Term Rewriting with Traversal Functions

M.G.J. van den Brand, P. Klint and J.J. Vinju

DRAFT $Revision : 1.132$
9th July 2001

Abstract

Term rewriting is an appealing technique for performing program analysis and program transformation. Tree (term) traversal is frequently used but is not supported by standard term rewriting.

We extend many-sorted, first-order term rewriting with *traversal functions* that automate tree traversal in a simple and type safe way. Traversal functions can be bottom-up or top-down traversals. They can be either sort preserving transformations, or mappings to a fixed sort.

We give examples and describe the semantics and implementation of traversal functions.

1 Introduction

1.1 Background

Program analysis and program transformation usually take the syntax tree of a program as starting point. Operations on this tree can be expressed in many ways, ranging from imperative or object-oriented programs, to attribute grammars and rewrite systems. One common problem that one encounters is how to express the *traversal* of the tree: visit all the nodes of the tree and extract information from some nodes or make changes to certain other nodes.

The kinds of nodes that may appear in a program's syntax tree are determined by the grammar of the language the program is written in. Typically, each rule in the grammar corresponds to a node category in the syntax tree. Real-life languages are described by grammars containing a few hundred up to one thousand grammar rules. This immediately reveals a hurdle for writing tree traversals: a naive recursive traversal function should consider many node categories and the size of its definition will grow accordingly. This becomes even more dramatic if we realize that the traversal function will only do some real work (apart from traversing) for very few node categories.

This problem asks for a form of automation that takes care of the tree traversal itself so that the human programmer can concentrate on the few node categories where real work is to be done. Stated differently, we are looking for a generic way of expressing tree traversals.

From previous experience [4, 5, 6] we know that term rewriting is a convenient, scalable technology for expressing analysis, transformation, and renovation of individual programs and complete software systems. In this paper we address therefore the question how tree traversals can be added to the term rewriting paradigm.

One important requirement is to have a typed design of automated tree traversals, such that terms are always well-formed. Another requirement is to have simplicity of design and use. These are both important properties of many-sorted first-order term rewriting that we want to keep.

1.2 Plan of the paper

In the remainder of this introduction we will discuss general issues in tree traversal (Section 1.3), briefly recapitulate term rewriting (Section 1.4), discuss why traversal functions are necessary in term rewriting (Section 1.5), explain how term rewriting can be extended (Section 1.6), and discuss related work (Section 1.7).

In Section 2 we present traversal functions in ASF+SDF [2, 10] and give various examples. The operational semantics of traversal functions is given in Section 3 and implementation issues are considered in Section 4. Section 5 describes the experience with traversal functions and Section 6 gives a discussion. Two appendices complete the paper. The semantics of traversal functions are described in Appendix A. Some larger examples of traversal functions are presented in Appendix B.

1.3 Issues in Tree Traversal

A simple tree traversal can have three possible goals:

(G1) Transforming the tree, e.g., replacing certain control structures that use `goto`'s into structured statements that use `while` statements.

(G2) Extracting information from the tree, e.g., counting all `goto` statements.

(G3) Extracting information from the tree and simultaneously transforming it, e.g., extracting declaration information and applying it to perform constant folding.

Of course, simple tree traversals can be combined into more complex ones.

The goal of a traversal is achieved by visiting all tree nodes in a certain *visiting order* and applying a *visiting function* to each node.

Some generic properties of left-to-right visiting orders can be found in Figure 1. We distinguish the standard visiting orders preorder (order: root, left subtree, right subtree; also called *top-down*), postorder (order: left subtree, root, right subtree) and endorder (order: left subtree, right subtree, root; also called *bottom-up*). In addition, we distinguish traversals that stop recurring at specific nodes and traversals that always continue until all nodes have been visited. In the table, further recursion stops at the nodes `g`, `3`, and `4`.

