THE ESSENCE OF STRATEGIC PROGRAMMING

An inquiry into trans-paradigmatic genericity

Ralf Lämmel
Vrije Universiteit
ralf@cs.vu.nl

Eelco Visser
Universiteit Utrecht
visser@acm.org

Joost Visser
CWI Amsterdam
Joost.Visser@cwi.nl

Abstract Strategic programming is an idiom for generic programming where the concept of a *strategy* plays a central role. A strategy is a generic, data-processing action. Strategies are first-class citizens as witnessed by a combinator style. Two important characteristics of strategies are that they can traverse into compound data, and that they can be customized by type-specific actions. We provide a general definition of strategic programming, and we demonstrate how this idiom can be realized inside several programming language paradigms.

Keywords: Strategies, traversal, generic programming, combinators

March 4, 2002
`http://www.program-transformation.org/Transform/TheEssenceOfStrategicProgramming`

1. Introduction

In various areas of programming, control over the application of basic data-processing actions is needed. In standard programming paradigms it is usual that control and basic actions become intertwined, precluding reusability of control patterns and basic actions. Strategic programming is a programming idiom for untangling concerns such as computation and control in which basic computations can be reused across different control constructs, and patterns of control can be defined generically as first-class entities.

Evaluation strategies: explicit versus implicit In rule-based programming (e.g., algebraic term rewriting, expert systems, abstract state machines), the programmer supplies a set of computation rules and relies on the system's built-in evaluation strategy (e.g., leftmost-innermost) for applying these rules to input data. The order in which the rules fire, and at which parts of the input data, is not specified by the programmer. In contrast, by explicitly programming (recursive) function calls, a functional programmer exhaustively specifies the order and location in which the functions are to be evaluated. In this approach, the evaluation strategy is explicit in the function calls, but heavily entangled with the other functionality.

These contrasting approaches trade lack of control over the evaluation strategy against tangling, and *vice versa*. The idiom of *strategic programming* allows one to have his cake and eat it too: complete control over the evaluation strategy can be obtained without tangling computation and control.

Traversal strategies: types versus reuse In several application areas, one can observe a need for traversal control. For example, in program transformation, the ordering of atomic transformations, checking for their applicability, the repetitive application of them, and their restriction to particular parts of the input program are essential to guarantee correctness and termination of many transformations. Similarly, in program analysis and understanding, control is needed to traverse call graphs, data-flow graphs, or other kinds of highly structured source models. Several program analyses require work-list algorithms that schedule element processing in, e.g., breadth-first or best-first fashion. In the field of document processing a similar need for control can be observed. Specific elements in documents need to be identified, retrieved, and converted. This requires control over the order, scope, nesting level, or path along which elements are visited.

When traversing highly structured data, such as parse trees, source models, or documents, the programmer usually trades type-safety for reusability. To obtain type-safety, the data must be heterogenous: each kind of tree, graph, or document element is assigned a specific type. But when the data structure is heterogenous, access to and traversal over its subelements will involve dealing with many specific types in specific ways. If no generic access or traversal is available, this approach implies lengthy traversal code which cannot be reused for differently typed data structures. The most obvious way to recover concision and reusability is to abandon type-safety and instead to employ homogenous, universal data representations. This dilemma occurs regardless of the fact if we assume strong static types, or if we resort to more liberal, even dynamic, well-formedness checking of heterogenous data structures. Existing notions of generic programming do not immediately remedy this dilemma of type-safety vs. reusability because, often, generic entities are not first-class and type-specific customization is cumbersome.

As we will see, this dilemma can be solved with strategic programming, because it caters for generic access to the subcomponents of heterogeneously typed data structures.

Strategic programming The need for control over evaluation and traversal strategies in different areas of computing, has led to a number of innovations, including rewriting strategies for tactics and tacticals in theorem proving [58], generalized folds [53], visitors [20], propagation patterns [47], and traversal functions [10]. In previous work [49, 64, 38, 44, 65] we have built on these innovations, culminating in a programming style which we have termed *strategic programming*. We have worked out detailed realizations of this style in several programming paradigms.

In this paper we generalize over these various realizations, providing a general characterization of strategic programming, and its central notion of a *strategy*. We reassess and compare several realizations in light of this general characterization.

Outline We will begin by formulating a definition of strategic programming in Section 2. In Sections 3 to 5, we will discuss realizations of the strategic programming idiom in three programming language paradigms in detail: term rewriting, functional programming, and object-oriented programming. Throughout the paper we will use examples from the area of *grammar engineering*, that is, our strategic programs analyse and transform syntax definitions [60, 14, 37, 43, 46]. A discussion of related work can be found in Section 6. The paper is

concluded in Section 7 with a comparison of the incarnations, a brief review of applications, and pointers to implementations.

2. Strategic Programming

In this section we will explain the notion of strategic programming. We will give defining characteristics of the concept of a *strategy*. These characteristics are embodied by a minimal set of basic strategy combinators. The expressiveness of these combinators is illustrated by the definition of several reusable traversal schemes and other parameterized strategies. Then we make an inventory of the challenges to be met by a strategic programming setting. We conclude with a discussion of the benefits to be expected from adopting strategic programming.

2.1. Ingredients

To define the notion of strategic programming, we need to delineate its principle notion: the concept of a *strategy*. In our opinion, this concept is best described by enumerating its defining characteristics.

Genericity Strategies are generic in the sense that they are applicable to data of any shape (type, sort, class).

Specificity Though generic, strategies provide access to the data structures on which they operate via type-specific operations.

Traversal Strategies enable generic traversal into subcomponents of compound data structures.

Partiality The application of a strategy to a given datum may fail. A mechanism for recovery from failure must be present.

Control Strategies can be used to generically model control over the application of basic computations.

First-class Strategies are first-class citizens in the sense that they can be named, can be passed as arguments, etc. Their first-class status enables a combinator-style of programming.

Together, these defining characteristics define an abstract notion of *strategy* that is not bound to any particular programming language or paradigm. The idiom of strategic programming, then, can be defined simply as program construction with the use of strategies.

Comb.	Signature	Meaning
$\_[\_]$	$\pi \times \tau \to \alpha_\tau$	Type specialization
`adhoc`	$\pi \times \alpha_\tau \to \pi$	$\begin{cases} \texttt{adhoc}(s,r)@t \Rightarrow r(t) & \text{(if } r(t) \text{ well-typed)} \\ \texttt{adhoc}(s,r)@t \Rightarrow s@t & \text{(otherwise)} \end{cases}$
`id`	π	$\texttt{id}@t \Rightarrow t$
`fail`	π	$\texttt{fail}@t \Rightarrow \mathit{undef}$
$\_;\_$	$\pi \times \pi \to \pi$	$(s;s')@t \Rightarrow s'@(s@t)$
$\_<\_+\_$	$\pi\times\pi\times\pi \to \pi$	$\begin{cases} (s<s'+s'')@t \Rightarrow s'@(s@t) & \text{(if } s@t \not\Rightarrow \mathit{undef}) \\ (s<s'+s'')@t \Rightarrow s''@t & \text{(if } s@t \Rightarrow \mathit{undef}) \end{cases}$
`all`	$\pi \to \pi$	$\texttt{all}(s)@f(t_1,\ldots,t_n) \Rightarrow f(s@t_1,\ldots,s@t_n)$
`one`	$\pi \to \pi$	$\texttt{one}(s)@f(\ldots,t_i,\ldots) \Rightarrow f(\ldots,s@t_i,\ldots)$

Figure 1. Basic strategy combinators.

2.2. Basic Combinators

The notion of *strategy* can be made significantly more concrete by prescribing a minimal set of strategy combinators that any strategic programming incarnation must supply. Figure 1 specifies such a set. We define the profiles of the combinators, and we sketch their meaning by indicating their effect when they are applied to data. Application of a data-processing action a to a datum t is written as $a@t$. The meaning of such an application is then indicated by judgements of the form $a@t \Rightarrow t'$. We do not suggest a very formal reading of the table because we aim at a general, language-independent definition of strategies.

Specificity vs. genericity In the figure, we use π to denote the type of strategies (i.e., nullary strategy combinators), and we use α_τ to denote the type of a basic action meant to process data of a specific type τ. In different paradigms, this latter type will be incarnated differently, e.g., as the type of a many-sorted rewrite rule in term rewriting, as the type of a unary monadic function in functional programming, and as the type of a visit method in object-oriented programming. Note that strategies π are generic data-processing actions whereas basic actions α_τ are type-specific.

The first two combinators in the table mediate between the generic and the specific world. $s[\tau]$ turns a strategy s into a basic action to process data of type τ by *type specialization*. The combinator `adhoc` performs *strategy update*. It updates a default strategy s with a basic action r such that the resulting strategy will behave like r on data of r's input type, and like s otherwise (type case). When we indicate the meaning of strategies we leave out type specialization for concision,

but we assume that a strategy is always first type-specialized before it is applied to a term of a specific type. In fact, type specialization is a standard notion required for polymorphic entities, whereas strategy update is an essential contribution of strategic programming.

Combinator style The lower part of Figure 1 defines the basic combinators for control and traversal. The nullary strategy `id` succeeds for any datum and returns its input without change. The strategy `fail` fails for any datum, indicated by the output *undef*. The sequence combinator _; _ applies its two argument strategies in succession. The conditional combinator _<_+_ first attempts application of its first argument strategy. If this application succeeds, the second argument is applied to the intermediate result. Otherwise, the third argument is applied to the original term. The unary combinators `all` and `one` both push their argument strategy one level down into the input datum. The expression $f(t_1, \ldots, t_n)$ does not mean to imply that strategies necessarily work on terms formed by constructor application. It is only meant to indicate that the input datum can in some sense be *compound*, and that its direct components can be retrieved.

As will become evident in subsequent sections, the minimal set of basic combinators need not coincide with the *primitives* of a strategic programming incarnation. Also, depending on the host paradigm, additional basic combinators might be appropriate, and the semantics of each combinator might slightly deviate. Nonetheless, the set of combinators given here provides a concrete guideline (norm, standard) for an incarnation of strategic programming.

2.3. Defined Combinators

The power of our admittedly small set of basic combinators can best be demonstrated with a few examples. Figure 2 shows a list of combinators defined in terms of the basic ones.

The first combinator defines left-biased `choice` in terms of the conditional combinator _<_+_. The next two combinators manipulate the success value of their argument strategy: `not` inverts this value, while `try` recovers from failure via `id` if necessary. The strategy `test(s)` has the same success value as `s`, but ignores the output of `s`. The `repeat` combinator applies its argument strategy until a fixpoint is reached. The `topdown` and `bottomup` combinators apply their argument at the root of the incoming datum, and at all its immediate and non-immediate components. The `topdown` combinator traverses the data in pre-order, while the `bottomup` combinator defines a post-order traversal. The combinators `once-bu` and `while-td` are variations on `bottomup` and `topdown`

Combinator	Definition
`choice`(s, s')	s<`id`+s'
`not`(s)	s<`fail`+`id`
`try`(s)	`choice`$(s,$ `id`$)$
`test`(s)	`not`(s)<`fail`+`id`
`repeat`(s)	`try`$(s;$ `repeat`$(s))$
`topdown`(s)	s; `all`$($`topdown`$(s))$
`bottomup`(s)	`all`$($`bottomup`$(s))$; s
`once-bu`(s)	`one`$($`once-bu`$(s))$<`id`+s
`while-td`(s)	s<`all`$($`while-td`$(s))$+`id`
`naive-innermost`(s)	`repeat`$($`once-bu`$(s))$
`innermost`(s)	`all`$($`innermost`$(s))$; `try`$(s;$ `innermost`$(s))$
`below`(s_b, s_t)	`once-td`$(s_t;$ `once-td`$(s_b))$
`strict-below`(s_b, s_t)	`below`$($`one`$(s_b), s_t)$

Figure 2. Some defined strategy combinators

that apply the argument strategy only to the first component at which it succeeds, or at all components above which it does not fail, respectively. The `naive-innermost` and `innermost` combinators both implement the leftmost innermost evaluation strategy, but the second is more efficient than the first. Finally, `below` and `strict-below` take two argument strategies and apply the first at a component nested inside a component where the second is applied. In the case of `below` these two components are allowed to coincide.

These examples demonstrate all but one of the defining ingredients of strategies: type-specific operations. Forcibly, these operations are highly dependent on the kind of data present in a given programming paradigm. For concrete examples we must therefore beg the reader's patience until subsequent sections. There we will demonstrate how generic algorithms can be implemented as generic defined strategy combinators, which can be instantiated for particular data structures by supplying strategies as arguments that employ type-specific operations.

2.4. Challenges

When realizing the strategic programming idiom in a specific programming paradigm, several challenges await us.

Semantics The semantics of strategies, as sketched in Figure 1 must be mapped onto the semantics of the host paradigm. This involves inhabiting notions such as *basic action*, *datum*, *subcomponent*, and *failure*.

Typing The typing of strategies is a major challenge. The requirement of generic traversal calls, as we will see, for sophisticated typing concepts. This concerns traversal combinators like `all` and `one`. Also, the tension must be resolved between genericity and data access via type-specific operations. This concerns the combinator `adhoc` for strategy update.

Access to host paradigm We want all programming features that are 'native' to the host paradigm to remain available to the programmer when using strategies. For instance, in object-oriented programming, strategies should blend with reference semantics, side-effects, and inheritance. In functional programming they should have value semantics, allow monadic I/O, and be strict or lazy depending on the host language.

Deviation from guideline combinator set As mentioned above, the set of basic combinators of Figure 1 is meant only as a guideline to clarify the concept of strategies. A strategic programming incarnation might offer additional combinators, or variations on the given set of basic combinators. The actual basic combinators offered by an incarnation are either primitive concepts of the incarnation, or the combinators are programmed using an even more fundamental interface supported by the incarnation. This latter option is intriguing because it offers the strategic programmer the possibility to extend the set of basic strategy combinators.

To what extent we have succeeded in meeting these challenges in the various incarnations of strategic programming will become clear in the upcoming sections.

2.5. Benefits

Programming with strategies offers a number of benefits over doing without them.

Separation of concerns Strategies allow one to implement conceptually separable concerns in distinct modules, whilst otherwise they would be entangled in a single code fragment. As a result, these concerns can be understood, reasoned about, developed, tested, maintained, and reused separately. Examples of (categories of) concerns that we encountered include traversal, evaluation, computation, control, accumulation, state, testing.

