bool $\times$ 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$^{6.1}$
exports
  context-free syntax
    Fun ":" TermSet $\rightarrow$ Decl
    Var ":" TermSet $\rightarrow$ Decl
    Term ":" TermSet $\rightarrow$ TermSet
  priorities
    Term ":"TermSet $\rightarrow$ TermSet $>$ TermSet "$\cup$"TermSet $\rightarrow$ TermSet

[decl1]	$d_1^*; f : \{\}; d_2^* = d_1^*; d_2^*$
[decl2]	$d_1^*; f : \{t\}; d_2^* = d_1^*; f : t; d_2^*$
[decl3]	$d_1^*; f : \{t_1^+; t_2^+\}; d_2^* = d_1^*; f : \{t_1^+\}; f : \{t_2^+\}; d_2^*$
[decl4]	$d_1^*; x : \{\}; d_2^* = d_1^*; d_2^*$
[decl5]	$d_1^*; x : \{t\}; d_2^* = d_1^*; x : t; d_2^*$
[decl6]	$d_1^*; x : \{t_1^+; t_2^+\}; d_2^* = d_1^*; x : \{t_1^+\}; x : \{t_2^+\}; d_2^*$
[trm1]	$t : \{\} = \{t : \text{nil}\}$
[trm2]	$t : \{\tau\} = \{t : \tau\}$
[trm3]	$t : \{t_1^+; t_2^+\} = t : \{t_1^+\} \cup t : \{t_2^+\}$

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 functions 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$^{B.9}$
exports
  context-free syntax

```
    var-types(TermSet) → Subst⊥
    var-eqs(TermSet)   → Eqs
```

equations

[var-types1] $\text{var-types}(\Phi) = \text{mgu}(\text{var-eqs}(\Phi))$

[var-eqs1] $\text{var-eqs}(\{\}) =$

[var-eqs2] $\text{var-eqs}(\{x : \tau_1;\ t_1^*;\ x : \tau_2;\ t_2^*\}) = \tau_1 \equiv \tau_2 +\!\!+\ \text{var-eqs}(\{t_1^*;\ x : \tau_2;\ t_2^*\})$

[var-eqs2] $\text{var-eqs}(\{x : \tau_1;\ t^*\}) = \text{var-eqs}(\{t^*\})$ **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$^{7.1}$
exports
context-free syntax

```
    new-var "(" MLS ")"                          → Term
    nv "(" MLS ")" "(" TermSet "," TermSet ")"   → Term
```

equations

[new-var-0] $\text{new-var}(\Gamma) = \text{nv}(\Gamma)(\{\}, \{\})$

[new-var-1] $\text{nv}(\Gamma)(\Phi, \{\}) = \text{nv}(\Gamma)(\Phi', \Phi')$ **when** $\Phi' = \text{add}(\texttt{Q}, \Phi)$

[new-var-2] $\text{nv}(\Gamma)(\Phi, \{x;\ t^*\}) = \text{if}\ \text{empty}(\pi_x(\Gamma))\ \text{then}\ x\ \text{else}\ \text{nv}(\Gamma)(\Phi, \{t^*\})$

7.5 Type Assignment (MLS-TA)

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.

imports MLS$^{6.1}$ MLS-TA-Aux$^{7.4}$ MLS-Projection$^{7.1}$ Term-Analysis$^{3.3}$ Matching$^{B.8}$
exports
context-free syntax

```
    "Wm" "⟦" MLS "⟧"              → MLS
```

"Wsp" "(" MLS ")" "⟦" Spec "⟧"	$\rightarrow$ Spec
"Wsg" "(" MLS ")" "⟦" Sig "⟧"	$\rightarrow$ Sig
"Wd" "(" MLS ")" "⟦" Decls "⟧"	$\rightarrow$ Decls
"Ws" "(" MLS ")" "⟦" Term "⟧"	$\rightarrow$ TermSet
"Wss" "(" MLS ")" "⟦" Terms "⟧"	$\rightarrow$ Terms
"Wtv" "(" MLS ")" "⟦" Term "⟧"	$\rightarrow$ TermSet
"Wt" "(" MLS ")" "⟦" Term "⟧"	$\rightarrow$ TermSet
"Wts" "(" MLS ")" "⟦" Terms "⟧"	$\rightarrow$ Terms
"We" "(" MLS ")" "⟦" Eqs "⟧"	$\rightarrow$ 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.

[wm-1]
$$\frac{\text{zero}(\max(\Gamma)) = \top}{\text{Wm}[\![\Gamma]\!] \ = \ \text{level } 0 \text{ Wsp}(\text{lift}(\text{up}(\Gamma)))[\![\pi_0(\Gamma)]\!]}$$

[wm-n]
$$\frac{\text{zero}(\max(\Gamma)) = \bot, \ \text{lift}(\text{Wm}[\![\text{up}(\Gamma)]\!]) = \Gamma'}{\text{Wm}[\![\Gamma]\!] \ = \ \Gamma'; \text{level } 0 \text{ Wsp}(\Gamma')[\![\pi_0(\Gamma)]\!]}$$

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.

[wsp]
$$\frac{\text{Wsg}(\Gamma)[\![\text{Sg}(\mathcal{S})]\!] = \Sigma, \ \Gamma' = \text{level } 0 \text{ signature } \Sigma}{\text{Wsp}(\Gamma)[\![\mathcal{S}]\!] \ = \ \text{signature } \Sigma; \text{equations We}(\Gamma; \Gamma')[\![\text{E}(\mathcal{S})]\!]}$$

Assign types to each section of a signature.

[w-cnc]
$$\frac{\text{sorts Wss}(\Gamma)[\![\text{S}(\Sigma)]\!] = \Sigma_2, \ \Gamma' = \Gamma; \text{level } 0 \text{ signature } \Sigma_2}{\begin{array}{rcl}\text{Wsg}(\Gamma)[\![\Sigma]\!] & = & \Sigma_2; \\ & & \text{functions Wd}(\Gamma')[\![\text{F}(\Sigma)]\!]; \\ & & \text{variables Wd}(\Gamma')[\![\text{V}(\Sigma)]\!]\end{array}}$$

The sorts in the declarations of sorts, functions and variables are treated as terms over the signature at the next level.

$$\begin{array}{lrcl}
\text{[wd-e]} & \text{Wd}(\Gamma)[\![]\!] & = & \\
\text{[wd-fun]} & \text{Wd}(\Gamma)[\![f : \tau]\!] & = & f : \text{Ws}(\Gamma)[\![\tau]\!] \\
\text{[wd-var]} & \text{Wd}(\Gamma)[\![x : \tau]\!] & = & x : \text{rn vars}(\Phi)[\{x\}]*(\Phi) \\
 & & & \textbf{when } \ \text{Ws}(\Gamma)[\![\tau]\!] = \Phi \\
\text{[wd-cnc]} & \text{Wd}(\Gamma)[\![d_1^+; \, d_2^+]\!] & = & \text{Wd}(\Gamma)[\![d_1^+]\!] +\!\!\!+ \text{Wd}(\Gamma)[\![d_2^+]\!]
\end{array}$$

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.