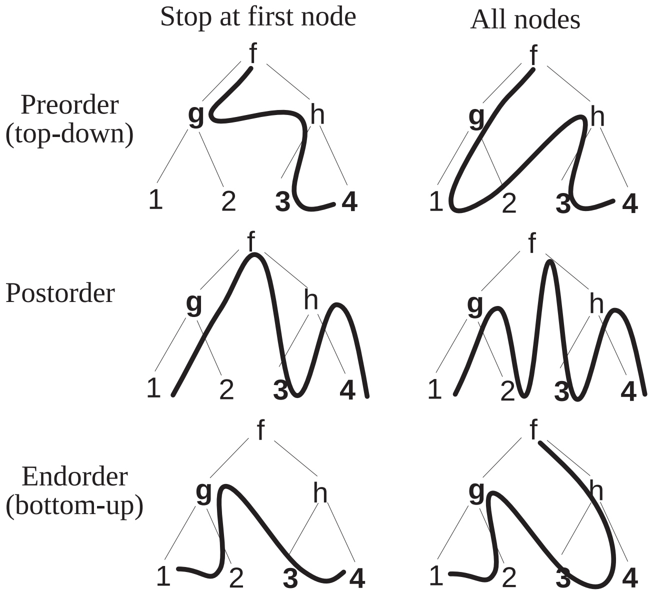

Figure 1: Principle ways of left-to-right tree traversal [20]. Nodes of special interest (g, 3, and 4) are in bold face.

The elementary visiting orders in Figure 1 and some simple combinations of them are obvious candidates for abstraction and automation. Of course, a similar table exists for right-to-left orders but we will not further consider them in this paper since left-to-right orders are used most prominently in the application areas we are interested in.

During a tree traversal, a visiting function should be applied to some or all nodes to achieve the intended effect of the traversal. The type of the visiting function depends on the type of the input nodes, which can be one of the following:

- The nodes are untyped. This is the case in, for instance, Lisp or Prolog. Ease of manipulation is provided at the expense of type safety.
- The nodes are typed and the tree is homogeneous, i.e., all nodes have the same type. This is the case in, for instance, C or Java when nodes are represented by a single data type. As with untyped nodes, homogeneous trees are manipulated easily because every permuatation of nodes is well typed.
- The nodes are typed and the tree is heterogeneous, i.e., nodes may have different types. This is the case in, for instance, C or Java when a separate data type is introduced for representing each construct in a grammar.

In this paper we will only discuss typed trees and focus on the traversal of heterogeneous trees. Other properties of visiting functions are:

- What is the type of their result value?
- What is the type of their other arguments?
- Does the result of the visiting function depend only on the current node that is being visited or does it also use information stored in deeper nodes or even information from a global state?

Obviously, tree traversals are heavily influenced by the type system of the programming language in which they have to be expressed (see Section 1.6 for a further discussion of types).

1.4 A Brief Recapitulation of Term Rewriting

Algorithm 1 An interpreter for innermost rewriting.

```
funct innermost(term, rules) ≡
  (fn, children) := decompose(term)
  children' := nil
  foreach child in children do
    children' := append(children', innermost(child, rules))
  od
  term := compose(fn, children')
  reduct := reduce(term, rules)
  return if reduct = fail then term else reduct fi
.
funct reduce(term, rules) ≡
  foreach rule in rules do
    (lhs, rhs) := decompose(rule)
    bindings := match(term, lhs)
    if bindings ≠ fail then
      return innermost(substitute(rhs, bindings), rules)
    fi
  od
  return fail
.
```

A basic insight in term rewriting is important for understanding the traversal functions described in this paper. Therefore we give a brief recapitulation of innermost term rewriting. For a full account see [19].

A *term* is a prefix expression consisting of constants (e.g., `a` or `12`), variables (e.g., `X`) or function applications (e.g., `f(a, X, 12)`). For simplicity, we will view constants as nullary functions. A *closed term* is a term without variables.

```
module Tree-syntax
imports Naturals
exports
  sorts TREE
  context-free syntax
    NAT             -> TREE
    f(TREE, TREE) -> TREE
    g(TREE, TREE) -> TREE
    h(TREE, TREE) -> TREE
```

Figure 2: SDF grammar for simple tree language.

A *rewrite rule* is a pair of terms $T_1 \to T_2$. Both T_1 and T_2 may contain variables provided that each variable in T_2 also occurs in T_1. A term *matches* another term if it is structurally equal modulo occurrences of variables (e.g., `f(a, X)` matches `f(a, b)`) and results in a *binding* (e.g., `X` is bound to `b`). The bindings resulting from matching can be used for *substitution*, i.e., replace the variables in a term by the values they are bound to.

Given a closed term T and a set of rewrite rules, the purpose of a rewrite rule interpreter is to find a subterm that can be reduced: the so-called *redex*. If subterm R of T matches with the left-hand side of a rule $T_1 \to T_2$, the bindings resulting from this match can be substituted in T_2 yielding T_2'. R is then replaced in T by T_2' and the search for a new redex is continued. Rewriting stops when no new redex can be found and we say that the term is then in *normal form*.