Reuse Strategies enable reuse in several dimensions. Within an application, a single concern, such as a particular traversal strategy or applicability condition, needs to be implemented only once in a reusable strat-

egy. Across applications, strategies can be reused that capture generic behaviour. Examples are defined combinators for traversal and control, such as `topdown` and `repeat`, but also combinators that capture generic algorithms, e.g., for free variable analysis or refactoring [61, 44, 40].

Robustness Using strategies, each concern can be implemented with explicit reference only to types and operations that are relevant to it. As a result, changes in specific data structures will not unduly affect the implementation of the concern. This isolation from changes in class hierarchy, term signatures, or algebraic data type definitions (depending on the paradigm) diminishes the resistance to change of software systems, and makes them more maintainable.

Concision When programming on heterogeneous data structures, strategic programming can be radically more concise than non-generic programming. Due to the generic access to specific data structures, the strategic programmer does not need to repeatedly implement for each type the behaviour that is conceptually generic to them all. Instead, generic behaviour can be captured in first-class combinators and reused across types.

2.6. Realizations of Strategic Programming

The idiom of strategic programming outlined above is not just a theoretical artifact. It has actually been realized within several programming paradigms and applied in the construction of numerous tools. In the upcoming three sections we will closely examine the realization of the strategic programming idiom in term rewriting (Section 3), functional programming (Section 4), and object-oriented programmming (Section 5). This examination will inhabit the ingredient notions of strategic programming and show interaction with type-specific operations, which is different for each paradigm. It will clarify the contribution that strategic programming can make to the individual paradigms, and the problems encountered when implementing a realization.

Each of these sections follows the same pattern. First we introduce the programming paradigm and illustrate the problems with tangling of logic and control. We then argue that the conventional solutions to this problem are not adequate. Subsequently, we discuss the realization of the strategic programming combinators, and show how they lead to a general solution of the tangling problem. Finally, we show the extras that the programming paradigm has to offer to the strategic programmer.

3. Term Rewriting

In conventional term rewriting systems the strategy that determines the application of rewrite rules is implicit; rules are usually applied exhaustively to a term. In areas such as program transformation and theorem proving many possible rewriting sequences exist for a given set of valid rewrite rules and a given term. That is, a collection of interesting rewrite rules is usually non-confluent and non-terminating. In a pure rewrite system, control over the application of rewrite rules can only be obtained by means of additional constructors and rewrite rules. There are many approaches that address aspects of the problem of control in rewriting; [63] surveys approaches to rewriting strategies in program transformation.

In this section we illustrate the problem of control in pure rewrite systems, discuss the usual approach to solving this problem, and show how the extension of a rewriting language with programmable rewriting strategies provides a more general solution. In particular, we explain the incarnation of strategic programming in the strategic rewriting language Stratego [64, 62].

As a running example throughout this paper we will use the Extended BNF language for syntax definition and consider various transformations on grammars to illustrate aspects of strategic programming.

3.1. Terms and Rewrite Rules

Term rewriting systems [15] consist of a set of rewrite rules that specify basic transformations on tree structured data, usually described by means of terms.

A *term* is built from simpler terms by means of *constructor application*, i.e., if t_1 to t_n are terms and C is a constructor, then $C(t_1, ..., t_n)$ is another term. Term languages can be described by means of many-sorted algebraic *signatures*, which assign sorts to the arguments of constructors. Terms can be used to represent *abstract syntax trees* of programs. For example, the signature in Figure 3 defines the structure of EBNF grammars. A grammar consists of a start non-terminal and a list of productions. A production consists of a non-terminal and the regular expression defining it. Regular expressions are composed from terminals, non-terminals and empty string, with the combinators star (zero or more repetitions), plus (one or more repetions), opt (zero or one), sequential composition, and alternative.

A *rewrite rule* of the form t_1 `->` t_2 declares the transformation of a term matching pattern t_1 to the instantiation of pattern t_2. A *pattern* is a term with variables. Rewrite rules can be used to express basic

```
module EBNF-core
imports Layout
signature
  constructors
    Grammar : NonTerminal * List ( Prod ) -> Grammar
    Prod : NonTerminal * RegExp -> Prod
    T : Terminal -> RegExp
    N : NonTerminal -> RegExp
    Empty : RegExp
    Star : RegExp -> RegExp
    Plus : RegExp -> RegExp
    Opt : RegExp -> RegExp
    Seq : RegExp * RegExp -> RegExp
    Alt : RegExp * RegExp -> RegExp
```

Figure 3. Signature for abstract syntax of EBNF.

```
module ebnf-laws
imports EBNF
rules
  SAR : Seq(Seq(e1, e2), e3) -> Seq(e1, Seq(e2, e3))
  SAL : Seq(e1, Seq(e2, e3)) -> Seq(Seq(e1, e2), e3)
  AAR : Alt(Alt(e1, e2), e3) -> Alt(e1, Alt(e2, e3))
  AAL : Alt(e1, Alt(e2, e3)) -> Alt(Alt(e1, e2), e3)

  DSAL : Seq(Alt(e1, e2), e3) -> Alt(Seq(e1, e3), Seq(e2, e3))
  DSAR : Seq(e1, Alt(e2, e3)) -> Alt(Seq(e1, e2), Seq(e1, e3))
  DASL : Alt(Seq(e1, e2), Seq(e1, e3)) -> Seq(e1, Alt(e2, e3))
  DASR : Alt(Seq(e1, e2), Seq(e3, e2)) -> Seq(Alt(e1, e3), e2)
  DASL : Alt(Seq(e1, e2), Alt(Seq(e1, e3), e4)) ->
         Alt(Seq(e1, Alt(e2, e3)), e4)
  DASR : Alt(Seq(e1, e2), Alt(Seq(e3, e2), e4)) ->
         Alt(Seq(Alt(e1, e3), e2), e4)
```

Figure 4. Some rewrite rules on regular expressions.

transformation rules and can be considered as operationalizations of the algebraic laws of a language. For example, the rewrite rules in Figure 4 express basic laws of regular expressions such as associativity and distributivity. These rules are considered valid when they preserve the semantics of the expressions that are transformed; in the case of EBNF, the expression before and after transformation should generate the same set of strings (we will not consider preservation of tree structure here). Many more laws exist. In conditional rewrite rules additional tests on the patterns can be stated.

A term is in *normal form* with respect to a set of rewrite rules, if it contains no subterms that match with the left-hand side of a rewrite rule.

(For conditional rewriting the notion of normal form is more complex.) Rewrite engines for term rewrite systems compute the normal form of terms with respect to sets of rules in specifications. This involves exhaustively applying rules to subterms until no more rules apply. A rewrite engine can employ different strategies to order the application of rules. In innermost rewriting all subterms of a term are normalized before rules are applied to the term itself. In outermost rewriting subterms closest to the root of the term are rewritten first. Thus rules are automatically applied throughout a term and no traversals over the syntax tree need to be defined. Exhaustive application of rewrite rules is supported by systems such as OBJ [21], ASF+SDF [17], and ELAN [6], among many others.

3.2. Encoding Control

The complete normalization approach of rewriting turns out not to be adequate for program transformation, because a library of valid and interesting rewrite rules for a given programming language will often form a non-terminating and/or non-confluent rewriting system. In general, it is not desirable to apply all rules at the same time, or to apply all rules under all circumstances.

As an example, consider again the set of rewrite rules in Figure 4. This rewrite system is non-terminating because rules `DSAL` and `DSAR` enable rules `DASR` and `DASL`, and vice versa. If we want to define a transformation to flatten out regular expressions to alternatives of sequences, i.e., disjunctive normal form, we could discard the `DASL` and `DASR` rules. However, if in another part of the transformation we would like to factor out common prefixes or suffixes of alternatives we need a rewrite system with the `DASL` and `DASR` rules. It is not possible to combine these rules in one rewrite system that terminates.

The implicit normalizing strategy of standard rewriting engines does not allow control over the selection of rules to apply. The common solution to this lack of control is the introduction of additional constructors (functions) that achieve normalization under a restricted set of rules. Figure 5 shows how the rewrite system in Figure 4 can be turned into a terminating rewrite system that defines the normalization to 'disjunctive normal form'. (We assume at this point that only the basic `Seq` and `Alt` combinators are used.) To normalize a formula to DNF the function `dnf` should be applied to it.

The `dnf` function mimics the innermost normalization strategy by recursively traversing terms. The auxiliary functions `dnfseq` and `dnfalt` are used to apply the distribution and associativity rules. In the defini-

```
module dnf
imports EBNF
signature
  constructors
    dnf    : RegExp -> RegExp
    dnfalt : RegExp * RegExp -> RegExp
    dnfseq : RegExp * RegExp -> RegExp
rules
  D1 : dnf(T(x))      -> T(x)
  D2 : dnf(N(x))      -> N(x)
  D3 : dnf(Empty)     -> Empty
  D4 : dnf(Alt(x, y)) -> dnfalt(dnf(x), dnf(y))
  D5 : dnf(Seq(x, y)) -> dnfseq(dnf(x), dnf(y))

  S1 : dnfseq(Seq(e1, e2), e3) -> dnfseq(e1, dnfseq(e2, e3))
  S2 : dnfseq(Alt(e1, e2), e3) -> dnfalt(seq(e1, e3), dnfseq(e2, e3))
  S3 : dnfseq(e1, Alt(e2, e3)) -> dnfalt(seq(e1, e2), dnfseq(e1, e3))
  S4 : dnfseq(x, y) -> Seq(x, y) // default

  A1 : dnfalt(Alt(e1, e2), e3) -> dnfalt(e1, dnfalt(e2, e3))
  A2 : dnfalt(x, y) -> Alt(x, y) // default
```

Figure 5. Functionalized rewrite system for 'disjunctive normal form' of regular expressions.

tion of the rules for `dnfseq` and `dnfalt` it is assumed that the arguments of these functions are already in normal form. (In functional programming such auxiliary functions are known as *smart constructors* [19].)

In the solution in Figure 5, the original rules have been completely intertwined with the `dnf` transformation. The associativity rules cannot be reused in the definition of a left factoring transformation. For each new transformation, e.g., factoring out common prefixes or suffixes, or desugaring EBNF combinators into simpler ones, a new traversal function and new auxiliary functions have to be defined. Many additional rules have to be added to traverse the term to find the places to apply the rules. Rules `S4` and `A2` are default rules that only apply if the other rules do not apply. Without this mechanism even more rules would have had to be used to handle the cases were the actual transformation rules do not apply. Default rules were introduced in ASF+SDF [17].

The problem illustrated in the example above is typical for all kinds of program transformations. Overcoming the problems of non-termination and non-confluence leads to encoding of control in terms of additional rewrite rules. This usually leads to a functional programming style of rewriting, overhead in the form of traversal rules for each constructor in the signature, intertwining of rules and function definitions—all of

which makes reuse of rules impossible, and leads to specifications that are much harder to understand.

3.3. Traversal Functions

The overhead caused by the definition of traversals over terms is a major obstacle in the implementation of controlled rewriting. Especially for the specification of transformations for large languages such as COBOL the overhead of traversals makes manual definition infeasible.

Based on the observation that a traversal function has a regular pattern, one can opt to generate traversal rules that capture the default behaviour of a traversal function [10, 11]. By overriding the cases where non-standard behaviour is needed, a useful traversal function is obtained. In this approach typically only a few rewrite rules have to be specified, corresponding to the non-default behaviour of the traversal. However, the number of generated rules still proves to be a source of overhead, be it for the compiler, not the programmer. Furthermore, providing a new traversal scheme requires the implementation of a new generator.

In the scheme described in [9] traversal functions are directly supported in the rewriting engine of ASF+SDF, thus making the generation of rules transparent to the programmer. Figure 6 illustrates the approach applied to the problem of normalization to disjunctive normal form. The specification is the same as that in Figure 5, but the `dnf` function has been declared a traversal function in the signature. The attribute `traversal(trafo,bottom-up)` declares that `dnf` performs a bottom-up traversal over its argument. This means that the function is first applied to the direct subterms (and, thus, recursively to all subterms) before it is applied at the term itself. Rules need to be declared only for those constructs that are transformed. The default behaviour is to reconstruct the term with the original constructor. Thus, in the example only rules `D4` and `D5` from Figure 5 need to be specified. The other `D` rules are obtained by default. For a signature with n constructors only m of which need to be handled in a special way, this saves $n - m$ rules. If n is large—the abstract syntax of COBOL has some 300–1000 constructors (depending on dialect and grammar style)—this is significant. But even for smaller languages, writing out the traversal time and again becomes a pain and obscures specifications.

In addition to the `bottom-up` traversal schema, ASF+SDF provides a `top-down` schema. A `top-down` function traverses down the tree and stops as soon as a rule applies. In addition a traversal can be a transformation (`trafo`) and/or a traversal which accumulates information along the way (`accu`). Finally, traversal functions can be parameterized with

```
module dnf
imports EBNF
signature
  constructors
    dnf    : RegExp -> RegExp {traversal(trafo,bottom-up)}
    dnfalt : RegExp * RegExp -> RegExp
    dnfseq : RegExp * RegExp -> RegExp
rules
  D4 : dnf(Alt(x, y)) -> dnfalt(x, y)
  D5 : dnf(Seq(x, y)) -> dnfseq(x, y)

  S1 : dnfseq(Seq(e1, e2), e3) -> dnfseq(e1, dnfseq(e2, e3))
  S2 : dnfseq(Alt(e1, e2), e3) -> dnfalt(seq(e1, e3), dnfseq(e2, e3))
  S3 : dnfseq(e1, Alt(e2, e3)) -> dnfalt(seq(e1, e2), dnfseq(e1, e3))
  S4 : dnfseq(x, y) -> Seq(x, y) // default

  A1 : dnfalt(Alt(e1, e2), e3) -> dnfalt(e1, dnfalt(e2, e3))
  A2 : dnfalt(x, y) -> Alt(x, y) // default
```

Figure 6. Disjunctive normal form with traversal function

additional arguments that contain static information to be used during traversal.

The advantage of traversal functions is that default traversal behaviour does not need to be implemented manually. This is similar to folds with updatable fold algebras in functional programming (see Section 4) or default visitors in object-oriented programming (see Section 5). However, the approach has a number of limitations.