[ws-1] $\quad \text{Ws}(\Gamma)[\![\tau]\!] = \text{srt}(\{\text{S}(\text{Sg}(\pi_0(\Gamma)))\})*(\text{Wtv}(\text{up}(\Gamma))[\![\tau]\!])$

[sort-1] $\mathrm{srt}(\Phi)(\tau) = \langle \top, \{\tau\}\rangle$ **when** $\Phi \geq \tau = \top$
[sort-2] $\mathrm{srt}(\Phi)(\tau) = \langle \bot, \{\tau\}\rangle$ **otherwise**

A list of sort terms at level n is a list of terms over level $n+1$.

[ws-def] $$\mathrm{Wss}(\Gamma)[\![ts]\!] = \mathrm{Wts}(\mathrm{up}(\Gamma))[\![ts]\!]$$

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' (Section 7.4) to make the types of different occurrences of the same variable equal.

[wt-vars] $$\frac{\mathrm{Wt}(\Gamma)[\![t]\!] = \Phi,\ \text{var-types}(\mathrm{avars}(\Phi)) = \sigma_\bot}{\mathrm{Wtv}(\Gamma)[\![t]\!] \ = \ \text{if fail?}(\sigma_\bot) \text{ then } \Phi \text{ else } \Downarrow_\bot(\sigma_\bot)*(\Phi)}$$

Functions and Variables Functions get assigned the type from the declaration in the signature.

[wt-fun] $$\mathrm{Wt}(\Gamma)[\![f]\!] = f : \pi_f(\Gamma)$$

The type assignment to variables is somewhat more complicated since undeclared variables are taken into account according to the following rules. Equation [wt-var1] deals with variables in types of the top signature. Equation [wt-var2] 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.

[wt-var1] $$\frac{\mathrm{up}(\Gamma) = \text{top-sig}}{\mathrm{Wt}(\Gamma)[\![x]\!] \ = \ \{x\}}$$

[wt-var2] $$\frac{\mathrm{up}(\Gamma) \neq \text{top-sig},\ \pi_x(\Gamma) = \Phi,\quad \text{if empty}(\Phi) \text{ then } \mathrm{Wt}(\mathrm{up}(\Gamma))[\![\text{new-var}(\mathrm{up}(\Gamma))]\!] \text{ else } \Phi = \{ts\}}{\mathrm{Wt}(\Gamma)[\![x]\!] \ = \ x : \{\text{rn vars} *(\ ts)[\{x\}]*(ts)\}}$$

Nil and Top Nil can not occur in well-formed specifications. Top can only occur as a top-level type.

[wt-nil] $\mathrm{Wt}(\Gamma)[\![\mathrm{nil}]\!] = \{\mathrm{nil} : \mathrm{nil}\}$
[wt-top] $\mathrm{Wt}(\Gamma)[\![\mathrm{top}]\!] = \{\mathrm{top} : \mathrm{top}\}$

Auxiliary Functions For the type assignment of non-atomic terms we need the following auxiliary functions.

hiddens

context-free syntax

$$\begin{array}{ll}
\text{srt(TermSet)} & \to (\text{Term} \Rightarrow \text{Bool} \times \text{TermSet}) \\
\text{app(MLS)} & \to (\text{Term} \Rightarrow \text{Bool} \times \text{TermSet}) \\
\text{pr(MLS)} & \to (\text{Term} \Rightarrow \text{Bool} \times \text{TermSet}) \\
\text{arr} & \to (\text{Term} \Rightarrow \text{Bool} \times \text{TermSet}) \\
\text{prd} & \to (\text{Term} \Rightarrow \text{Bool} \times \text{TermSet}) \\
\text{ann} & \to (\text{Term} \Rightarrow \text{Bool} \times \text{TermSet}) \\
\text{eqn} & \to (\text{Term} \Rightarrow \text{Bool} \times \text{Eqs}) \\
\text{new-arrow(MLS)} & \to \text{Term}
\end{array}$$

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.

[wt-app] $$\mathrm{Wt}(\Gamma)[\![t_1\ t_2]\!] \;=\; \mathrm{app}(\Gamma) *((\mathrm{Wt}(\Gamma)[\![t_1]\!] \bowtie \mathrm{Wt}(\Gamma)[\![t_2]\!]) \bowtie \{\mathsf{a} : \text{new-arrow}(\Gamma)\}\ )$$

[app1] $$\frac{\mathrm{mgu}(\mathrm{type}(t_1) \equiv \tau_1;\ \mathrm{type}(t_2) \equiv \tau_2) = \sigma}{\mathrm{app}(\Gamma)((t_1,\, t_2),\, \mathsf{a} : (\tau_1,\, \tau_2,\, \tau_3)) \;=\; \langle \top,\, \{\sigma(t_1\ t_2 : \tau_3)\}\rangle}$$

[app2] $$\mathrm{app}(\Gamma)((t_1,\, t_2),\, \tau) \;=\; \langle \bot,\, \{t_1\ t_2 : \mathrm{nil}\}\rangle \quad \textbf{otherwise}$$

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.

[na] $$\frac{\begin{array}{c}\text{new-var}(\mathrm{up}(\Gamma)) = x,\ \ x' = \mathrm{prime}(x),\ \ \tau_0 = x \to x', \\ \text{if zero}(\max(\Gamma)) \text{ then } \{\tau_0\} \text{ else } \mathrm{Wt}(\mathrm{up}(\Gamma))[\![\tau_0]\!] = \{\tau_1;\, t^*\}\end{array}}{\text{new-arrow}(\Gamma) \;=\; \tau_1,\, \mathrm{dom}(\mathrm{term}(\tau_1)),\, \mathrm{cod}(\mathrm{term}(\tau_1))}$$

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.