In accordance with the tree traversal orders described earlier, different methods for selecting the redex may yield different results. In this paper we limit our attention to leftmost innermost rewriting in which the redex is searched in a left-to-right, bottom-up fashion.

The operation of a rewrite rule interpreter is shown in more detail in Algorithm 1. The functions *match* and *substitute* are not further defined, but have a meaning as just sketched. The functions *decompose* and *compose* manipulate terms and also rules, and *append* appends an element to the end of a list. Observe how function *innermost* first reduces the children of the current term before attempting to reduce the term itself. This realizes a bottom-up traversal of the term. Also note that if the reduction of the term fails, it returns itself as result. The function *reduce* performs, if possible, one reduction step. It searches all rules for a matching left-hand side and, if found, the bindings resulting from the successful match are substituted in the corresponding right-hand side. This modified right-hand side is then further reduced with innermost rewriting.

In Section 3 we will extend Algorithm 1 to cover traversal functions as well.

1.5 Why Traversal Functions in Term Rewriting?

Rewrite rules are very convenient to express transformations on trees and one may wonder why traversal functions are needed at all. We will clarify this by way of simple trees containing natural numbers. A tree is either a natural number or it is constructed with one of the binary constructors `f`, `g` or `h`. Figure 2 displays an SDF grammar for this tree language.

Transformations on these trees can now be defined easily. For instance, if we want to replace all occurrences of `f` by `h`, then the following rule suffices:

```
[t1] f(T1, T2) = h(T1, T2)
```

Applying this rule to the term `f(f(g(1,2),3),4)` leads to a normal form in two steps (using innermost reduction):

```
f(f(g(1,2),3),4) -> f(h(g(1,2),3),4) -> h(h(g(1,2),3),4)
```

Similarly, if we want to replace all subtrees of the form `f(g(T1, T2), T3)` by `h(T1, h(T2, T3))`, we can achieve this by the single rule:

```
[t2] f(g(T1, T2), T3) = h(T1, h(T2, T3))
```

If we apply this rule to `f(f(g(1,2),3),4)` we get a normal form in one step:

```
f(f(g(1,2),3),4) -> f(h(1,h(2,3)),4)
```

Note how in both cases the standard (innermost) reduction order of the rewriting system takes care of the complete traversal of the term. This elegant approach has, however, three severe limitations:

- If we want to have the combined effect of rules `[t1]` and `[t2]`, we get unpredictable results, since the two rules interfere with each other: the combined rewrite system is said to be *non-confluent*. Applying the above two rules to our sample term `f(f(g(1,2),3),4)` may lead to either `h(h(g(1,2),3),4)` or `f(h(1,h(2,3)),4)`, depending on whether `[t1]` or `[t2]` is applied first.

  One solution to this problem is to introduce new function symbols that eliminate the interference between rules. If we introduce the symbols `trafo1` and `trafo2` for this purpose, we can explicitly control the outcome of the combined transformation by the order in which we apply `trafo1` and `trafo2` to the initial term. The downside of this approach is that rewrite rules are needed to define `trafo1` and `trafo2` as shown in Figure 3. Observe that equations `[2]` and `[5]` in the figure correspond to the original equations `[t1]` and `[t2]`, respectively. The other equations are just needed to define the tree traversal. This definition requires explicit knowledge of *all* rules in the grammar (in this case the definitions of `f`, `g` and `h`). In this example, the number of rules per function is directly related to the size of the Tree language. For large grammars this is clearly undesirable.

```
module Tree-trafo12
imports Tree-syntax
exports
context-free syntax
  trafo1(TREE) -> TREE
  trafo2(TREE) -> TREE
equations
[0] trafo1(N)         = N
[1] trafo1(f(T1, T2)) = h(trafo1(T1), trafo1(T2))
[2] trafo1(g(T1, T2)) = g(trafo1(T1), trafo1(T2))
[3] trafo1(h(T1, T2)) = h(trafo1(T1), trafo1(T2))

[4] trafo2(N)         = N
[5] trafo2(f(g(T1,T2),T3)) =
         h(trafo2(T1), h(trafo2(T2), trafo2(T3)))
[6] trafo2(g(T1, T2)) = g(trafo2(T1), trafo2(T2))
[7] trafo2(h(T1, T2)) = h(trafo2(T1), trafo2(T2))
```

Figure 3: Definition of `trafo1` and `trafo2`.