First of all, there is no separation of rules from strategies. A rule is bound to one specific traversal via the traversal function. It is not possible to reuse rules in different traversals, for example, to normalize under different rule sets. Furthermore, rules are intertwined with strategies, making it hard to distinguish the basic transformation rules from the traversal code, and to argue about correctness of the whole.

Secondly, the traversal function schema provides a limited range of traversals. The bottom-up variant does a full traversal of the tree. The top-down variant stops as soon as it has found a rule application, this requires explicit definition of recursion in rules. Although it is possible to implement a wide range of traversals, this requires gluing together the basic traversals in an ad-hoc manner. It is not possible to define new traversal schemas in a reusable way, i.e., as a new traversal attribute. That would require extending the rewrite engine.

Finally, the traversals provided by the language capture an abstraction, i.e., certain traversal schemata. There is no possibility in the language to give further abstractions for alternative traversal schemata, or

for more elaborate functionality involving traversals. This is desirable for building libraries with language independent strategies. For example, defining substitution without variable capture is similar for many languages, given the shape of variables and variable bindings. Extrapolating the traversal function approach, more and more such abstractions will be captured as additional primitives in the rewrite engine. At some point it will make sense to extend the language with a mechanism for specifying such abstractions generically.

In short, the traversal function approach addresses the easy use of certain traversal schemes, but fails to address the other characteristics of strategic programming, i.e., genericity, partiality, and first-classness. Other approaches to providing control over rewrite rules such as the 'sequence of normalizations' approach of TAMPR [7] similarly address only a few strategic programming ingredients, thus limiting the programmer when venturing outside the standard repertoire.

3.4. Programmable Rewriting Strategies

Stratego [64, 62] is a language for term rewriting with strategies. It provides a full incarnation of the strategic programming idiom by providing the basic combinators of Figure 1 as language primitives (with some slight notational deviations).

In Stratego, pure rewrite rules can be combined in various ways using strategy combinators, instead of encoding control in rewrite rules, Thus, it is possible to select the rules that are needed in a particular transformation and to carefully control their application. Thus, the problem of non-confluence and non-termination in term rewriting is solved by making the rewriting strategy programmable.

Type specialization and strategy update The data manipulated by Stratego's rewrite rules and strategies are first-order ground terms. The typing system is very liberal: only term constructors that are mentioned in the program need to be declared. Stratego strategies are generic by definition. Application of a strategy `s` to a term `t` is written `<s>t`. No explicit type specialization is needed. Labeled rewrite rules provide data-structure specific operations. A rule is referred to by its label. In fact, the label of a rewrite rule can be used directly as a named strategy. Thus, a rewrite rule can be applied to a term of any type, just like strategies. If there is no match the application fails.

Because rewrite rules double as strategies, there is no need for a strategy update combinator like `adhoc`. Instead, the choice combinator can be used to update strategies with rules. Such updates do not involve a type case, but rely on success and failure of pattern matches.

Control and traversal combinators A strategy is a program which may fail, for example, when the left-hand side of a rule does not match with a term. Control over strategies is defined using sequential composition (`_;_`), non-deterministic local choice (`_+_`), guarded deterministic local choice (`_<_+_`), and non-determinstic global choice (`_++_`). The local choice operators commit after one of the branches has succeeded, even if the continuation fails. The global choice operator guarantees a choice which makes the continuation succeed, if possible, and is implemented using global backtracking.

Traversals are defined by means of the one-step traversal operators `all(_)`, `one(_)` and a few variations. Type-specific traversal can be defined by means of congruence operators. For each n-ary constructor `C` in the signature the corresponding congruence operator `C(s1,...,sn)` transforms `C` terms by applying the argument strategies to the direct subterms, i.e., the application `<C(s1,...,sn)>C(t1,...,tn)` yields the term `C(<s1>t1,...,<sn>tn)`.

First-class pattern matching Rewrite rules are composed from more primitive operations. The left-hand side of a rule *matches* the subject term against a term pattern and the right-hand side of a rule *constructs* a new term by instantiating a term pattern. Implicitly, the *scope* of the pattern variables is restricted to the rule, i.e., their bindings cannot be observed outside a rule. Instead of restricting these operations to rewrite rules, Stratego provides them as first-class citizens in their own right. Thus the operation `?t` matches the subject term to the term pattern `t`, and the operation `!t` replaces the subject term by the instantiation of the term pattern `t`. The scope of the pattern variables in the patterns of the `?t` (match) and `!t` (build) operations can be managed by means of the scope operator `{x1,...,xn:s}`, which delimits the scope of the variables `xi` to the strategy expression `s`.

Rewrite rules are defined in terms of these basic actions. That is, a conditional rewrite rule `L : t1 -> t2` is translated to the strategy definition `L = {x1,...,xn: ?t1 ; !t2}`, where the `xi` are the variables in the rule. Since match and build are first-class citizens they can be used at will. For instance, a partially bound match operation can be passed to a traversal to look for a subterm with a particular shape. Furthermore, the operations can be used to implement syntactic abstractions such as apply and match `<s> t1 => t2`, which denotes `!t1; s; ?t2` and anonymous rules `\ t1 -> t2 \`, which denotes `{x1,...,xn: ?t1; !t2}`, where the `xi` are the variables of left-hand side `t1`.

```
strategies
  try(s)       = s <+ id
  repeat(s)    = rec x(try(s; x))
  map(s)       = rec x([] + [s | x])
  topdown(s)   = rec x(s; all(x))
  bottomup(s)  = rec x(all(x); s)
  alltd(s)     = rec x(s <+ all(x))
  oncetd(s)    = rec x(s <+ one(x))
  innermost(s) = rec x(all(x); try(s; x))
```

Figure 7. Defined strategies

Defined strategies Strategy definitions allow the definition of strategy operators as abstractions of strategy expressions. Definitions can be recursive, but recursive strategies can also be defined using the `rec_(_)` shorthand notation, which allows for anonymously recursive strategy expressions. Figure 7 shows a few definitions of generic strategies in Stratego notation. The Stratego Standard Library defines a wide range of generic and specific strategies, including the ones in Figure 7.

The `map(s)` strategy is an example of data-type specific traversal in which a strategy `s` is applied to the elements of a list. Although this strategy is type specific, it can actually be applied to terms of arbitrary type; when the term is not a list the strategy simply fails.

Implementation and extensibility Stratego is its own language with the basic strategy combinators forming its basic constructs. Stratego is implemented by compilation to C code. The implementation of term representation is provided by the ATerm Library [8], which maintains *maximal sharing* of subterms. As a consequence, checking for equality of terms is a constant time operation. Furthermore, the library takes care of garbage collection and persistence of terms for storage and exchange with other tools. Failure is implemented by means of C's exception handling mechanism represented by the `setjmp` and `longjmp` functions. The implementation of full backtracking is an extension of this mechanism provided by the Choice Point Library [55].

Stratego is open to extension with new basic operators. User defined primitives can be added by providing an implementation in C, which can be accessed via the `prim(f,[t1,...,tn])` operator. Such primitive functions can be implemented as 'high-level' C functions with ATerms as arguments and results. Strategies are passed as C function pointers. As an example consider the C implementation of the `all(_)` traversal primitive in Figure 8. (The actual implementation handles ATerm lists as a separate case.) In this fashion, the Stratego Library provides primitives for arithmetic, I/O, and process management.

```
ATerm _all(ATerm t, ATerm f(ATerm))
{
  Symbol c = ATgetSymbol((ATermAppl) t);
  int i, arity = ATgetArity(c);
  ATerm kids[arity];
  for(i = 0; i < arity; i++)
    kids[i] = f(ATgetArgument(t, i));
  return (ATerm) ATmakeApplArray(c, kids);
}
```

Figure 8. C implementation of all combinator

3.5. Composing Strategies

We will now define several transformations on EBNF grammars to illustrate the application of strategies and discuss several extensions of the basic scheme.

Rule selection Figure 9 defines several simplification strategies for regular expressions. The strategies `Simplify` and `Desugar` are abstractions for sets of rules. The `Simplify` strategy combines a number of unit and idempotency rules on regular expressions. The `Desugar` strategy combines three rules that define regular operators in terms of sequence and alternative as defined in Figure 10.

The other strategies combine different associativity and distributivity rules with the standard simplification rules. The `left-factorize` strategy associates sequences to the right (`SAR`) and factorizes on the left (`DASL`), i.e., if the prefix of two adjacent alternatives is equal they can be factored out. Right factorization works just the other way around; associate sequences to the left (`SAL`) and factor out common suffixes

```
module ebnf-simplify
imports EBNF ebnf-laws ebnf-traversals ebnf-sugaring ebnf-inline
strategies
  Simplify =
    SEL + SER + AEE + SE + PE + OE + ADL + ADR + SDL + SDR
    + OO + SS + PS + OS + PO + OP + AOL + AOR + AAI

  left-factorize  = innermost(SAR + DASL + AAR + Simplify)
  right-factorize = innermost(SAL + DASR + AAR + Simplify)
  Desugar     = DefOpt + DefStar + DefPlus
  dnf         = innermost(SAR + DSAL + DSAR + AAR + Simplify)
  dnf-desugar = innermost(SAR + DSAL + DSAR + AAR + Desugar + Simplify)
  ebnf-sugar  = topdown(try(RecStar + RecPlus + RecOpt))
```

Figure 9. Normalization of regular expressions under different sets of rules

(DASR). The dnf strategy implements the simplification to 'disjunctive normal form' discussed in Sections 3.2 and 3.3 in addition to standard simplifications. The strategy dnf-desugar adds desugaring rules to dnf normalization. Finally, strategy ebnf-sugar replaces BNF definitions of optionals and lists with the corresponding regular operators in a single topdown traversal. Thus, using a single set of rules and generic definitions of rewriting strategies, different transformations can be implemented. Note that the strategies in Figure 9 concisely cover a much wider range of grammar transformations than Figures 5 or Figure 6, and that all EBNF term constructors are taken into account. Note also that these strategies can be applied to entire grammars just as well as to individual regular expressions.

New constructors Figure 10 presents the definitions of the desugaring rules mentioned above. The rules express the regular operators Opt, Star, and Plus in terms of the alternatives and sequences and auxiliary non-terminals. Since the definitions of Star and Plus need to be recursive, the auxiliary recursion operator Rec is introduced, which represents a fixed point of its argument expression. For example, a Star(e) expression is defined as the fixed point over non-terminal N(x) of empty or e followed by N(x), for some new name x. Thus, it is possible in Stratego, as in other algebraic specification languages, but not in (mainstream) functional programming languages, to extend a signature imported from another module with additional constructors. Traversals implemented

```
module EBNF
imports EBNF-core
signature
  constructors
    Delta : RegExp
    Parens : RegExp -> RegExp
    Rec : NonTerminal * RegExp -> RegExp
```

```
module ebnf-sugaring
imports EBNF ebnf-traversals
rules
  DefOpt :
    Opt(e) -> Alt(Empty, e)
  DefStar :
    Star(e) -> Rec(x, Alt(Empty, Seq(e, N(x)))) where new => x
  DefPlus :
    Plus(e) -> Rec(x, Alt(e, Seq(e, N(x)))) where new => x
```

Figure 10. Desugaring of regular expressions.

using generic traversal strategies can cope with such extensions since only relevant constructs are touched.

Unique names The auxiliary non-terminals used in the desugaring in Figure 10 are generated by the `new` primitive strategy, which generates a unique new name. This is possible as a result of the maximal sharing implementation of terms; all strings in all 'live' terms are known such that the name generator can guarantee that no existing string is produced as new name, thus avoiding name clashes.

Generic term (de)construction Figure 11 defines a transformation from EBNF to BNF. The expression defininining a BNF production consists of alternatives consisting of sequences of non-terminals and terminals, i.e., regular expressions in disjunctive normal form without iteration operators. The `dnf-desugar` strategy from Figure 9 achieves this normal form, except that it leaves the recursive definitions of the iteration operators in place. To achieve proper BNF these need to be lifted out and made into proper productions. The strategy in Figure 11 achieves this by means of the generic `collect-split(f)` strategy, which applies a transformation `f` to the subterms of a term, possibly splitting it into a pair `(t,ts)` of a residual term `t` which stays behind and a list of terms `ts` which is collected. In the translation to BNF, the `Rec` terms produced by desugaring are replaced by the non-terminal with the recursion variable as its name. At the same time the `Rec` terms are collected after being transformed to `Prod` terms, i.e., productions. The list

```
module ebnf-to-bnf
imports EBNF lib ebnf-simplify
strategies
  ebnf-to-bnf =
  iowrap(
    dnf-desugar;
    Grammar(id, collect-split(\Rec(x,e) -> (N(x),[Prod(x,e)])\); conc)
  )
```

```
strategies
  collect-split(splitter) =
    rec x(CollectSplit(x, splitter <+ !(<id>,[])))
rules
  CollectSplit(s, splitter) :
    c#(as) -> (t, <union> (ys, <unions> xs))
    where <unzip(s)> as => (bs, xs);
          <splitter> c#(bs) => (t, ys)
```

Figure 11. Simplification of grammars in EBNF to grammars in BNF.

of transformed original productions and the list of collected productions are then concatenated by `conc`.

The generic definition of `collect-split` employs the `_#(_)` operator for generic (de)construction of a term into its constructor and lists of arguments. In the left-hand side of the `CollectSplit` rule, a term is deconstructed into its constructor `c` and arguments `as`. In the condition of the rule a new term is composed from the same constructor `c` and the transformed list of arguments `bs`.

Dynamic rules Another limitation of term rewriting that was not mentioned above is the fact that rewrite rules are context-free. That is, when matching the left-hand side of a rule, only the information present in the subject node and its subterms can provide information about the applicability of the rule. For some problems such as inlining and bound variable renaming, information from the context of a term is needed. In conventional settings this problem is solved by threading information along the traversal over the tree by means of additional parameters of the traversal function. This affects all control functions that are called along the way. Stratego provides *scoped dynamic rewrite rules* [66], which allows a traversal to generate a new rewrite rule at the point where context information is available and apply it at a later stage, without explicitly threading the information.

The application of dynamic rules is illustrated by the conversion from BNF to EBNF in Figure 12. The `ebnf-sugar` strategy recognizes productions that define optionals or lists and transforms their bodies accordingly. For example, `X ::= eps | A X` is translated to `X ::= A*`. Next we would like occurrences of `X` to be replaced by the regular expression `A*`, i.e., we would like to inline the definition of `X`.