[wt-pr] $$\mathrm{Wt}(\Gamma)[\![t_1,\, t_2]\!] = \mathrm{pr}(\Gamma) * (\mathrm{Wt}(\Gamma)[\![t_1]\!] \bowtie \mathrm{Wt}(\Gamma)[\![t_2]\!])$$

[pr1] $$\mathrm{pr}(\Gamma)(t_1,\, t_2) = \langle \top,\, \{(t_1,\, t_2) : \mathrm{type}(t_1) \times \mathrm{type}(t_2)\}\rangle \quad \textbf{when}\ \ \mathrm{zero}(\max(\Gamma)) = \top$$

[pr2] $$\mathrm{pr}(\Gamma)(t_1,\, t_2) = \langle \top,\, t_1,\, t_2 : \mathrm{Wt}(\mathrm{up}(\Gamma))[\![\mathrm{type}(t_1) \times \mathrm{type}(t_2)]\!]\rangle \quad \textbf{when}\ \ \mathrm{zero}(\max(\Gamma)) = \bot$$

Arrow and Product Arrow and product are defined in terms of application of the functions $(\to)$ and $(\times)$ to their arguments. After type assignment the binary notation is restored for readability.

[wt-arr] $\mathrm{Wt}(\Gamma)[\![t_1 \to t_2]\!] = \mathrm{arr}{*}(\mathrm{Wt}(\Gamma)[\![(\to)\ (t_1, t_2)]\!])$

[arr1] $\mathrm{arr}(t) = \langle \top, \{t_1 \to t_2 : \mathrm{type}(t)\}\rangle$ $\quad$ **when** $\mathrm{bapp}(t) = (\to)\ (t_1, t_2)$

[arr2] $\mathrm{arr}(t) = \langle \bot, \{t\}\rangle$ **otherwise**

[wt-prd] $\mathrm{Wt}(\Gamma)[\![t_1 \times t_2]\!] = \mathrm{prd}{*}(\mathrm{Wt}(\Gamma)[\![(\times)\ (t_1, t_2)]\!])$

[prd1] $\mathrm{prd}(t) = \langle \top, \{t_1 \times t_2 : \mathrm{type}(t)\}\rangle$ $\quad$ **when** $\mathrm{bapp}(t) = (\times)\ (t_1, t_2)$

[prd2] $\mathrm{prd}(t) = \langle \bot, \{t\}\rangle$ **otherwise**

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.

[wt-ann] $\mathrm{Wt}(\Gamma)[\![t : \tau]\!] = \mathrm{ann}{*}(\mathrm{Wt}(\Gamma)[\![t]\!] \bowtie \mathtt{a} : \mathrm{Wt}(\mathrm{up}(\Gamma))[\![\tau]\!])$

[ann1] $\mathrm{ann}(t : \tau_1, \mathtt{a} : \tau_2) = \langle \top, \{\sigma(t : \tau_2)\}\rangle$ **when** $\mathrm{mgu}(\tau_1 \equiv \tau_2) = \sigma$

[ann2] $\mathrm{ann}(t, \mathtt{a} : \tau) = \langle \bot, \{t : \tau\}\rangle$ **otherwise**

Lists of Terms

[wt-terms1] $\mathrm{Wts}(\Gamma)[\![]\!] =$

[wt-terms2] $\mathrm{Wts}(\Gamma)[\![t]\!] = ts$ **when** $\{ts\} = \mathrm{Wt}(\Gamma)[\![t]\!]$

[wt-terms3] $\mathrm{Wts}(\Gamma)[\![t_1^+; t_2^+]\!] = \mathrm{Wts}(\Gamma)[\![t_1^+]\!] +\!\!+ \mathrm{Wts}(\Gamma)[\![t_2^+]\!]$

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.

[wt-eqn] $\mathrm{We}(\Gamma)[\![t_1 \equiv t_2]\!] = \mathrm{eqn}{*}(\mathrm{Wt}(\Gamma)[\![t_1]\!] \bowtie \mathrm{Wt}(\Gamma)[\![t_2]\!])$

[eqn1] $\mathrm{eqn}(t_1, t_2) = \langle \top, \sigma_2 \circ \sigma_1(t_1 \equiv t_2)\rangle$
when $\mathrm{var\text{-}types}(\mathrm{avars}(t_1, t_2)) = \sigma_1,$
$\mathrm{mgu}(\sigma_1(\mathrm{type}(t_1) \equiv \mathrm{type}(t_2))) = \sigma_2$

[eqn2] $\mathrm{eqn}(t_1, t_2) = \langle \bot, t_1 \equiv t_2\rangle$ **otherwise**

[we-eqns-0] $\mathrm{We}(\Gamma)[\![]\!] =$

[we-eqns-n] $\mathrm{We}(\Gamma)[\![\varphi_1^+; \varphi_2^+]\!] = \mathrm{We}(\Gamma)[\![\varphi_1^+]\!] +\!\!+ \mathrm{We}(\Gamma)[\![\varphi_2^+]\!]$

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 $\Phi \gtrdot t'$ 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 9 (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 *(*$\text{spine}(t') = \text{spine}(t)$*) such that* $\Gamma \vdash_{\text{term}} t'$ *and if* $\Phi = \text{Wt}(\Gamma)[\![t]\!]$*, then* $\Phi \gtrdot t'$ *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 10 *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 11 (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 12 *Type assignment is idempotent, i.e.,*

$$\bigcup_{t' \in \text{Wt}(\Gamma)[\![t]\!]} \text{Wt}(\Gamma)[\![t']\!] = \text{Wt}(\Gamma)[\![t]\!].$$