With static rewrite rules this cannot be achieved, unless we know at specification-time which non-terminals to replace with which bodies. By means of dynamic rewrite rules this can be achieved. The `DeclareProduction` strategy in Figure 12 matches a production and *dynamically* generates a rewrite rule `Inline` mapping the defined non-terminal to the defining regular expression of the production. A rule defined within the `rules(_)` construct inherits the bindings to any pattern variables from the context.

The `inline(pred)` strategy traverses a grammar, declaring each production it encounters provided the predicate `pred` succeeds for it. Subsequently the grammar is traversed again to apply the `Inline` rule. In the `bnf-to-ebnf` strategy the `inline` strategy is instantiated such that only non-terminals defining iteration operators are inlined.

```
module bnf-to-ebnf
imports EBNF lib ebnf-simplify
strategies
  bnf-to-ebnf =
    iowrap(
      dnf; ebnf-sugar;
      inline(Prod(id, Star(id) + Plus(id) + Opt(id));
             not(recursive-prod))
    )
```

```
module ebnf-inline
imports EBNF dynamic-rules
strategies
  inline(pred) =
    {| Inline :
       topdown(try(pred; DeclareProduction));
       topdown(try(Inline)) |}
  DeclareProduction =
    ?Prod(x, e);
    rules( Inline : N(x) -> e )
  recursive-prod =
    Prod(?x, oncetd(?N(x)))
```

Figure 12. Replacing recursive productions by regulare expression combinators

4. Functional Programming

The functional programming paradigm meets a few requirements for strategic programming in a natural way. One is used to programming with combinators, e.g., functional composition or monadic binding (recall sequential composition of strategies via `_;_`). Moreover, effects like success and failure are straightforward to handle with monads (recall the ingredient of partiality). Parametric polymorphism allows us to write functions which can be applied to all possible types (recall the ingredient genericity). One is also used to the use of recursion schemes, especially for lists—think of `foldl` and `foldr`. Those are related to traversal. However, *generic* term traversal is not straightforward to achieve. Also, the idea of mixing polymorphic and monomorphic functions as suggested by the basic combinator `adhoc` is alien. Before we remedy these problems, we review some less generic approaches to traversal. Then, we will spell out the functional incarnation of strategic programming based on the simple formula "strategies as functions". We will also pay attention to challenges imposed on a type system if we want to model functional strategies.

We will give our examples in Haskell throughout this section, although functional strategies also make perfect sense for other functional languages. The model we discuss in the sequel relies on a few extensions

```
module Ebnf where
 type Grammar     = (NonTerminal, [Prod])
 type Prod        = (NonTerminal, RegExp)
 data RegExp      = Terminal Terminal
                  | NonTerminal NonTerminal
                  | Empty
                  | Star RegExp
                  | Plus RegExp
                  | Opt RegExp
                  | Seq RegExp RegExp
                  | Alt RegExp RegExp
 type Terminal    = String
 type NonTerminal = String
```

Figure 13. Abstract syntax of EBNF as Haskell datatypes.

on Haskell 98 [23] which will be pointed out on the way. We will use the EBNF grammar from the previous section (see Figure 3) and corresponding grammar-engineering examples for illustrative purposes. In Figure 13, the grammar is shown after its transposition to a Haskell system of datatypes.

4.1. Tangled Traversal

Let us investigate the standard style of traversal programming in the functional paradigm. It turns out that term traversal is usually spelled out in ordinary recursive functions using pattern matching. We call this style "tangled traversal" because it defines functions specific to certain datatypes in a way that a traversal scheme is encoded exhaustively for the datatypes, and the traversal scheme is not separated from truly problem-specific functionality. To give a simple example, let us consider the problem of determining all used nonterminals in an EBNF grammar. To this end, we need a family of functions to descend into EBNF terms and to collect nonterminals all over the place. Such tangled functions are shown in Figure 14.

We encounter basically the same problems as in the term-rewriting context when a functionalized style with additional constructors was employed. In functional programming, a traversal is encoded by pattern matching to cover all constructors of a type, and by a family of functions to cover all types involved in the traversed data structures. In the example, we need to descend *down to* regular expressions. In addition, we also descend *into* compound regular expressions to collect the contributing sets (in fact, for simplicity: lists) of used nonterminals. Generic and specific functionality is obviously intertwined in the functions. If any of these two parts of the traversal would have to be revised, a new exhaus-

```
used :: Grammar -> [NonTerminal]
used = concat . map usedInProd . snd
 where
  usedInProd :: Prod -> [NonTerminal]
  usedInProd =  usedInRE . snd
  usedInRE :: RegExp -> [NonTerminal]
  usedInRE (NonTerminal no) = [no]
  usedInRE (Star sy)        = usedInRE sy
  usedInRE (Plus sy)        = usedInRE sy
  usedInRE (Opt sy)         = usedInRE sy
  usedInRE (Seq sy1 sy2)    = (usedInRE sy1) ++ (usedInRE sy2)
  usedInRE (Alt sy1 sy2)    = (usedInRE sy1) ++ (usedInRE sy2)
  usedInRE _                = []
```

Figure 14. Used nonterminals in an EBNF in tangled traversal style.

tive family of traversal functions is due. Also, encoding the traversal scheme becomes cumbersome when more complex systems of datatypes as pointed out for term-rewriting incarnation.

4.2. Traversal by Generalized Folds

Let us look for improvements over tangled traversal. It is folklore that all algebraic datatypes admit a concept of generalized folds—in fact, there are quite a few different variations such as catamorphic and paramorphic folds, non-monadic and different kinds of monadic folds [53]. This concept indeed suggests to separate out the traversal scheme of folding into a fold function, and then to use a fold algebra to collectively represent all the functionality "replacing" the various constructors. In Figure 15, we show a (paramorphic and monadic) fold-algebra type, and part of the corresponding family of fold functions for our sample system dealing with the abstract syntax of EBNF. The fold algebra type is parameterized by the result types of folding for the various datatypes in the EBNF system. The fold function for `RegExp` is given as an instance of a corresponding `Fold` class where we perform a case discrimination over forms of regular expressions, and we recurse into terms via the overloaded `fold` function. In a given equation (e.g., for `sy@(Star sy')`), the `fold` function is recursively called for the subcomponents (`sy'` in the example). The results are passed to the corresponding algebra member (i.e., `star`). The member is looked up via record-component selection (cf. `star alg ...`). In fact, since we are using paramorphic folds [51], we also pass the original term `sy` to the fold member.

The fold-algebra type and the fold function can be reused. In fact, they can also be generated. Generalized folds support separation of concerns, and reuse of the traversal scheme.

```
data Monad m => FoldAlg gr pr re st m = FoldAlg
 { grammar     :: Grammar -> (st,[pr]) -> m gr,
   prod        :: Prod    -> (st,re)   -> m pr,
   terminal    :: RegExp  -> st        -> m re,
   nonterminal :: RegExp  -> st        -> m re,
   empty       :: RegExp               -> m re,
   star        :: RegExp  -> re        -> m re,
   plus        :: RegExp  -> re        -> m re,
   opt         :: RegExp  -> re        -> m re,
   seq         :: RegExp  -> re -> re  -> m re,
   alt         :: RegExp  -> re -> re  -> m re,
   string      :: String               -> m st
 }

class Fold alg x y | alg x -> y where
 fold :: alg -> x -> y

instance Monad m => Fold (FoldAlg gr pr re st m) RegExp (m re) where
 fold alg sy@(Terminal st)    = fold alg st  >>= terminal alg sy
 fold alg sy@(NonTerminal st) = fold alg st  >>= nonterminal alg sy
 fold alg sy@Empty            = empty alg sy
 fold alg sy@(Star sy')       = fold alg sy' >>= star alg sy
 fold alg sy@(Plus sy')       = fold alg sy' >>= plus alg sy
 fold alg sy@(Opt sy')        = fold alg sy' >>= opt alg sy
 fold alg sy@(Seq sy1 sy2)    = fold alg sy1 >>= \sy1' ->
                                fold alg sy2 >>= \sy2' ->
                                seq alg sy sy1' sy2'
 fold alg sy@(Alt sy1 sy2)    = fold alg sy1 >>= \sy1' ->
                                fold alg sy2 >>= \sy2' ->
                                alt alg sy sy1' sy2'
```

Figure 15. The fold-algebra type for EBNF and folding over regular expressions.

Updatable fold algebras The mere separation of fold functions and fold algebras is not sufficient for truly generic programming. In [45], we observed that the key requirement for scalable traversal technology is that one can *easily* mix genericity and specificity, that is, one can refine suitable generic fold algebras so that only problem-specific members of the algebra need to be touched. This is the key to reuse and genericity. One can, for example, think of a completely generic fold algebra for collecting entities via a traversal. A corresponding fold algebra `crush` is shown in Figure 16. We use the `mappend` operation of a `Monoid` to combine intermediate values. The empty collection corresponds to the other operation `mempty` of a `Monoid`. Such generic fold algebras can again be generated from a given system of datatypes.

Generic programming can now commence by updating generic fold algebras for those constructors which require specific functionality. In

```
crush :: (Monad m, Monoid u) => FoldAlg u u u u m
crush =  FoldAlg {
  grammar = \_ -> \(no,ps)    -> return (no `mappend`
                                         (foldl mappend mempty ps)),
  prod    = \_ -> \(no,sy)    -> return (no `mappend` sy),
  terminal    = \_ -> \st     -> return st,
  nonterminal = \_ -> \st     -> return st,
  empty  = \_                 -> return mempty,
  star   = \_ -> \re          -> return re,
  plus   = \_ -> \re          -> return re,
  opt    = \_ -> \re          -> return re,
  seq    = \_ -> \re1 -> \re2 -> return (re1 `mappend` re2),
  alt    = \_ -> \re1 -> \re2 -> return (re1 `mappend` re2),
  string = \_                 -> return mempty
 }
```

Figure 16. A fold algebra for collection.

Haskell, record-update notation is appropriate since fold algebras are just records. We can encode our earlier example to collect all used nonterminals in a much more concise manner (when compared to tangled traversal, or exhaustive construction of a fold algebra). The following one-liner implements the traversal problem:

```
used = fold crush{nonterminal = \(Nonterminal st) -> \_ -> Just [st]}
```

The resulting style of "updatable generic fold algebras" [45] addresses traversal genericity in the same sense as strategic programming. However, strategies obviously go beyond generalized folds since traversal schemes can be composed from basic combinators like `all` and `one`. It is important to note that strategic traversal combinators do not recurse by themselves. Recursion is rather expressed explicitly by means of (recursive) strategy definitions. In fact, updatable fold algebras suffer from the same problem as the traversal-function approach in term rewriting when only a fixed set of traversal schemata is considered.

4.3. Strategies as functions

Since strategies are meant to admit a concept of application, that is, they can be applied to a concrete datum, the desired functional model of strategies suggestes itsself: We want strategies to correspond to functions. Since strategies are required to be generic, we obviously need polymorphic functions. In fact, we favour monadic polymorphic functions since we want to use monads [68] to deal with effects like partiality, state and others. In a first approximation, we can think of strategies as

```
id     = return
fail   = const mzero
seq    = \f g x -> f x >>= g
choice = \f g x -> (f x) `mplus` (g x)
```

Figure 17. Strategy combinators as polymorphic function combinators.

functions of the following type:

$$\forall x\ m.\texttt{Monad}\ m\ \Rightarrow\ x \rightarrow m\ x$$

Here, we assume that strategies preserve the input type, that is, the resulting datum is of the same type as the input datum. We will later explore another type scheme but let us stick to *type preservation* for the moment.

Strategy combinators If strategies are polymorphic functions, then strategy combinators are polymorphic function combinators. In Figure 17, we show the first few implementations of strategy combinators in Haskell. In this stage, we leave out the traversal combinators `all` and `one`, and also the `adhoc` combinator because their treatment requires more effort. All these combinators involve `Monad` or even `MonadPlus` operations. The combinator `id` is defined as the monadic identity function in terms of the monad member `return`. The combinator `fail` is defined as a constant function (i.e., a function ignoring its argument) which always returns the failure element `mzero` (of an accordingly extended monad). The combinator `seq` denotes monadic function composition. The combinator `choice` models left-biased choice again using an operation of an extended monad which copes with addition (i.e., `MonadPlus` in Haskell). To deal with partiality or different kinds of choice, we only have to select a suitable monad instance, e.g., the list monad, or a more sophisticated monad with other forms of backtracking [26]. We refrain from giving the types of the function combinators in this stage because those deserve some proper discussion.

Basic traversal combinators The encoding of the traversal combinators can receive inspiration from the fold-algebra approach. Traversal combinators like `all` and `one` can be encoded (in Haskell) in a way similar to the `fold` function, that is, per class-overloading to provide a definition of the traversal combinator via an instance per term type, and per pattern matching to exhaust all possible constructors via an equation per constructor. Along the way, we might need to go beyond Haskell 98 because multi-parameter classes and functional dependencies are convenient [33]. In Figure 18, we show the completely systematic

```
all (Terminal st)    f = f st >>= return . Terminal
all (NonTerminal st) f = f st >>= return . Nonterminal
all Empty            f = return Empty
all (Star re)        f = f re >>= return . Star
all (Plus re)        f = f re >>= return . Plus
all (Opt re)         f = f re >>= return . Opt
all (Seq re1 re2)    f = f re1 >>= \re1' ->
                         f re2 >>= \re2' ->
                         return (Seq re1' re2')
all (Alt re1 re2)    f = f re1 >>= \re1' ->
                         f re2 >>= \re2' ->
                         return (Alt re1' re2')
```

Figure 18. Instance of `all` for regular expressions.

code for the `all` instance for `RegExp`. Traversal combinators are, of course, not parameterized by fold algebras as in the case of `fold` but by strategies (say, functions). Also the traversal combinators do not recurse by themselves, but they only apply argument strategies to children.

Strategy update There is one combinator left which needs to be integrated with functional programming to enable the strategic programming idiom, namely the combinator `adhoc` for updating a polymorphic function according to a monomorphic special case. This expressiveness is not available in widely used functional languages. However, it can be simulated via an encoding technique. Let us sketch the underlying notion of intensional type analysis or intensional polymorphism [22]. Intensional type analysis allows one to define a polymorphic function by type case on its type parameter. The prime application of this concept was to be able to compile (parametric) polymorphism in an efficient manner, that is, to allow different code depending on the actual parameter type. Type case is, like the standard **case** form of expression, about case discrimination. However, the patterns are not constructor patterns but type patterns, and the case expression is not an expression but a type. Suppose, α is the term type covered by the update *upd* in an application of `adhoc`, and β is the type of the term passed to the updated strategy, then we can encode the combinator `adhoc` in terms of type case as follows:

$$\begin{array}{lcl} \texttt{adhoc}\ s\ upd & = & \textbf{typecase}\ \beta\ \textbf{of} \\ & & \quad \alpha \rightarrow upd; \\ & & \quad \_ \rightarrow apply\ s \end{array}$$

Type case can be simulated by Haskell type classes as employed for the traversal combinators, too. The somewhat involved encoding is described at length in [39].

To summarize the Haskell incarnation, the strategy combinators can be encoded either directly as plain functions, or we have to resort to Haskell classes to provide the definitions via overloading for all possible term types. In fact, there a several models conceivable to provide functional strategies, and we will come back to this issue in the subsection on typeful strategies.

4.4. Functional strategic programming

Given all the basic combinators, strategic functional programming can commence. Strategy definitions are simply function definitions. All the library definitions given earlier in Figure 2 carry over modulo some notational variations. Before we work out an example of a functional strategic program, we slightly enrich our incarnation to allow for different kinds of strategies than indicated in the definition section.

Type-unifying strategies The combinator `id` is inherently type-preserving because it preserves the very input term. Also, the way we explained the combinators `all` and `one` in the definition section, they are type-preserving by definition since they preserve the outermost constructor, and this can only be faithful if the types of all children are preserved as well. For some other combinators type preservation is not implied. To give an example, we can very well compose functions with different (although suitably related) types via `seq`. In previous work we have shown [45, 38, 44] that there is one more prime type scheme for strategies in addition to type preservation, namely *type unification*. That is, a strategy application always returns a result of a fixed (say, unified) type regardless of the type of the input datum. In fact, the kind of traversal which we need for the earlier example to collect all used nonterminals will ultimately rely on a type-unifying strategy.

In Figure 19, we define two strategy combinators which specifically deal with type-unifying strategies. The nullary combinator `build` constructs a strategy from a value to always return the given value. One could also say that `build` embodies the monadic constant function constructor. The binary combinator `pass` allows us to compose two functions which share the input but where the result of the first application

```
build = const . return
pass  = \f g x -> f x >>= \y -> g y x
comb  = \o s s' -> s  `pass` \u  -> s' `pass` \u' -> build (o u u')
crush = \s -> comb mappend s (allTU (crush s))
```

Figure 19. Some type-unifying combinators.

is sent as additional input to the second. We illustrate this simple type-unifying combinator via the defined strategy `comb`. It lifts a binary combinator from the value to the function level. Finally, we define the scheme `crush` for collection of entities via a traversal. To this end, we assume a type-unifying variant `allTU` of the basic strategy combinator `all`. In this variant, we do not preserve the outermost constructor but we rather reduce all the intermediate results obtained by the application of the argument strategy to the children. Reduction is performed via `Monoid` operations. The important property of the strategy definition `crush` is that it is *derived* from more basic strategy combinators using just the ordinary scheme of function definition. Recall that `crush` was primitive in the more restricted setting of fold algebras.

We can now easily reconstruct the example to collect all used nonterminals in an EBNF. We have to apply the scheme `crush` to a strategy which is capable of identifying nonterminals. The latter strategy `identify` is constructed from a monomorphic function which deals with `Symbol`s.

```
used  = adhoc crush identify
  where
    identify (Nonterminal st) = Just [st]
    identify _                = Just []
```

Higher-order style A by-product of a truly functional model is that we immediately can deal with strategies in higher-order style. In fact, the higher-order character of functional strategies is already relevant for the encoding of the functional strategy combinators. Also, the higher-order style is in turn a prerequisite for the monadic style of programming. Monads allow us to deal with certain effects along the performance of a strategy. We use monads in a pervasive manner to enable the partiality ingredient of strategies. In fact, the use of a `MonadPlus` for this purpose compensates for modelling potential failure in any other manner that would possible require a language extension. Ultimately, all kinds of monads would be appropriate, e.g., for reflective purposes to keep track of number of failures, depth of the term and others, or a state monad to keep track of certain codes assigned in a strategy. The monadic style (and hence, higher-order style) of strategic programming is illustrated in the sequel.

In Figure 20, we show a code snippet dealing with yaccification [46] of EBNFs, that is, removal of extended BNF expressiveness to obtain a grammar which only uses BNF (say, YACC) forms. We only indicate one elimination rule, namely removal of stars. The interesting bit in the definition is the way how auxiliary grammar rules arising from the

A type-preserving strategy for yaccification

```
yaccify t = unState ((bottomup (adhoc id yaccify')) t) (1,[])
  where
    yaccify' :: RegExp -> State New RegExp
    yaccify' (Star sy) =
      next >>= \i -> let no = show i in
      extend [(no,Alt Empty (Seq sy (NonTerminal no)))] >>= \() ->
      return (NonTerminal no)
    yaccify' sy        = return sy
```

State monad used for accumulation of new rules and generation of nonterminals

```
newtype State s a = State (s -> (s,a))
unState (State f) = f
type New          = (Int,[Prod])
```

Operations of state monad

```
next      :: State New Int
next      =  State (\(i,eb) -> ((i+1,eb),i))
extend    :: [Prod] -> State New ()
extend ps =  State (\(i,ps') -> ((i,ps'++ps),()))
```

Figure 20. Functional strategy for Yaccification.

yaccification are synthesized and accumulated. To this end, we use a `State` monad. The state type `YState` for yaccification encapsulates two values; firstly a kind of maximum counter to generate fresh nonterminal names, secondly a set of rules accumulated along the yaccification. By using a state monad, we can simply describe yaccification as a type-preserving strategy while the emerging set of auxiliary rules and the maximum counter do not need to be entangled in the strategy. There is a monad operation `next` to supply the next legal integer from which a nonterminal can be constructed. There is another monad operation `extend` to register some more auxiliary rules. These rules together with the transformed EBNF amount to the final yaccified grammar. Note that it is important that we use a bottom-up traversal because only in this case a single traversal is sufficient to eliminate all EBNF expressiveness. We could further elaborate the use of the state monad to perform cashing. This would avoid the repeated generation of auxiliary rules for a given EBNF form.

Let us use the above examples to briefly point out a few differences between the term-rewriting and the functional incarnation. In Stratego, the aspect of generating fresh symbols was enabled by the `new` primitive, whereas we use monadic programming for this purpose. Note that our state is more complex in that we also hold auxiliary generated rules.

Too liberal rank-1 types with implicit top-level quantification

```
id     :: Monad m => x -> m x
fail   :: MonadPlus m => x -> m x
seq    :: Monad m => (x -> m x) -> (x -> m x) -> x -> m x
choice :: MonadPlus m => (x -> m x) -> (x -> m x) -> x -> m x
```

Rank-2 types with explicit universal quantification

```
id     :: forall m x. Monad m => x -> m x
fail   :: forall m x. MonadPlus m => x -> m x
seq    :: forall m. Monad m =>
               (forall x. (x -> m x)) ->
               (forall x. (x -> m x)) ->
               (forall x. (x -> m x))
choice :: forall m. MonadPlus m =>
               (forall x. (x -> m x)) ->
               (forall x. (x -> m x)) ->
               (forall x. (x -> m x))
```

Figure 21. Rank-1 vs. Rank-2 types for strategy combinators.

Recall that these auxiliary rules were first stored in the regular expressions themselves in the Stratego example, and then a separate step was needed to separate them out. The first example dealing with the collection of used nonterminals is also enlightening for a comparison. In Stratego, a type-unifying traversal is usually implemented by means of the combinator `_#(_)` for generic term deconstruction and subsequent list-processing. Such an approach could not be typed in the functional setting. Hence, we assumed a dedicated traversal combinator `allTU` to perform the pairwise reduction without exposure of a list to the strategic programmer.

4.5. Typeful strategies

Let us elaborate on the "Strategies as functions" idea. So far we did not spot any major problems in giving types to the strategy combinators, or to strategies themselves. We only pointed out that strategies are polymorphic and monadic functions, and that the traversal combinators and the `adhoc` combinators require some encoding effort to support these combinator on a per-type basis. This explanation was driven by our interest to inhabit strategies in Haskell. Let us now analyse possible models of functional strategies, and the implications for a typeful treatment at a more conceptual level.

Rank-2 types We start with the observation that simple polymorphic type schemes are not sufficient. We need rank-2 types since strategy combinators are combinators on necessarily polymorphic functions. In Figure 21, we illustrate this important difference via the types for some basic strategy combinators. For simplicity, we restrict ourselves to the type-preserving scheme in the figure. Consider, for example, sequential composition. In the rank-1 case, the type variable `x` is implicitly quantified at the top-level. Hence, the combinator could also be used to compose monomorphic functions (resulting in a monomorphic function) once the type parameter `x` was supplied. This case must be ruled out because we need to compose polymorphic functions for the sake of genericity of strategies. The rank-2 version obviously insists on polymorphic functions for the argument and result types of the combinators. Yet another reason why we need to insist on rank-2 types relates to the traversal combinators. For short, the strategy constructed via a traversal combinator like in `all(s)` must be quantified over a different type variable than `s` itself simply because `s` will be applied to various children of a term which `all(s)` is applied to. All the terms are potentially of different types. Hence, the type of the argument strategy of a traversal combinator needs to be explicitly quantified.

Trade-offs Rank-2 types are not part of Haskell 98, but they are supported, for example, in the GHC implementation of Haskell. Using rank-2 types implies a certain trade-off because type inference is not decidable, and the types, in general, get more complicated. Conceptually, rank-2 types are however in place. In the discussion below we will see that we can use other type system extensions instead of rank-2 types although this usually implies some encoding obligations. All these extensions have their trade-offs, and they indicate that strategic programming imposes challenges on a type system.

Kinds of polymorphism Let us now investigate in some detail with what kind of polymorphism we are faced anyway. The standard reading of the types in Figure 21 suggests parametric polymorphism (especially, in Haskell). It turns out however that parametric polymorphism [12] and other well-established kinds of polymorphism are not immediately suitable in our context. Strategies goes beyond parametric polymorphism for two reasons. Firstly, the `adhoc` combinator allows us to enforce specific behaviour for a given type. Secondly, one can also easily check that generic traversal functions do not meet parametricity [67, 50, 2]. To give a simple example, `all(fail)` succeeds for constant constructors, and it fails for any term with at least one child. Hence, we can ob-

serve the values corresponding to the type parameter. The issue about `adhoc` suggests to consider ad-hoc polymorphism [12] but this form on its own is again not sufficient because it does not allow us to write sufficiently generic functions which are applicable to all (term) types, but still expose uniform behaviour for most types. We mentioned intensional polymorphism [22, 13] to provide the concept underlying `adhoc` but it is not sufficient on its own because it does not cover the traversal part of our setting. The same is true for dynamic typing [3]. Finally, we might consider polytypism but again it misses some important ingredient, namely it lacks the type-case expressiveness needed for `adhoc`. A simple conclusion is hence that only a combination of some of these forms of polymorphism can be appropriate.

Models of strategies Despite these complications, a number of models of functional strategies can be identified. Some of these models can be qualified as implementation models because they imply some generative support for the traversal combinators and/or `adhoc`. Others can be worked out as proper language extensions so that even the definition of `all`, `one`, and `adhoc` can be given in a generic manner. We give a brief overview with appropriate pointers to further reading.

The discussion so far suggests to consider strategies as rank-2 polymorphic functions with dedicated support for type case and generic traversal. This support might be delivered by a generative tool approach as covered by the RSF variant of Strafunski (see `http://www.cs.vu.nl/Strafunski/`), or it can be integrated as a language extension with a functional language as described in [39]. Other related proposals for language extensions are discussed in [38, 18].

A kind of rank-2 polymorphism was suggested for Haskell under the name first-class polymorphism [32]. It is an extension of Haskell 98. It restricts explicit quantifiers to be placed on components of datatype constructors. Decidability of type inference is recovered this way, and most of the resulting encoding can be hidden from the strategic programmer. This option is studied in [39]. Since this is a Haskell-centric approach, the expressiveness for traversal and strategy update requires generative support. The corresponding functionality can be captured by a Haskell-class. In this setting, the type of (type-preserving) strategies is defined as a datatype as follows:

$$\textbf{data}\ \mathit{TP}\ m = \mathit{TP}\ (\forall \alpha.\ \mathit{Nonparametricity}\ \alpha\ \Rightarrow\ \alpha\ \rightarrow m\ \alpha)$$

This type makes ultimately clear that we go beyond parametric polymorphism due to the class constraint on the universally quantified constructor component of TP.

A very much different model is discussed in [44]. There, we encode strategies as functions on a universal representation type. The use of the universal representation is completely hidden from the programmer. Such strategies are not directly applied on terms of algebraic datatypes but a dedicated combinator for function application makes sure that explosion to and implosion from the universal representation type is hiddenly performed. The very convenient feature of this model is that it is very easily supported via a generative approach, and in fact, the current distribution of Strafunski is based on this model. The definition of traversal combinators on a universal representation type is trivial. The definition of new traversal combinators can be easily investigated in this manner. As for the implementation of `adhoc` we resort to a dynamic typing discipline [3] (again entirely hidden from the strategic programmer). The generative support for dynamic typing is well in line with the universal representation idea.

To summarize, a typeful approach to strategies most naturally employs rank-2 types, some simple form of intensional polymorphism as for `adhoc`, also a simple form of polytypism to provide generic traversal for all term types. In [39], we argue that this combination is economic when compared to other designs of generic programming. It is shown that the required language extension to allow functional strategies can be reduced to two simple combinators. The alternative to a language extension is to use an existing functional language with additional generative support as in the current distribution of Strafunski for Haskell. The functional incarnation is particularly appropriate to investigate strong typing for strategies, It also turns strategies into first-class citizens since functions are first-class.

5. Object-Oriented Programming

In this section we will explain how an incarnation of the strategic programming idiom can be provided in the object-oriented paradigm. In particular, we will explain the implementation of strategies as *generic visitor combinators* [65] in the class-based language Java. The challenges to be met by this incarnation include blending strategies with reference semantics, encapsulation, inheritance, side-effects, and subtype polymorphism. We will first explain how evaluation and traversal concerns are usually implemented in tangled fashion.