7.6 Disambiguation and Confluence

We saw in Section 7.2 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 Section 5 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 lhs but completely unrelated rhss. 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` on the rhss force the right types in the lhss. 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 lhs of an equation. An example is the definition of the generalization of `zip` to generalized products, 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 1. Another example are 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 lhs and rhs 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);
```

7.7 Typechecking (MLS-TC)

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.

imports MLS-TA[7.5] MLS-NWF[7.3]
exports
  context-free syntax
    tc "⟦" MLS "⟧" $\rightarrow$ EBool

tc "(" MLS ")" "⟦" Term "⟧" → EBool
tc "(" MLS ")" "⟦" Eqs "⟧" → EBool

equations

[tc-mspec] $$\mathrm{tc}[\![\Gamma]\!] \;=\; \vdash_{\mathrm{mls}} \mathrm{Wm}[\![\Gamma]\!]$$

[tc-term] $$\frac{\mathrm{Wm}[\![\Gamma]\!] = \Gamma',\;\; \mathrm{Wt}(\Gamma')[\![t]\!] = \{t';\, t^*\}}{\mathrm{tc}(\Gamma)[\![t]\!] \;=\; \vdash_{\mathrm{mls}} \Gamma' \leadsto \Gamma' \vdash_{\mathrm{term}} t'}$$

[tc-eqns] $$\frac{\mathrm{Wm}[\![\Gamma]\!] = \Gamma'}{\mathrm{tc}(\Gamma)[\![\mathcal{E}]\!] \;=\; \vdash_{\mathrm{mls}} \Gamma' \leadsto \Gamma' \vdash_{\mathrm{eqs}} \mathrm{We}(\Gamma')[\![\mathcal{E}]\!]}$$

7.8 Typechecking Modular Specifications (MMLS-TC)

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.

imports MMLS$^{6.3}$ MLS-TC$^{7.7}$
exports
context-free syntax
tc "⟦" Modules "⟧" → EBool
tc1 "⟦" Modules "⟧" → EBool
equations

[tc-mods] $\mathrm{tc}[\![M^*]\!] = \mathrm{tc1}[\![\mathrm{flat}(M^*)]\!]$
[tc1-mods1] $\mathrm{tc1}[\![]\!] = \top$
[tc1-mods2] $\mathrm{tc1}[\![\mathrm{module}\, f\, \Gamma;]\!] = \text{errors in module "} f \text{"} :: \mathrm{tc}[\![\Gamma]\!]$
[tc1-mods1] $\mathrm{tc1}[\![M_1^+\ M_2^+]\!] = \mathrm{tc1}[\![M_1^+]\!] \wedge \mathrm{tc1}[\![M_2^+]\!]$

This concludes the specification of the syntax, semantics and typechecking of modular multi-level specifications.

8 Discussion and Concluding Remarks

8.1 Related Work

In Section 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 (1996) 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. Chapter 2 of van Deursen *et al.* (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).

8.2 Extensions

The formalism MLS presented in this chapter is a sophisticated specification formalism for 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.

Grammars as Signatures The motivation for this work is to extend 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 Visser (1995a), 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 (Klint and Visser, 1994).

In Visser (1995b) 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.

In this chapter 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. It seems possible to generalize MLS to a multi-level grammar formalism with a decidable parsing problem by not allowing chain and empty productions

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 like `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))`, ... 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.

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 Section 5.11 the generalization of the `zip` function to arbitrary products of lists is defined by means of functions at the level of types (the map function `(*)`). In Section 5.12 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 Section 5.11. 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 solution. It is an interesting question whether there exists a union of these solutions such that many cases of type equations can be dealt with more generically.

Modules The formalism has a rudimentary modularization scheme based on syntactic inclusion, i.e., imports are expanded before typechecking. How difficult is it 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?

8.3 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.

Appendices

A Library Modules

In this section several modules of common data types are presented.

A.1 Layout

exports

lexical syntax

`[␣\t\n]`	$\rightarrow$	LAYOUT
“%%”~`[\n]`*	$\rightarrow$	LAYOUT
“%”~`[%\n]`*“%”	$\rightarrow$	LAYOUT
“−−”~`[\n]`*	$\rightarrow$	LAYOUT
~`[*]`	$\rightarrow$	Aux
“*”+~[*/]	$\rightarrow$	Aux
“/*”Aux*“*”+“/”	$\rightarrow$	LAYOUT

A.2 Booleans

imports Layout$^{A.1}$

exports

sorts Bool

context-free syntax

“$\top$”	$\rightarrow$	Bool	
“$\bot$”	$\rightarrow$	Bool	
“$\neg$” Bool	$\rightarrow$	Bool	
Bool “$\wedge$” Bool	$\rightarrow$	Bool	{**assoc**}
Bool “$\vee$” Bool	$\rightarrow$	Bool	{**assoc**}
“(” Bool “)”	$\rightarrow$	Bool	{**bracket**}

priorities

“$\neg$”Bool $\rightarrow$ Bool $>$ Bool “$\wedge$”Bool $\rightarrow$ Bool $>$ Bool “$\vee$”Bool $\rightarrow$ Bool

variables

$[b][\mathit{0}\text{-}\mathit{9}']* \rightarrow$ Bool

equations

[conj-1]	$\top \wedge b \;=\; b$	[disj-1]	$\top \vee b \;=\; \top$	[neg-1]	$\neg\,\top \;=\; \bot$
[conj-2]	$\bot \wedge b \;=\; \bot$	[disj-2]	$\bot \vee b \;=\; b$	[neg-2]	$\neg\,\bot \;=\; \top$

A.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
```

imports Layout$^{A.1}$ Booleans$^{A.2}$
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

[e-assoc1] $(e_1; e_2); e_3 = e_1; e_2; e_3$
[e-assoc1] $(e_1 : e_2) : e_3 = e_1 : (e_2; e_3)$
[if-t] if $\top$ then e_1 else $e_2 = e_1$
[if-f] 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 $\wedge$, $\leadsto$ and ::. The operator $\wedge$ is a symmetric conjunction that yields $\top$ if both arguments do and otherwise the conjunction of the errors. The operator $\leadsto$ is an assymetric conjunction that prefers the error in its first argument discarding the error in 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 “$\leadsto$” 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 “$\leadsto$”EBool $\rightarrow$ EBool $>$ EBool “$\wedge$”EBool $\rightarrow$ EBool
  variables
    “e”[0-$9'$]$*$ $\rightarrow$ Error
    “eb”[0-$9'$]$*$ $\rightarrow$ EBool

equations

[conj1]	$\top \wedge eb = eb$
[conj2]	$eb \wedge \top = eb$
[conj4]	$e_1 \wedge e_2 = e_1; e_2$
[kill1]	$\top \leadsto eb = eb$
[kill2]	$e \leadsto eb = e$
[blck]	$eb :: \top = \top$
[blck]	$\top :: eb = eb$
[blck]	$e_1 :: e_2 = e_1 : e_2$
[if-t]	$\text{if } \top \text{ then } eb_1 \text{ else } eb_2 = eb_1$
[if-f]	$\text{if } \bot \text{ then } eb_1 \text{ else } eb_2 = eb_2$

A.4 Naturals

imports Booleans$^{A.2}$
exports
  sorts Nat
  lexical syntax
    [0-9]+ $\rightarrow$ Nat
  context-free syntax

succ(Nat)	$\rightarrow$ Nat	
pred(Nat)	$\rightarrow$ Nat	
Nat “+” Nat	$\rightarrow$ Nat	{**left**}
max(Nat, Nat)	$\rightarrow$ 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 Term Utilities

In this section we define several data types and operations on terms.

B.1 Binary Operators

imports Types$^{3.2}$ Terms$^{2.1}$
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
"⊕"[*0-9′*]∗ → BinOp

equations

[bin1] $t_1 \oplus t_2 = (\oplus)\ (t_1,\, t_2)$
[bin2] $t_1\ .t_2.\ t_3 = t_2\ (t_1,\, t_3)$
[list1] $[t_1,\, t_2] = t_1 :: [t_2]$
[list2] $[t] = t :: [\,]$ **otherwise**
[list1] $\langle t_1,\, t_2\rangle = t_1 \mathbin{\hat{}} \langle t_2\rangle$
[list2] $\langle t\rangle = t \mathbin{\hat{}} \langle\,\rangle$ **otherwise**

B.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
```

imports Error-Booleans$^{A.3}$ OLS$^{3.4}$
exports
context-free syntax

```
    “\”” Term “\”” not a well-formed sort declaration → Error
    sort “\”” Term “\”” not declared                   → Error
    “\”” Term “\”” not a well-formed sort              → Error
    sort “\”” Term “\”” matches no sort declaration   → Error

    function “\”” Term “\”” multiply declared          → Error
    variable “\”” Term “\”” multiply declared          → Error
    function “\”” Term “\”” not declared               → Error
    function “\”” Term “\””
    with type “\”” Term “\”” not declared              → Error
    variable “\”” Term “\”” not declared               → Error

    term “\”” Term “\”” not well-formed                → Error
    pair “\”” Term “\”” not well-formed                → Error
    application “\”” Term “\”” not well-formed         → Error
    product “\”” Term “\”” not well-formed             → Error
    arrow “\”” Term “\”” not well-formed               → Error
    annotation of “\”” Term “\””
    with “\”” Term “\”” not well-formed                → Error

    “\”” Term “\”” is not a function                   → Error
    type of argument “\”” Term “\””
    does not match type of domain “\”” Term “\””       → 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.3 Term Functions

The sort TermToTerm 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.

imports Terms$^{2.1}$ Booleans$^{A.2}$ Types$^{3.2}$
exports
  sorts TermToTerm
  context-free syntax

```
    TermToTerm "(" Term ")"            → Term
    "id"                               → TermToTerm
    TermToTerm "∘" TermToTerm          → TermToTerm  {assoc}
    "if" Bool "then" Term "else" Term  → Term
    TermToTerm "∗" "(" Terms ")"       → Terms
    eq(Term, Term)                     → Bool
    Term "∈" Terms                     → Bool
```

  variables
    "ρ"[*0-9′*]∗ → TermToTerm

equations

[iden]	$\mathrm{id}(t) = t$
[comp]	$\rho_1 \circ \rho_2(t) = \rho_2(\rho_1(t))$
[id-comp]	$\mathrm{id} \circ \rho = \rho$
[id-comp]	$\rho \circ \mathrm{id} = \rho$
[l-map1]	$\rho{*}() =$
[l-map2]	$\rho{*}(t) = \rho(t)$
[l-map3]	$\rho{*}(t_1^+;\, t_2^+) = \rho{*}(t_1^+) +\!\!+\; \rho{*}(t_2^+)$
[if-t]	$\text{if } \top \text{ then } t_1 \text{ else } t_2 = t_1$
[if-f]	$\text{if } \bot \text{ then } t_1 \text{ else } t_2 = t_2$
[eq1]	$\mathrm{eq}(t, t) = \top$
[eq2]	$\mathrm{eq}(t, t') = \bot$ **otherwise**
[l-member0]	$t \in\ = \bot$

[l-member1] $t \in t' = \mathrm{eq}(t, t')$
[l-member2] $t \in t_1^+; t_2^+ = t \in t_1^+ \vee t \in t_2^+$

B.4 Equation Functions

Map TermToTerm functions over equations and lists of equations.

imports Term-Functions$^{B.3}$ Equations$^{2.2}$
exports
  context-free syntax
    TermToTerm "(" Eq ")" → Eq
    TermToTerm "∗e" "(" Eqs ")" → Eqs
    "if" Bool "then" Eqs "else" Eqs → Eqs
equations

[map-eq] $\rho(t_1 \equiv t_2) = \rho(t_1) \equiv \rho(t_2)$
[map-eqs] $\rho{*}e(\varphi^*) =$ **when** $\varphi^* =$
[map-eqs] $\rho{*}e(\varphi; \varphi^*) = \rho(\varphi) +\!\!+ \rho{*}e(\varphi^*)$
[ift] if $\top$ then $\mathcal{E}_1$ else $\mathcal{E}_2 = \mathcal{E}_1$
[ift] if $\bot$ then $\mathcal{E}_1$ else $\mathcal{E}_2 = \mathcal{E}_2$

B.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 (∪), intersection (∩), difference (/), emptiness ('empty'), membership (∈) and subset (⊆). The Cartesian product × yields the set of pairs of the elements of two sets.

imports Term-Functions$^{B.3}$ Terms$^{2.1}$ Booleans$^{A.2}$
exports
  sorts TermSet
  context-free syntax
    "{" Terms "}" → TermSet
    TermSet "∪" TermSet → TermSet {**left**}
    TermSet "∩" TermSet → TermSet {**left**}
    TermSet "/" TermSet → TermSet {**left**}
    TermSet "×" TermSet → TermSet {**right**}
    TermToTerm "∗" "(" TermSet ")" → TermSet
    "if" Bool "then" TermSet "else" TermSet → TermSet
    trms(TermSet) → Terms
    "(" TermSet ")" → 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

[s-double]	$\{t_1^*;\, t;\, t_2^*;\, t;\, t_3^*\} = \{t_1^*;\, t;\, t_2^*;\, t_3^*\}$
[s-union]	$\{t_1^*\} \cup \{t_2^*\} = \{t_1^*;\, t_2^*\}$
[s-istc1]	$\{\} \cap \Phi = \{\}$
[s-isct2]	$\{t_1^+;\, t_2^+\} \cap \Phi = \{t_1^+\} \cap \Phi \cup \{t_2^+\} \cap \Phi$
[s-isct3]	$\{t\} \cap \Phi = \text{if } t \in \Phi \text{ then } \{t\} \text{ else } \{\}$
[s-diff1]	$\{\} \,/\, \Phi = \{\}$
[s-diff2]	$\{t_1^+;\, t_2^+\} \,/\, \Phi = \{t_1^+\} \,/\, \Phi \cup \{t_2^+\} \,/\, \Phi$
[s-diff3]	$\{t\} \,/\, \Phi = \text{if } t \in \Phi \text{ then } \{\} \text{ else } \{t\}$
[s-prd5]	$\{t_1\} \times \{t_2\} = \{t_1,\, t_2\}$
[s-prd1]	$\{\} \times \Phi = \{\}$
[s-prd3]	$\{t_1^+;\, t_2^+\} \times \Phi = \{t_1^+\} \times \Phi \cup \{t_2^+\} \times \Phi$
[s-prd2]	$\Phi \times \{\} = \{\}$
[s-prd4]	$\Phi \times \{t_1^+;\, t_2^+\} = \Phi \times \{t_1^+\} \cup \Phi \times \{t_2^+\}$
[s-map]	$\rho*(\{ts\}) = \{\rho*(ts)\}$
[s-ift]	$\text{if } \top \text{ then } \Phi_1 \text{ else } \Phi_2 = \Phi_1$
[s-iff]	$\text{if } \bot \text{ then } \Phi_1 \text{ else } \Phi_2 = \Phi_2$
[s-trms]	$\text{trms}(\{ts\}) = ts$
[s-empty1]	$\text{empty}(\{\}) = \top$
[s-empty2]	$\text{empty}(\{t^+\}) = \bot$
[s-member]	$t \in \{ts\} = t \in ts$
[s-subset1]	$\{\} \subseteq \Phi = \top$
[s-subset2]	$\{t\} \subseteq \Phi = t \in \Phi$
[s-subset3]	$\{t_1^+;\, t_2^+\} \subseteq \Phi = \{t_1^+\} \subseteq \Phi \wedge \{t_2^+\} \subseteq \Phi$