5.1. Encapsulation of data and operations

Objects encapsulate data together with *methods* that operate on them. The data of an object are stored in its *fields* (instance variables). Each

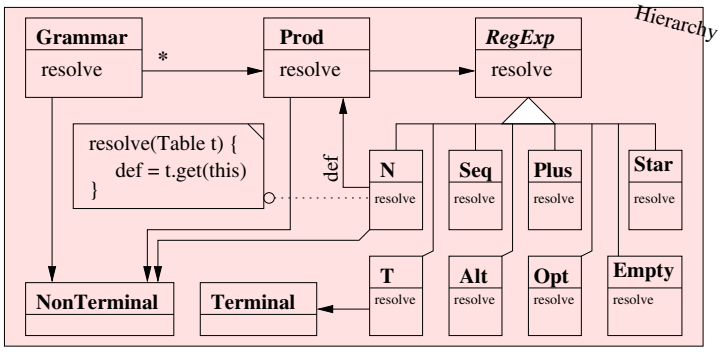

Figure 22. Class hierarchy for the EBNF syntax.

field can hold either a value of a basic type (e.g., integers, strings, booleans), or a *reference* to another object. All access to an object's fields is done via its methods, which are invoked by other objects via references to this object. In class-based languages, the types of objects are *classes* that can be related to each other via *inheritance*. A system of object types is a class hierarchy. During execution, the objects and their references to each other form an object graph, which instantiates the class hierarchy.

For example, Figure 22 shows the UML diagram of a class hierarchy to represent EBNF grammars. In this hierarchy, the context-free non-terminals `Grammar`, `Prod`, and `RegExp` are represented by classes. `RegExp` is an abstract class with concrete subclasses for each of its various alternative productions. The lexical non-terminals `NonTerminal` and `Terminal` are also represented by concrete classes. Note that with respect to the grammar, there is an additional reference `def` from `N` to `Prod`. We will use this reference below.

Behaviour can be added to a class hierarchy by adding methods to its constituent classes. In our EBNF hierarchy, a series of methods are shown that implement *resolution* of the def-use relation between rules and non-terminal symbols. Note that `resolve` methods have been added to almost every class in the hierarchy. Most of these simply call the `resolve` methods of the classes to which they hold a reference. Only the `resolve` methods of `Grammar` and `NonTerminal` are more interesting. The former builds a rule-lookup table and passes it to the `resolve` methods of each of its constituent rules. The latter looks up its corresponding rule in the lookup table and initializes its `def` reference with this rule (as indicated by a note in the figure).

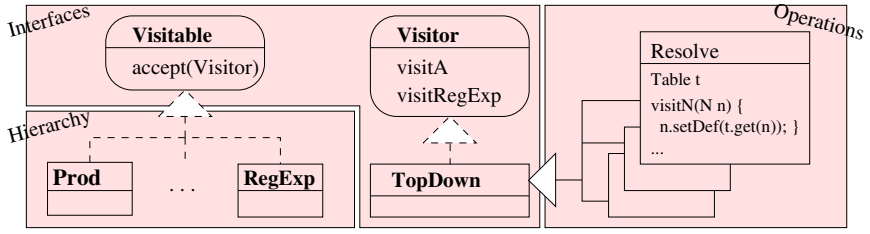

Figure 23. The Visitor pattern

Thus, in the object-oriented paradigm, structured data is represented by object graphs, and, typically, strong coupling exists between such data and the operations on it.

5.2. Tangling of Concerns

The encapsulation of data with methods provides an important innovation over procedural imperative programming, where global variables are a source of confusion and error. On the other hand, such encapsulation has distinct disadvantages for the implementation of concerns that are in some sense 'orthogonal' to the data (cross-cutting), such as evaluation and traversal strategies. Strict enforcement of the encapsulation maxims prohibits separation of control or traversal code from data-specific operations; both are entangled in the methods throughout the class hierarchy.

Such tangling is borne out in several ways by our def-use resolution example. Firstly, grammar *representation* and grammar *resolution* can be seen as distinct concerns, because other EBNF applications might use the same representation, but implement different behaviour. These concerns are now implemented as bits of resolution behaviour smeared out across the entire hierarchy, where each bit of behaviour is strongly coupled with a single class. Secondly, within the resolution concern, we can distinguish *traversal* as one distinct subconcern, and building and using the lookup table as a second subconcern. The implementation of these subconcerns is tangled in the sense that the traversal code can not be used independently of the lookup code, or *vice versa*.

5.3. The Visitor Pattern

Several *design patterns* have been proposed to combat tangling of concerns in object-oriented systems. In particular, the *Visitor* design pattern is used to separate behaviour from the class hierarchy on which it operates [20]. The essentials of the Visitor pattern are depicted in Figure 23. To make a hierarchy of classes *visitable*, one introduces two

interfaces: `Visitable` and `Visitor`. The `Visitor` interface introduces a `visitX` method for each class `X` in the hierarchy. The `Visitable` interface introduces an `accept` which takes a `Visitor` as argument. Each class `X` implements `Visitable`'s `accept` method by invoking the corresponding `visitX` method of the argument visitor. Operations on the class hierarchy can now be constructed by providing implementations of the `Visitor` interface.

Though it is no essential part of the *Visitor* pattern, visitors typically perform some kind of traversal over the object graph that accepts them. For this reason, visitor implementations are usually built by extending a default visitor that implements as fixed traversal scheme. Figure 23 shows a default `TopDown` visitor.

Following the visitor pattern, the resolution of the def-use relation between non-terminals and productions can now be implemented as a visitor `Resolve` which extends `TopDown`. This is shown in the figure. In a sense, the `resolve` methods have been extracted from the hierarcy, and turned into visit methods of these visitors. The resolve methods that perform traversal only have gone to `TopDown`, while the more interesting resolve methods have gone to `Resolve`. The lookup table is no longer passed as an argument, but encapsulated together with the visit methods inside the `Resolve` visitor.

Unfortunately, the visitor pattern succeeds only partially in untangling the various concerns involved in the resolution problem. The `TopDown` visitor can be reused independently from the `Resolve` visitor, but not *vice versa*. Also, redefinition of visit methods in `Resolve` must explicitly restart the traversal on subcomponents, if traversal is to continue. Thus, while cleanly separating behaviour from data, the visitor pattern still tangles traveral and computational behaviour. In fact, the visitor pattern suffers from limitations similar to those of traversal functions in term rewriting (Section 3) and of folds in functional programming (Section 4). The basic actions implemented in an extension of the default visitor can not be reused with visitors that provide different default traversal behaviour. Also, the range of traversal schemes is limited. Providing new ones requires substantial programming effort, or, if default visitors are generated, extension of the generation tool.

5.4. Visitor Combinators

More generally speaking, the following limitations of the *Visitor* pattern can be idenitified. Firstly, visitors resist composition. When separate visitors have been developed to implement various basic operations, they can not be composed to create more complex visitors. Secondly,

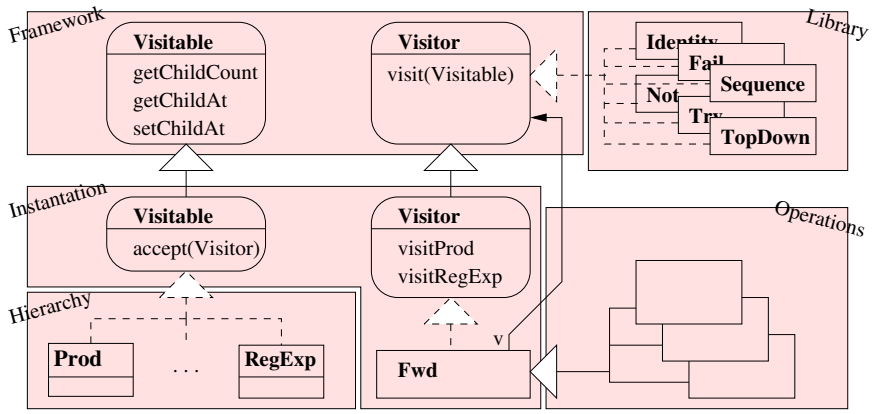

Figure 24. Visitor Combinators

traversal and control are still tangled in the visit methods. As a result, traversal and evaluation strategies can not be easily adapted or reused. Thirdly, though separated from the class-hierarchy, visitors are strongly dependent on it. As a result, they are brittle with respect to changes in the hierarchy, and they can not be reused across hierarchies.

These limitations can be remedied with *generic visitor combinators* [65]. Visitor combinators are small, reusable classes capturing basic functionality that can be composed in different constellations to obtain new functionality. As will become clear, they are the object-oriented incarnation of strategy combinators.

The essence of visitor combinators is conveyed in Figure 24. Note the following differences with respect to the plain visitor pattern:

Generic interfaces The hierarchy-specific `Visitor` and `Visitable` interfaces have spawned supertypes (with the same names, but in different packages). These additional interfaces are *generic*, i.e., independent of the class hierarchy. The generic `Visitor` interface contains a single `visit` method that takes and returns an object of the generic type `Visitable`. The `Visitable` interface contains three methods that allow generic access to an object's subcomponents (children).

`Forward` combinator The visitor `TopDown` has been replaced by the visitor *combinator* `Forward` as default implementation of the hierarchy-specific `Visitor` interface. `Forward` is a combinator, because it takes a generic visitor as argument, which is modeled by a reference `v` initialized at construction time. Each of the visit methods of `Forward` is implemented by *forwarding* to this argument, i.e., by calling `v`'s generic visit method. (The implementation of `Forward` will be detailed below.)

Library Implementations of the generic `Visitor` interface are collected in a library of visitor combinators. Among these is `TopDown`. In contrast to the default visitor of Figure 23, this `TopDown` (i) is supplied with node actions, not through specialzation, but via an argument visitor, (ii) is defined, not by type-specific access to subcomponents, but via the generic term interface of `vistiable`. A more detailed treatment of the content of the library follows below.

The generic interfaces `Visitable` and `Visitor` form a *framework* that is independent of any class hierarchy. A class-hierarchy *instantiates* the framework via the hierarchy-specific `Visitable`, `Visitor`, and `Forward`. By instantiating the framework for a given hierarchy, the library of generic visitor combinators is made available for operating on the hierarchy. Operations on the hierarchy can then be constructed by (i) passing a default visitor to `Forward`, (ii) specializing `Forward`, (iii) passing specialized visitors to generic visitor combinators.

Support is available for both the framework and its instantation. JJTraveler [65, 16] provides a combined visitor combinator framework and library for Java. Its library contains a range of reusable generic combinators. JJForester [36] is a visitor and parser generator that automates instantiation of JJTraveler's framework. It takes a grammar as input and generates a class-hierarchy that instantiates the framework, as indicated in Figure 24.

5.5. Visitors as Strategies

We will now detail the correspondance between generic visitor combinators and strategic programming. Along the way, we will explain how the notions of *datum*, *basic action*, *strategy*, *failure*, *type case*, *type specialization*, *strategy application* etc. are inhabited.

Typing strategies As strategies are data-processing actions, some kind of method may seem to be the obvious candidate to inhabit the notion of *strategy* in the object-oriented world. However, methods (unlike functions in functional programming) are not first-class citizens, and thus we cannot use them directly as strategies. We must *encapsulate* them inside (first-class) objects to be able to pass them around, name them, return them, etc. Generic visitor objects can be used to fulfill this role.

The generic `Visitor` interface of the visitor combinator framework declares a single, fully generic `visit` method, which takes a `Visitable` as argument and returns a `Visitable` as a result:

$$\_.\texttt{visit}(\_) : \texttt{Visitor} \times \texttt{Visitable} \rightarrow \texttt{Visitable}$$

The type of its visit method allows the `Visitor` interface to inhabit the type π of strategies, while the `Visitable` interface is the supertype of all data that can be processed strategically. Accordingly, *strategy application* (without type specialization) is performed by invoking a visitor's visit method with a visitable as argument.

$$\begin{array}{rcl} \pi & \equiv & \texttt{Visitor} \\ \texttt{\_@\_} & \equiv & \texttt{\_.visit(\_)} \end{array}$$

This typing of strategies is somewhat imprecise, because the generic visit method's type does not guarantee that the *concrete* type of its argument is preserved in the result. To strengthen the type of strategies, parameterized types could be used, as available in e.g., Eiffel. The interface `Visitor` would then be parameterized with respect to `Visitable`.

Type specialization The basic actions of the object-oriented paradigm are methods. As stated, these are not first class. The specific visit methods encapsulated in a hierarchy-specific visitor, however, will be viewed as basic actions.

The strategy combinator _[_] of type $\pi \times \tau \rightarrow \alpha_\tau$ turns a strategy into a basic action on data of a specific type τ. In the object-oriented incarnation, _[_] is modeled by a combination of subtype polymorphism and explicit type coercion.

```
v[X]  ≡  X visitX(X x) { return (X) v.visit(x); }
```

Here, the input datum `x` of type `X` is implicitly cast to its supertype `Visitable` when it is supplied as actual argument to `visit`. The result of the method invocation is explicitly cast back down from `Visitable` to `X`. Again, if parameterized types had been used, typing would be more precise, and these casts would be redundant.

Basic combinators As the type π of strategies is modeled by the `Visitor` interface, so can strategies themselves be modeled with implementations of this interface. Figure 25 shows six such implementations (in Java) that correspond exactly to the basic strategy combinators of Figure 1 (page 5).

The `Identity` combinator is implemented to immediately return its argument `Visitable`. This definition already implies a deviation in semantics from the previous incarnations due to Java's reference semantics. If visitable would be *cloned* before being returned, one would stay closer to the other incarnations. The `Fail` combinator is implemented to throw a `VisitFailure` exception whenever it is invoked. Thus, we use the exception mechanism to model strategy success and failure. The `Sequence`

```
class Identity implements Visitor {
  public Visitable visit(Visitable x) { return x; }
}
```

```
class Fail implements Visitor {
  public Visitable visit(Visitable x) {
    throw new VisitFailure();
} }
class VisitFailure extends Exception {}
```

```
class Sequence implements Visitor {
  public Sequence(Visitor v1, Visitor v2) {
    this.v1 = v1; this.v2 = v2;
  }
  Visitor v1, v2;
  public Visitable visit(Visitable x) {
    return v2.visit(v1.visit(x));
} }
```

```
public class IfThenElse implements Visitor {
  public IfThenElse(Visitor c, Visitor t, Visitor f) {
    condition = c; trueCase = t; falseCase = f;
  }
  Visitor condition, trueCase, falseCase;
  public Visitable visit(Visitable x) throws VisitFailure {
    Visitable y;
    try { y = condition.visit(x); }
    catch (VisitFailure vf) { return falseCase.visit(x); }
    return trueCase.visit(y);
} }
```

```
class All implements Visitor {
  public All(Visitor v) { this.v = v; }
  Visitor v;
  public Visitable visit(Visitable x) {
    Visitable result = x;
    for (int i=0; i<x.getChildCount(); i++) {
      result = result.setChildAt(i,v.visit(result.getChildAt(i)));
    }
    return result;
} }
```

```
public class One implements Visitor {
  public One(Visitor v) { this.v = v; }
  protected Visitor v;
  public Visitable visit(Visitable x) throws VisitFailure {
    for (int i = 0; i < x.getChildCount(); i++) {
      try { return x.setChildAt(i,v.visit(x.getChildAt(i))); }
      catch(VisitFailure f) { }
    }
    throw new VisitFailure();
} }
```

Figure 25. Implementation of basic combinators in Java.

```
public class Not extends IfThenElse {
  public Not(Visitor s) {
    super(s,new Fail(),new Identity());
} }
```

```
public class TopDown extends Sequence {
  public TopDown(Visitor v) {
    super(v,null);
    v2 = new All(this);
} }
```

Figure 26. Implementation of defined combinators in Java.

combinator is the first combinator that actually has strategy arguments. These arguments are modeled by instance variables `v1` and `v2`. At creation time, the variables are initialized by the constructor method. The visit method of `Sequence` simply applies its two argument visitor in sequence, where the result of the first is passed as an argument to the second. The `IfThenElse` combinator has three strategy arguments, that are modeled similarly as in the case of `Sequence`. The visit method is implemented to first attempt application of the `condition` visitor inside a try-catch clause. If no `VisitFailure` is raised, the `trueCase` visitor is applied to the intermediate result. If failure does occur, the `falseCase` visitor is applied to the original visitable instead. The `All` and `One` combinators both make use of the term interface offered by `Visitable`. In their visit methods, they loop over all children of the visitable `x`. `All` applies its argument visitor to one child at each iteration. `One` attempt application within a catch-try clause, and exits from the iteration at the first successful application. If no applications are successful, a `VisitFailure` exception is thrown after iteration.

Defined strategies Apart from visitor combinators that represent basic strategy combinators, we can provide implementation of the `Visitor` interface that represent defined combinators. Definitions of `not` and `topdown` are shown in Figure 26. The `Not` visitor is defined by extending the `IfThenElse` visitor, while `TopDown` is defined by extending `Sequence`. In general, defined strategy combinators are implemented by extending the visitor combinator that implements the outermost symbol of their *definiens*. In the case of `Not`, the implementation is completed by a constructor method that invokes the constructor method of the super class (`super`) with the proper arguments. In the case of `TopDown`, a complication arises, because `topdown` is defined *recursively*, and in Java, `this` can only be used *after* `super` has been called. For this reason, `super` is given a `null` reference as second argument, and the corre-

sponding instance variable `v2` is set to its proper value (which references `this`) by a separate assignment.

Following these simple guidelines, all defined combinators of Figure 2 (page 7) can be implemented as generic visitor combinators, and added to the visitor combinator library of Figure 24.

Strategy update The forwarding combinator `Forward` allows the construction of hierarchy specific visitors, by reusing and specializing the behaviour of a generic visitor. Schematically:

```
class MyVisitor extends Forward {
  public MyVisitor() { super(new Identity()); }
  public X visitX(X x) {
    ...
} }
```

Here, the behaviour of `Identity` serves as default, while the behaviour for objects of type `X` is specialized by redefining `Forward`'s `visitX` method. Any visitor can be supplied as default.

Thus, the `Forward` combinator fulfulls the role of the `adhoc` strategy combinator of type $\pi \times \alpha_\tau \rightarrow \pi$. Recall that `Visitor` stands for π. The type `X` $\rightarrow$ `X` of `Forward.visitX(_)` stands for α_τ. Thus, the visitor argument of `Forward` corresponds to the first argument of `adhoc`, and the visit method redefinition stands for the second argument. Of course, the `Forward` combinator may contain several visit method refinements, while the `adhoc` combinator needs to be applied repeatedly if adhoc behaviour is to be specified for several types.

To understand exactly how `Forward` implements the type case behaviour of `adhoc`, we must inspect the implementation of the generic visit method in the definition of `Forward`. This definition is given in Figure 27. Note that the condition of the `if` statement uses run-time type information (RTTI) to test whether the incoming visitable belongs to the given class-hierarchy. If so, the overloaded `accept` method is used to dispatch to the particular `visitX` method of `this` instance of `Forward`. By default, this `visitX` method will invoke the argument visitor `v` (relying on type casts to model type specialization, see above). If the visitable `x` falls outside the given class hierarchy, the argument visitor `v` will also be invoked.

5.6. Object-oriented strategic programming

In this section we will walk through a number of examples of strategic programming with visitor combinators. We start of with a new implementation of the resolution problem.

```
public class Forward implements hierarchy.Visitor, generic.Visitor {
  public Forward(generic.Visitor v) {
    this.v = v;
  }
  generic.Visitor v;
  public generic.Visitable visit(generic.Visitable x) {
    if (x instanceof hierarchy.Visitable) {
      return ((hierarchy.Visitable) x).accept(this);
    } else {
      return v.visit(x);
  } }
  ...
  public A visitA(A a) { return (A) v.visit(a); }
  ...
}
```

Figure 27. Implementation of a `Forward` combinator.

```
public class Resolve extends Forward {
  public Resolve() { super(new Identity()); }
  Hashtable table = new Hashtable();
  public Grammar visit_Grammar(Grammar grammar) {
    Iterator prods = grammar.getProdList().iterator();
    while (prods.hasNext()) {
      Prod p = (Prod) prods.next();
      table.put(p.getNonTerminal(),p);
    }
    return grammar;
  }
  public NonTerminal visit_NonTerminal(NonTerminal nonTerminal) {
    if (table.containsKey(nonTerminal)) {
      nonTerminal.setDefinedBy(table.get(nonTerminal));
    }
    return nonTerminal;
} }
```

Figure 28. A visitor combinator for resolution.

Resolution Using visitor combinators, the traversal concern and the concern of building and using the lookup table can be separated cleanly. The former concern is captured by the combinator `TopDown` (Figure 26). The latter concern is captured by the combinator `Resolve` shown in Figure 28. Note that the visit methods of `Resolve` need not restart the visitor on the children of a visitable to keep the traversal going. The resolution problem is now solved by simply applying the `TopDown` combinator to the `Resolve` combinator as follows:

```
(new TopDown(new Resolve())).visit(grammar);
```

```
public class Visited implements Visitor {
  Set visited = new HashSet();
  public Vistable voidVisit(Visitable x) throws VisitFailure {
    if (!visited.contains(x)) {
      visited.add(x);
      throw new VisitFailure();
    } else {
      return x;
} } }
```

Figure 29. The combinator `Visited` remembers where it has been.

Reachability Once the def-use relation between non-terminals and productions has been resolved, we can use the EBNF object graph to answer reachability questions, such as whether a given non-terminal can be reached from another. Such reachability problems play a major role in the area of program analysis [59].

Since the object graph after resolution is no longer tree-shaped, we need to take care not to let our traversals run into cycles. For this purpose we will use the combinator `Visited` shown in Figure 29. This combinator maintains a set of visitables to remember where it has been. When it encounters a new visitable it adds it to its set, and throws an exception to signal this.

Our reachability problem can be solved correctly using the `onceTD` combinator. It is wiser, however, to use a breadth-first traversal since fewer nodes are likely to be visited before the target node is reached. A combinator `OnceBreadthFirst` is shown in Figure 30. This combinator uses a queue of `pending` visitables to schedule application of its argument strategies. At each node, the argument `v1` is applied first. If successful, the traversal is done, otherwise, `v2` is applied. If `v2` succeeds, its children are added to the queue of pending visitables. Finally, the traversal is continued with the head of the queue. If the queue is emptied before successful application of `v1` has occured, the overall traversal ends with failure.

Finally, a visitor is needed to identify a given visitable. Such a visitor is shown in Figure 31. The `Equals` visitor takes a visitable `y` as argument, which it stores in an instance variable. The visit method employs the `equals` method to test for success.

Using `Visited`, `OnceBreadhtFirst`, and `Equals`, the reachability problem can be solved as indicated by the following code snippet, where `start` and `target` are visitables:

```
(new OnceBreadthFirst(new Equals(target),
                      new Not(new Visited()))).visit(start);
```

```
public class OnceBreadthFirst {
  public OnceBreadthFirst(Visitor v1, Visitor v2) {
    this.v1 = v1;
    this.v2 = v2
  }
  LinkedList pending = new LinkedList();
  Visitor v1;
  Visitor v2;
  public Visitable visit(Visitable x) throws VisitFailure {
    Visitable result = x;
    try { return v1.visit(result);} catch (VisitFailure vf) {}
    try {
      result = v2.visit(result);
      for (int i = 0; i < result.getChildCount(); i++) {
        pending.addLast(result.getChildAt(i));
      }
    } catch (VisitFailure vf) {}
    if (pending.size() != 0) {
      Visitable next = (Visitable) pending.removeFirst();
      next = this.visit(next);
    } else {
      throw new VisitFailure();
    }
    return result;
} }
```

Figure 30. Breadth-first traversal. Traversals stops upon first successful application of `v1`. Traversal is cut off below nodes where `v2` fails.

```
class Equals implements Visitor {
  public Equals(Visitable y) { this.y  = y; }
  Visitable y;
  public Visitable visit(Visitable x) throws VisitFailure {
    if (x.equals(y)) return x;
    else throw new VisitFailure();
} }
```

Figure 31. A visitor that tests for equality to a given visitable.

Thus, a breadth-first traversal is initiated at the `start` visitable. At each node, the `Equals` visitor is applied to determine whether the `target` has been reached, and the negated `Visited` is applied to determine whether the traversal must be cut off. The traversal terminates successfully, if and only if `target` is reachable from `start`.

6. Related Work

Strategic programming as a generic programming idiom has emerged from numerous approaches that contain some of its ingredients. In the foregoing sections, we already gave many pointers, and more can be

found in our previous work underlying the incarnations we described. In this section, we will not repeat these references or attempt to complete the list (which seems quite impossible). Rather we will discuss related approaches that are of special interest in light of our view that strategic programming requires *all* ingredients enumerated in Section 2. The synergy of strategic programming is gone if any ingredient is not present.

Non-strategic programming in XSLT As an(other) example of a non-incarnation, let us consider XSLT [70]. This is the mainstream language for document processing. It certainly allows for generic programming because traversal over the complete document is regulated by the document processing model of XSLT based on template rules. In Figure 32, we show a default rule to descend into all elements (cf. `<xsl:apply-templates select= ...>`) and to apply all templates all over the place. To start with, this approach lacks concision (recall the concise strategic definitions of top-down and bottom-up traversals). At least two ingredients of strategic programming are truly missing. There is no notion of partiality. Templates simply apply or not. Any kind of backtracking model is not available. More seriously, XSLT templates are not first-class. As discussed for basic rewriting and tangled traversal in functional programming, this leads to tangling traversal control and computation. In fact, XSLT even lacks appropriate type-specific operations since one cannot even easily pattern match on elements of a certain structure. This was observed in [28]. The resulting inappropriateness of XSLT for intentionally generic document transformation is discussed and illustrated in [41]. Besides, an XSLT-like language design is hardly accessible for typeful (generic) transformations, although there are some theoretical results on typing essential fragments of the XSLT expressiveness [54].

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http:...">
<xsl:template match=
 "*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
   <xsl:apply-templates select=
    "*|@*|comment()|processing-instruction()|text()"/>
  </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Figure 32. XSLT template to perform deep traversal by default.

```
topdown(G,X,Z) :- call(G,X,Y), all(topdown(G),Y,Z).

all(G,X,Y) :- X =.. [F|Xs], map(G,Xs,Ys), Y =.. [F|Ys].

map(_,[],[]).
map(G,[X|Xs],[Y|Ys]) :- call(G,X,Y), map(G,Xs,Ys).
```

Figure 33. Indication of a Prolog instance.

Strategic programming in Prolog The list of incarnations we have covered is by no means complete. The abstract characterization of strategic programming can also be instantiated in other contexts. It turns out that logic programming as a paradigm is also easily made fit for strategic programming. In fact, we restrict ourselves to Prolog in the sequel. There are a few features of Prolog which make it easy to incarnate strategic programming. Firstly, Prolog's prime support for term representation allows us to construct and destruct terms even generically without any problems. The key concept is here the so-called univ operator (recall "`=..`") very similar to `_#(_)` in Stratego. Secondly, the higher-orderness of strategy combinators and strategy definitions is easily handled via the meta-programming facilities in Prolog (see, for example, higher-order style in [56]). Thirdly, Prolog is basically untyped so that we are initially not hampered as for the definition of challenging combinators like `all` and `one`. In Figure 33, we show a snippet of a Prolog incarnation of strategic programming. At the top, the `topdown` scheme from Figure 2 is rehashed. Sequential composition is encoded this time as conjunction of Prolog. Strategy application is issued by the meta-predicate `call`. As we see in the first parameter position of `all`, strategy expressions are passed around as incomplete Prolog goals. The actual implementation of the `all` combinator indeed simply destructs the term at hand, `map`s over the list of children, and then constructs a term of the same shape. The `map` predicate by itself is in no way original, not even in logic programming. This incarnation is considered in more detail in [42] including issues of typing. A Prolog incarnation of strategic programming is interesting on its own because one might study the additional benefits of logical unification in this context. In fact, the definition of `map` given above will not cope with noninstantiated variables but this is trivially repaired by a guarding literal `not var(X)`.

RhoStratego RhoStratego [18] is a language experiment in which elements from strategic programming are combined with higher-order functional programming. Instead of forging strategic programming onto an

existing language as done for Haskell in Section 4, a new language is designed with strategic programming capabilities.