B.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.

imports Term-Sets$^{B.5}$
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

[vs-fun]	$vs(f) = \{\}$
[vs-nil]	$vs(\mathrm{nil}) = \{\}$
[vs-top]	$vs(\mathrm{top}) = \{\}$
[vs-pr]	$vs(t_1, t_2) = vs(t_1) \cup vs(t_2)$
[vs-app]	$vs(t_1\ t_2) = vs(t_1) \cup vs(t_2)$
[vs-prd]	$vs(t_1 \times t_2) = vs(t_1) \cup vs(t_2)$
[vs-arr]	$vs(t_1 \to t_2) = vs(t_1) \cup vs(t_2)$
[vs-set]	$vs(\{ts\}) = vs*(\ ts)$
[vs-trms]	$vs*(\ ) = \{\}$
[vs-trms-1]	$vs*(\ t) = vs(t)$
[vs-trms-n]	$vs*(\ t_1^+; t_2^+) = vs*(\ t_1^+) \cup vs*(\ t_2^+)$

[vars-var]	$\mathrm{vars}(x) = \{x\}$
[vars-ann-other]	$\mathrm{vars}(t : \tau) = \mathrm{vars}(t) \cup \mathrm{vars}(\tau)$
[ovars-var]	$\mathrm{ovars}(x) = \{x\}$
[ovars-ann]	$\mathrm{ovars}(t : \tau) = \mathrm{ovars}(t)$
[tvars-var]	$\mathrm{tvars}(x) = \{\}$
[tvars-ann]	$\mathrm{tvars}(t : \tau) = \mathrm{tvars}(t) \cup \mathrm{vars}(\tau)$
[avars-ann]	$\mathrm{avars}(x) = \{\}$
[avars-ann]	$\mathrm{avars}(x : \tau) = \{x : \tau\}$
[avars-ann]	$\mathrm{avars}(t : \tau) = \mathrm{avars}(t)$ **otherwise**

B.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 [s-var-*i*].

imports Term-Functions$^{B.3}$ Terms$^{2.1}$ Types$^{3.2}$
exports
  sorts ASubst Subst
  context-free syntax

Var ":=" Term	→ ASubst	
"[" ASubst∗ "]"	→ Subst	
Subst	→ TermToTerm	
"⇓"(TermToTerm)	→ Subst	
Subst "+" Subst	→ Subst	{**assoc**}
"(" Subst ")"	→ Subst	{**bracket**}

  variables

"*as*"[*0-9′*]∗	→ ASubst
"*as*" "∗"[*0-9′*]∗	→ ASubst∗
"*as*" "+"[*0-9′*]∗	→ ASubst+
"*σ*"[*0-9′*]∗	→ Subst

equations

[s-var-1]	$[x := t\ as^*](x) = t$
[s-var-2]	$[y := t\ as^*](x) = [as^*](x)$ **when** $\mathrm{eq}(x, y) = \bot$
[s-var-3]	$[](x) = x$
[s-fun]	$\sigma(f) = f$
[s-nil]	$\sigma(\mathrm{nil}) = \mathrm{nil}$
[s-top]	$\sigma(\mathrm{top}) = \mathrm{top}$
[s-pr]	$\sigma(t_1, t_2) = \sigma(t_1), \sigma(t_2)$
[s-app]	$\sigma(t_1\ t_2) = \sigma(t_1)\ \sigma(t_2)$
[s-prd]	$\sigma(t \times \tau) = \sigma(t) \times \sigma(\tau)$
[s-arr]	$\sigma(t \to \tau) = \sigma(t) \to \sigma(\tau)$
[s-ann]	$\sigma(t : \tau) = \sigma(t) : \sigma(\tau)$
[s-back]	$\Downarrow(\sigma) = \sigma$
[s-back]	$\Downarrow(\mathrm{id}) = []$

[s-union] $[as_1^*] + [as_2^*] = [as_1^* \; as_2^*]$
[s-empty] $[] \circ \sigma = \sigma$
[s-empty] $\sigma \circ [] = \sigma$
[s-comp-n] $\sigma \circ [x := t \; as^*] = [x := \sigma(t)] + \sigma \circ [as^*]$

Failure Substitutions A failure substitution is a substitution or the value $\bot$ (fail), which denotes failure for partial functions producing substitutions like matching and unification. The operation $+_\bot$ 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 are undefined.

sorts $\text{Subst}_\bot$
context-free syntax
Subst $\rightarrow \text{Subst}_\bot$
"$\bot$" $\rightarrow \text{Subst}_\bot$
$\text{Subst}_\bot$ "$+_\bot$" $\text{Subst}_\bot$ $\rightarrow \text{Subst}_\bot$ {**non-assoc**}
$\text{Subst}_\bot$ "$\oplus$" $\text{Subst}_\bot$ $\rightarrow \text{Subst}_\bot$ {**non-assoc**}
$\text{Subst}_\bot$ "$\circ_\bot$" $\text{Subst}_\bot$ $\rightarrow \text{Subst}_\bot$ {**non-assoc**}
"if" Bool "then" $\text{Subst}_\bot$ "else" $\text{Subst}_\bot$ $\rightarrow \text{Subst}_\bot$
"fail?"($\text{Subst}_\bot$) $\rightarrow$ Bool
"$\Downarrow_\bot$"($\text{Subst}_\bot$) $\rightarrow$ Subst
"(" $\text{Subst}_\bot$ ")" $\rightarrow \text{Subst}_\bot$ {**bracket**}
variables
"$\sigma_\bot$"[$'$]$*$ $\rightarrow \text{Subst}_\bot$
priorities
{**non-assoc**: $\text{Subst}_\bot$ "$+_\bot$"$\text{Subst}_\bot \rightarrow \text{Subst}_\bot$, $\text{Subst}_\bot$ "$\oplus$"$\text{Subst}_\bot \rightarrow \text{Subst}_\bot$, $\text{Subst}_\bot$ "$\circ_\bot$"$\text{Subst}_\bot \rightarrow \text{Subst}_\bot$} $>$ "if"Bool "then"$\text{Subst}_\bot$ "else"$\text{Subst}_\bot \rightarrow \text{Subst}_\bot$

equations

[fs-ift] $\text{if } \top \text{ then } \sigma_\bot \text{ else } \sigma_\bot{}' = \sigma_\bot$
[fs-iff] $\text{if } \bot \text{ then } \sigma_\bot \text{ else } \sigma_\bot{}' = \sigma_\bot{}'$
[comp1] $\sigma_1 +_\bot \sigma_2 = \sigma_1 + \sigma_2$
[comp0] $\sigma_\bot +_\bot \bot = \bot$
[comp0] $\bot +_\bot \sigma_\bot = \bot$
[comp1] $\bot \oplus \sigma_\bot = \bot$
[comp2] $\sigma_\bot \oplus \bot = \bot$
[comp3] $[] \oplus \sigma_\bot = \sigma_\bot$
[comp4] $\sigma_\bot \oplus [] = \sigma_\bot$
[comp5] $[x := t \; as^*] \oplus \sigma = \text{if } \mathrm{eq}(t', x) \vee \mathrm{eq}(t', t)$
$\text{then } [x := t] +_\bot ([as^*] \oplus \sigma)$
$\text{else } \bot$
when $\sigma(x) = t'$

[comp]	$\bot \circ_\bot \sigma_\bot = \bot$
[comp]	$\sigma_\bot \circ_\bot \bot = \bot$
[comp]	$\sigma_1 \circ_\bot \sigma_2 = \Downarrow(\sigma_1 \circ \sigma_2)$
[fail-f]	$\text{fail?}(\sigma) = \bot$
[fail-t]	$\text{fail?}(\bot) = \top$
[back-fs]	$\Downarrow_\bot(\sigma) = \sigma$

B.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.

imports Substitution$^{B.7}$ Term-Sets$^{B.5}$
exports
context-free syntax

Terms ":=" Terms	$\to$ Subst$_\bot$
Term "$\geq$" Term	$\to$ Bool
Term "$>$" Term	$\to$ Bool
Term "$\doteq$" Term	$\to$ Bool
TermSet "$\geq$" Term	$\to$ Bool

equations

[m-var]	$x := t = [x := t]$
[m-fun]	$t := t = []$
[m-pr]	$t_1, t_2 := t_3, t_4 = t_1; t_2 := t_3; t_4$
[m-app]	$t_1\ t_2 := t_3\ t_4 = t_1; t_2 := t_3; t_4$
[m-prd]	$t_1 \times t_2 := t_3 \times t_4 = t_1; t_2 := t_3; t_4$
[m-pr]	$t_1 \to t_2 := t_3 \to t_4 = t_1; t_2 := t_3; t_4$
[m-prd]	$t_1 : t_2 := t_3 : t_4 = t_1; t_2 := t_3; t_4$
[m-trms-0]	$:= \; = []$
[m-trms-n]	$t_1; t_1^+ := t_2; t_2^+ = t_1 := t_2 \oplus t_1^+ := t_2^+$
[no-match]	$t := t' = \bot$ **otherwise**
[m-geq]	$t_1 \geq t_2 = \neg\, \text{fail?}(t_1 := t_2)$
[m-gtr]	$t_1 > t_2 = t_1 \geq t_2 \wedge \neg\, t_2 \geq t_1$
[m-eq]	$t_1 \doteq t_2 = t_1 \geq t_1 \wedge t_2 \geq t_1$
[m-set-0]	$\{\} \geq t = \bot$
[m-set-1t]	$\{t'\} \geq t = t' \geq t$
[m-set-n]	$\{t_1^+; t_2^+\} \geq t = \{t_1^+\} \geq t \vee \{t_2^+\} \geq t$

B.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.

imports Variables$^{B.6}$ Substitution$^{B.7}$ Equation-Functions$^{B.4}$
exports
context-free syntax

$$\begin{array}{ll} \text{mgu(Eqs)} & \rightarrow \text{Subst}_\perp \\ \text{Term ``}\overset{?}{=}\text{'' Term} & \rightarrow \text{Bool} \end{array}$$

equations

$$\begin{array}{lrcl}
[\text{u-refl}] & \text{mgu}(t \equiv t) & = & [] \\
[\text{u-var}] & \text{mgu}(x \equiv t) & = & [x := t] \quad \textbf{when} \ \ x \in \text{vars}(t) = \perp \\
[\text{u-var'}] & \text{mgu}(t \equiv x) & = & [x := t] \quad \textbf{when} \ \ x \in \text{vars}(t) = \perp \\
[\text{u-pr}] & \text{mgu}(t_1, t_2 \equiv t_3, t_4) & = & \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4) \\
[\text{u-app}] & \text{mgu}(t_1\ t_2 \equiv t_3\ t_4) & = & \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4) \\
[\text{u-prd}] & \text{mgu}(t_1 \times t_2 \equiv t_3 \times t_4) & = & \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4) \\
[\text{u-arr}] & \text{mgu}(t_1 \rightarrow t_2 \equiv t_3 \rightarrow t_4) & = & \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4) \\
[\text{u-ann}] & \text{mgu}(t_1 : t_2 \equiv t_3 : t_4) & = & \text{mgu}(t_1 \equiv t_3;\, t_2 \equiv t_4) \\
[\text{u-es-0}] & \text{mgu}() & = & [] \\
[\text{u-es-n}] & \text{mgu}(\varphi_1^+;\, \varphi_2^+) & = & \text{mgu}(\Downarrow_\perp(\sigma_\perp) * e(\varphi_2^+)) \circ_\perp \sigma_\perp \\
 & & & \textbf{when} \ \ \text{mgu}(\varphi_1^+) = \sigma_\perp \\
[\text{u-fail}] & \text{mgu}(\mathcal{E}) & = & \perp \quad \textbf{otherwise} \\
[\text{ueq}] & t_1 \overset{?}{=} t_2 & = & \neg\ \text{fail?}(\text{mgu}(t_1 \equiv t_2))
\end{array}$$

B.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 much like the functions in Chapter 1 of van Deursen *et al.* (1996) to rename variables in λ expressions. 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.

imports Variables$^{B.6}$ Substitution$^{B.7}$

exports
  context-free syntax

prime(Var)	→ Var
deprime(Var)	→ Var
base(Var)	→ Var
get-fresh(Var, TermSet)	→ Term
rn TermSet "[" TermSet "]"	→ Subst
rn Term "[" Term "]"	→ Term
add(Var, TermSet)	→ TermSet

hiddens
  variables
    "*c*+"[*0-9′*]∗ → CHAR+

equations

[prm-var] $\text{prime}(\text{var}(c^+)) = \text{var}(c^+\ \texttt{"'"})$
[dprm-var-1] $\text{deprime}(\text{var}(c^+\ \texttt{"'"})) = \text{deprime}(\text{var}(c^+))$
[dprm-var-2] $\text{deprime}(x) = x$ **otherwise**

The function 'base' takes off all trailing digits and primes of a variable. The equations for the function 'base' are not shown.