The main observation underlying RhoStratego is that the case construct in conventional functional programming languages is too restrictive. Instead of combining the choice between several cases via pattern matching into one construct, the case alternatives are made into first-class entities corresponding to rules in a rewriting system. Pattern match failure is also separated from the case construct and attached to individual rules, i.e., if the pattern of a rule does not match, the rule fails. Any expression involving application of a rule can thus fail. The choice between alternatives (the | in ML) is made into a combinator for dealing with pattern match failure.

Another restriction of pattern matching in case constructs is the fact that a pattern should have a constructor at the left end of the application spine. By allowing a variable at the constructor position it becomes possible to write a function that traverses the arguments of a constructor one by one. Thus it is possible to define the primitive traversals `all` and `one` as functions.

Polytypic programming The goals of polytypic programming [52, 29, 4] are very close to our's. Let us first compare the intentions. In polytypic programming, the notion of genericity is very much driven by the idea of *mathematical program construction*, and it is geared towards notions of polymorphic functional programming, type systems, and categorical semantics. By contrast, strategic programming arose from certain, sometimes pragmatic needs in language processing, program transformation and related application areas. These needs include scalability, flexibility, writeability, readability, and a simple implementation. At the technical level, in terms of expressiveness, there are important differences, too. To this end, let us consider typical representatives of polytypic language designs, namely PolyP [29] and Generic Haskell [24, 25, 27]. In PolyP, only single datatypes are considered. This immediately documents that polytypic programming was initially not concerned with heterogeneous data. Meanwhile, systems of datatypes are enabled in Generic Haskell, and different proposals even address the refinement of initially constructor-free definitions by ad-hoc cases for specific types. But even then, generic function update (say, strategy update) is not possible. To enable this essential ingredient of strategic programming would require a concept like type case [22]. Also, it is interesting to notice that rank-2 polytypic definitions were not considered to be an issue so far, although such type schemes are immediately relevant when being faced with the combinator style of strategic programming.

Otherwise, we should point out that the type systems which form the basis of polytypic definitions go beyond our so-far covered incarnations. In Generic Haskell, one can for example, even perform induction on type constructors of a first-order kind.

Aspect-oriented and adaptive programming Aspect-oriented programming [35, 1] (AOP) aims at separation of concerns by allowing features that *cross-cut* a given system's primary modularization to be factored out. Clearly, strategic programming qualifies as an instance of aspect-oriented programming. We briefly discuss differences with other instances of AOP, such as adaptive programming and AspectJ.

In adaptive programming, and its implementation by the Demeter system [48], a notion is present of *traversal strategies* for object structures. Demeter's strategies are high-level descriptions of paths through object graphs in terms of source node, target node, intermediate nodes, and predicates on nodes and edges. These high level descriptions are translated (at compile time) into 'dynamic roadmaps': methods that upon invocation traverse the object structure along a path that satisfies the description. During traversal, a visitor can be applied.

To complement Demeter's declarative strategies, a domain-specific language (DSL) has been proposed to express recursive traversals at a lower, more explicit level [57]. This traversal DSL sacrifices some compactness and adaptiveness in order to gain more control over propagation and computation of results, and to prevent unexpected traversal paths due to underspecification of traversals.

Neither this traversal DSL, not Demeter's strategies provide a full incarnation of strategic programming. For instance, Demeter's strategies are not first-class, not fully generic, and provide very limited means of (traversal) control. The DSL provides more contol, but at the further expense of genericity.

AspectJ [34] is a general purpose aspect-oriented extension of Java. It complements classes with *aspects*: modular units that capture cross-cutting concerns. Aspects employ name-based or property-based *pointcut designators* to pinpoint where in the execution flow the aspect's code should be activated. Aspects and strategies differ in many ways. Most notably, aspects are defined and applied to base functionality that is unaware of the aspects. By contrast, strategy combinators are parameterized building blocks, whose functionality is only activated through composition.

7. Summary

Contribution We have given a programming paradigm-independent characterisation of *strategic programming* – a generic programming idiom where programs are constructed using *strategies*. The kind of genericity enabled by strategies is geared towards processing heterogeneous data, such as parse trees and documents. Crucial ingredients of this genericity are means to traverse into compound data, and to customize generic functionality for a given specific type. We have shown this idiom can be incarnated in different programming paradigms, in particular in the rewriting, functional, and object-oriented paradigms. Only a few well-chosen combinators are required to realize such incarnations. Also, we have demonstrated that to achieve type-safety, the genericity of strategies can be naturally combined with various forms of typing. Figure 34 provides a comparative overview of the various strategic programming incarnations discussed in this paper.

Applications We have used the application domain of grammar engineering throughout this paper. Strategic programming have been applied by us and others to a range of other applications. We will briefly provide pointers to a few:

	TR	FP	OOP	LP
datum	many-sorted term	term of algebraic datatype	object structure	basically untyped term
basic action	rewrite rule	monadic monomorphic function	specific visit method	simple predicate
strategy	rewrite rule	polymorphic monadic function	generic visitor	meta-predicate
strategy application	`<_>_`	function application	`visit` method call	meta-predicate call
subcomponent	subterm	subterm	referenced object	subterm
type specialization	implicit	implicit	type casts	implicit
strategy update	via `choice`	type case	RTTI and dynamic binding	via `choice`
types	liberal type checking	rank-2	subtype polymorphism	basically untyped
partiality	built-in	`MonadPlus`	exceptions	backtracking

Figure 34. Overview of incarnations.

- SSL [61]: The Stratego language has been used to design a *library for language definition and implementation*, including algorithms for free variable collection, substitution, and unification. These are defined in a generic, i.e. language-independent manner by suitably parameterised traversals.
- XT [14]: a bundle of *tools for program transformation*. This bundle is based on Stratego, but uses a number of further packages to cover all aspects of transformation systems, especially packages for parsing and pretty printing.
- Strafunski [44]: A Haskell-based generic programming bundle, including a strategy library, and a pre-compiler. It has been used to specify Java *refactorings*. An even more generic refactoring framework is spelled out in [40], where specifications of program refactorings can be reused for rather different languages. Especially for the latter work the higher-orderness and the strong typing in the functional incarnation proved to be essential.
- ControlCruiser [16]: a *program understanding* tool that reconstructs and visualizes COBOL control flow. The JJForester/JJTraveler architecture has been used for the implementation.
- HSX [31]: an implementation in Stratego of the warm fusion algorithm for *deforestation* of lazy functional programs.
- CodeBoost [5]: a transformation system implemented in Stratego for *domain-specific optimization* of C++ programs in the domain of numeric programming.
- CobolX [69]: an environment based on XT for implementing COBOL *program transformations*. It addresses a few important pragmatic issues of transformation systems, e.g. the preservation of layout during transformation.

Availability The rewriting, the functional and the object-oriented incarnations are supported by corresponding programming environments. You can download the software from the following locations:

- `http://www.stratego-language.org`
- `http://www.cs.vu.nl/Strafunski`
- `http://www.jjforester.org`

These sites also provide documentation, and further information on applications and related research.

References

[1] Special issue on aspect-oriented programming. *Communications of the ACM*, 44(10), Oct. 2001. 52

[2] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. In ACM, editor, *Conference record of POPL'93*, pages 157–170, New York, NY, USA, 1993. ACM Press. 34

[3] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic Typing in Polymorphic Languages. In *Proc. of the 1992 ACM Workshop on ML and its Applications*, pages 92–103, San Francisco, U.S.A., June 1992. Association for Computing Machinery. 35, 36

[4] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In S. Swierstra, P. Henriques, and J. Oliveira, editors, *Advanced Functional Programming, Third International School, AFP '98*, volume 1608 of *LNCS*, pages 28–115, Braga, Portugal, Sept. 1999. Springer-Verlag. 51

[5] O. S. Bagge, M. Haveraaen, and E. Visser. CodeBoost: A framework for the transformation of C++ programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, 2001. 54

[6] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First Workshop on Rewriting Logic and Applications 1996 (WRLA'96). 12

[7] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transforming system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 353–372. Birkhäuser, 1997. 16

[8] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software—Practice & Experience*, 30:259–291, 2000. 18

[9] M. G. J. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, Centrum voor Wiskunde en Informatica, 2001. 14

[10] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Generation of Components for Software Renovation Factories from Context-free Grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000. 3, 14

[11] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996. 14

[12] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985. 34, 35

[13] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. *ACM SIGPLAN Notices*, 34(1):301–312, Jan. 1999. 35

[14] M. de Jonge, E. Visser, and J. Visser. Xt: a bundle of program transformation tools. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. 3, 54

[15] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier, 1990. 10

[16] A. v. Deursen and J. Visser. Building program understanding tools using visitor combinators. In *10th International Workshop on Program Comprehension (IWPC 2002)*. To Appear, 2002. 41, 54

[17] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996. 12, 13

[18] E. Dolstra and E. Visser. First-class rules and generic traversal. Technical Report UU-CS-2001-38, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2001. 35, 50

[19] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *Semantics, Applications and Implementation of Program Generation (SAIG'00)*, Springer Lecture Notes in Computer Science, 2000. 13

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 3, 38

[21] J. A. Goguen and T. W. et. al. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International Computer Science Laboratory, March 1992. 12

[22] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 130–141, New York, NY, USA, 1995. ACM Press. 29, 35, 51

[23] *Haskell 98: A Non-strict, Purely Functional Language*, Feb. 1999. `http://www.haskell.org/onlinereport/`. 24

[24] R. Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, Sept. 1999. Technical report, Universiteit Utrecht, UU-CS-1999-28. 51

[25] R. Hinze. A New Approach to Generic Functional Programming. In T. W. Reps, editor, *Conference record of POPL'00*, pages 119–132, Jan. 2000. 51

[26] R. Hinze. Deriving backtracking monad transformers. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 186–197, N.Y., Sept. 18–21 2000. ACM Press. 28

[27] R. Hinze and S. Jones. Derivable type classes. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001. 51

[28] H. Hosoya and B. Pierce. Regular Expression Pattern Matching. In *POPL '01. Proc. Conference on Principles of Programming Languages*, New York, NY, USA, 2001. ACM Press. 49

[29] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Conference record of POPL'97*, pages 470–482. ACM Press, 1997. 51

[30] J. Jeuring, editor. *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, July 2000. 58, 59

[31] P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):1–34, 2000. 54

[32] M. Jones. First-class polymorphism with type inference. In *Conference record of POPL'97*, pages 483–496, Paris, France, 15–17 Jan. 1997. 35

[33] M. P. Jones. Type Classes with Functional Dependencies. In G. Smolka, editor, *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany*, volume 1782 of *LNCS*, pages 230–244. Springer-Verlag, Mar. 2001. 28

[34] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001. 52

[35] G. Kiczales, J. Lamping, et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, number 1241 in LNCS. Springer Verlag, 1997. 52

[36] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Electronic Notes in Theoretical Computer Science*, 44, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA). 41

[37] R. Lämmel. Grammar Adaptation. In J. Oliveira and P. Zave, editors, *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001. 3

[38] R. Lämmel. Typed Generic Traversals in S'_γ. Technical report, CWI, Aug. 2001. 3, 30, 35

[39] R. Lämmel. First-class Polymorphism With Type Case and Folds over Constructor Applications, Jan. 2002. Draft paper. 29, 35, 36

[40] R. Lämmel. Towards Generic Refactoring. Technical Report cs.PL/0203001, arXiv, Mar.1 2002. Available at `http://www.cwi.nl/~ralf`. 9, 54

[41] R. Lämmel and W. Lohmann. Format Evolution. In J. Kouloumdjian, H. Mayr, and A. Erkollar, editors, *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001. 49

[42] R. Lämmel and G. Riedewald. Prological Language Processing. In M. van den Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, volume 44 of *ENTCS*. Elsevier Science, Apr. 2001. (electronic notes). 50

[43] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001. 3

[44] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In S. Krishnamurthi and C. Ramakrishnan, editors, *Proc. Practical Aspects of Declarative Programming PADL 2002, 4th International Symposium, Portland, OR, USA*, volume 2257 of *LNCS*. Springer-Verlag, Jan. 2002. 3, 9, 30, 36, 54

[45] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In Jeuring [30], pages 46–59. 26, 27, 30

[46] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M. van den Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, volume 44 of *ENTCS*. Elsevier Science, Apr. 2001. (electronic notes). 3, 31

[47] K. J. Lieberherr. *Adaptive Object-Oriented Software—The Demether Method with Propagation Patterns*. PWS Publishing Company, 1995. 3

[48] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, July 1997. 52

[49] B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag. 3

[50] Q. Ma and J. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 7th International Conference, PA, USA, March 1991, Proceedings*, volume 598 of *LNCS*, pages 1–40. Springer-Verlag, New York, NY, 1992. 34

[51] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992. 25

[52] L. Meertens. Calculate Polytypically! In H. Kuchen and S. Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *LNCS*, pages 1–16. Springer-Verlag, 1996. 51

[53] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. FPCA'91*, volume 523 of *LNCS*. Springer-Verlag, 1991. 3, 25

[54] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proc. 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 11–22. ACM, 2000. 49

[55] P.-E. Moreau. A choice-point library for backtrack programming. In *JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic*, 1998. 18

[56] L. Naish. Higher-order logic programming in Prolog. Technical Report 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, Feb. 1996. 50

[57] J. Ovlinger and M. Wand. A language for specifying recursive traversals of object structures. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 70–81, 1999. 52

[58] L. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, Aug. 1983. 3

[59] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11):701–726, Nov. 1998. 47

[60] M. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000. 3

[61] E. Visser. Language Independent Traversals for Program Transformation. In Jeuring [30], pages 86–104. 9, 54

[62] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001. 10, 16

[63] E. Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers. 10

[64] E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proc. of ICFP'98*, pages 13–26, Sept. 1998. 3, 10, 16

[65] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices, OOPSLA 2001 Conference Proceedings*, 36(11):270–282, Nov. 2001. 3, 36, 40, 41

[66] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001. 22

[67] P. Wadler. Theorems for Free! In *Proc. of FPCA'89, London*, pages 347–359. ACM Press, New York, Sept. 1989. 34

[68] P. Wadler. The essence of functional programming. In *Conference record of POPL'92*, pages 1–14. ACM Press, 1992. 27

[69] H. Westra. Configurable transformations for high-quality automatic program improvement. CobolX: a case study. Master's thesis, Utrecht University, Utrecht, The Netherlands, February 2002. 54

[70] XSL Transformations (XSLT) Version 1.0, Nov. 1999. `http://www.w3.org/TR/xslt`. 49