[add1] $\text{add}(\text{var}(c_1^+), \{\text{var}(c_2^+)\}) = \{\text{var}(c_1^+\ c_2^+)\}$
[add2] $\text{add}(x, \{\}) = \{x\}$
[add3] $\text{add}(x, \{t_1^+; t_2^+\}) = \text{add}(x, \{t_1^+\}) \cup \text{add}(x, \{t_2^+\})$
[add] $\text{add}(x, \{t\}) = \{x\}$ **otherwise**

[f-1] $\text{get-fresh}(x, \Phi) = \text{if } x \in \Phi \text{ then get-fresh}(\text{prime}(x), \Phi) \text{ else } x$

[f-3] $\text{rn } \{\}[\Phi] = []$
[f-4] $\text{rn } \{x;\ t^*\}[\Phi] = [x := y] + \text{rn } \{t^*\}[\{y\} \cup \Phi]$
when $\text{get-fresh}(\text{deprime}(x), \Phi) = y$

Rename a term with respect to the variables in another term.

[f-5] $\text{rn } t_1[t_2] = \text{rn } \Phi_1 \cap \Phi_2[\Phi_2](t_1)$
when $\text{vars}(t_1) = \Phi_1,\ \text{vars}(t_2) = \Phi_2$

References

Bergstra, J. and Sellink, M. (1996). Sequential data algebra primitives. Technical Report P9602, University of Amsterdam, Programming Research Group. Available by anonymous ftp from ftp.fwi.uva.nl, file pub/programming-research/reports/1996/P9602.ps.Z.

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.

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.

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.

Cardelli, L. (1993). Typeful programming. SRC Research Report 45, May 24, 1989. Revised January 1, 1993, Digital Equipment Corporation.

Cardelli, L. (1996). Type systems. Draft. To appear in CRC Handbook of Computer Science and Engineering.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, **17**(4), 471–522.

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.

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.

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.

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, *Proc. First Int. 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. Most recent version available at ftp://ftp.cwi.nl/pub/gipe/reports/SDFManual.ps.Z.

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.

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).* ACM SIGPLAN Notices.

Jones, M. P. (1992). A theory of qualified types. In B. Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, New York.

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).

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. Also as TR P9426, Programming Research Group, University of Amsterdam, ftp://ftp.fwi.uva.nl /pub/programming-research/ reports/1994/ P9426.ps.Z.

Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, **4**, 258–282.

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.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**(3), 348–375348–375.

Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML*. MIT Press.

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. (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.

Nazareth, D. (1995). *A Polymorphic Sort System for Axiomatic Specification Languages*. Ph.D. thesis, Technische Universität München. Technical Report TUM-I9515, http://www4.informatik.tu-muenchen.de/~nazareth/phd.html.

Nipkow, T. and Prehofer, C. (1995). Type reconstruction for type classes. *Journal of Functional Programming*, **5**(2), 201–224.

Poigné, A. (1986). On specifications, theories, and models with higher types. *Information and Control*, **68**, 1–46.

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.

van Deursen, A., Heering, J., and Klint, P., editors (1996). *Language Prototyping. An Algebraic Specification Approach.* AMAST Series in Computing. World Scientific Publishing Inc., Singapore.

Vigna, S. (1995). Specifying IMP(G) using ASF+SDF: A case study. In M. G. J. v. d. 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. (1995a). A family of syntax definition formalisms. In M. G. J. v. d. 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. (1995b). 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.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *16'th Symposium on Principles of Programming Languages*, Austin, Texas. ACM Press.

Walters, H. and Kamperman, J. (1996). Epic 1.0 (unconditional), an equational programming language. Technical Report CS-R9604, CWI. Available as http://www.cwi.nl/epic/articles/epic10.ps.

Wechler, W. (1992). *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag.

Index

[P9502] J.A. Bergstra and P. Klint. *The discrete time ToolBus.*

[P9501] J.A. Hillebrand and H.P. Korver. *A well-formedness checker for μCRL.*

[P9426] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.*

[P9425] B. Diertens and A. Ponse. *New features in PSF II: iteration and nesting.*

[P9424] M.A. Bezem and A. Ponse. *Two finite specifications of a queue.*

[P9423] J.J. van Wamel. *Process algebra with language matching.*

[P9422] R.N. Bol, L.H. Oei J.W.C. Koorn, and S.F.M. van Vlijmen. *Syntax and static semantics of the interlocking design and application language.*

[P9421] J.A. Bergstra and A. Ponse. *Frame algebra with synchronous communication.*

[P9420] M.G.J. van den Brand and E. Visser. *From Box to TeX: An algebraic approach to the construction of documentation tools.*

[P9419] J.C.M. Baeten, J.A. Bergstra, and Gh. Stefanescu. *Process algebra with feedback.*

[P9418] L.H. Oei. *Pruning the search tree of interlocking design and application language operational semantics.*

[P9417] B. Diertens. *New features in PSF I: interrupts, disrupts, and priorities.*

[P9416] S.M. Üsküdarlı. *Generating visual editors for formally specified languages.*

[P9415] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. *Control and data transfer in the distributed editor of the ASF+SDF meta-environment.*

[P9414] M.G.J. van den Brand and C. Groza. *The algebraic specification of annotated abstract syntax trees.*

[P9413] A. Ponse, C. Verhoef, and S.F.M. van Vlijmen (editors). *Workshop on Algebra of Communicating Processes May 16-17, 1994 Utrecht University.*

[P9218b] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. *Axiomatizing probabilistic processes: ACP with generative probabilities (revised version of P9218).*

[P9412] C. Groza. *An experiment in implementing process algebra specifications in a procedural language.*

[P9411] M.J. Koens and L.H. Oei. *A real time muCRL specification of a system for traffic regulation at signalized intersections.*