- This way of expressing transformations does allow neither passing additional parameters to the transformation nor returning additional results.
- In ordinary (typed) term rewriting only type-preserving rewrite rules are allowed, i.e., the type of the left-hand side of a rewrite rule has to be equal to the type of the right-hand side of that rule. Subterms can only be replaced by subterms of the same type, thus enforcing that the complete term remains well-typed. In this way, one cannot express non-type-preserving traversals such as the (abstract) interpretation or analysis of a term. In such cases, the original type (e.g., integer expressions of type `EXP`) has to be translated into values of another type (e.g., integers of type `INT`).

The traversal functions presented in this paper, solve these three problems.

1.6 Extending Term Rewriting with Traversal Functions

We take a many-sorted, first-order, term rewriting language as our point of departure. Suppose we want to traverse syntax trees of programs written in a language L, where L is described by a grammar consisting of n grammar rules.

A specific tree traversal will then be described by m ($m <= n$) rewrite rules, covering all possible constructors that may be encountered during a traversal of the syntax tree. The value of m largely depends on the structure of the grammar and the specific traversal problem. Typically, a significant subset of all constructors needs to be visited resulting in tenths to hundreds of rules that have to be written for a given large grammar and a specific traversal function.

The question now is: how can we avoid writing these m rewrite rules? There are several general approaches to this problem:

- One solution is the use of higher-order term rewriting [18, 12, 16]. This allows writing patterns in which the context of a certain language construct can be captured by a (higher-order) variable thus eliminating the need to explicitly handle the constructs that occur in that context. We refer to [17] for a simple example of higher-order term rewriting.

- One can extend the rewriting language with *generic strategy primitives* that enable the formulation of arbitrary tree traversals. Such primitives could, for instance, be the sequential traversal of two subtrees or traversing a subtree satisfying a certain predicate.

- Another option is to extend the rewriting language with a fixed set of *built-in tree traversals*. For instance, a preorder traversal could be offered as primitive.

Higher-order term rewriting is a very powerful mechanism, which can be used to express type-safe tree traversals. On the other hand, it introduces complex semantics and implementation issues. So, for tree traveral alone, this technique seems rather heavy weight.

Providing traversal primitives is hard to combine with static typing mechanisms. Having types is relevant for static type checking, program documentation, and program comprehension. It is also beneficial for efficient implementation and optimization. However, an untyped approach gives greater flexibility.

In general, when allowing generic traversal primitives, rules cannot be typed statically using any conventional typing system. Whether general strategy primitives can be provided in a type-safe manner is a matter of ongoing research. See [28, 22] for a further discussion of this topic.

In ordinary (typed) term rewriting only type-preserving rewrite rules are allowed, i.e., the type of the left-hand side of a rewrite rule has to be equal to the type of the right-hand side of that rule. Subterms can only be replaced by subterms of the same type, thus enforcing that the complete term remains well-typed.

By extending ordinary term rewriting with built-in tree traversals, one can provide some primitives that allow type-preserving and even a class of non-type-preserving traversals in a type-safe manner. This is the main subject of the current paper.

In this paper we will extend the many-sorted, first-order term rewriting language ASF+SDF with three types of traversal primitives: transformers, accumulators, and accumulating transformers. As we shall see, each use of these traversal primitives can be statically typed.

	Untyped	**Typed**
Strategy primitives	Stratego [29]	ELAN [3]
Built-in strategies	Renovation Factories [7]	Traversal Functions

Figure 4: Classification of traversal approaches.

1.7 Related Work

1.7.1 Directly Related Work

We classify four directly related approaches in Figure 4 and discuss them below.

ELAN [3] is a language of many-sorted, first-order, rewrite rules extended with a strategy language that controls the application of individual rewrite rules. Its strategy primitives (e.g., don't know choice, don't care choice) allow formulating non-deterministic computations. Currently, ELAN does not support generic tree traversals since they are not easily fitted in with ELAN's type system.

Stratego [29, 30] is an untyped term rewriting language that provides user-defined strategies. Among its strategy primitives are several generic traversal operators that allow the definition of all traversal directions of Figure 1 in an abstract manner. Therefore, tree traversals are first class objects that can be reused. Stratego provides a library with all kinds of named traversal strategies such as, for instance, `bottomup(s)`, `topdown(s)` and `innermost`.

Transformation Factories [7] are an approach in which ASF+SDF rewrite rules are generated from language definitions. After the generation phase, the user instantiates a visiting function by overriding default traversal behavior. Note that the generated rewrite rules are well-typed.

Transformation Factories provide two kinds of traversals: transformers and analyzers. A transformer transforms the node it visits. An analyzer is the combination of a traversal, a combinator function and a default value. The generated traversal function reduces each node to the default value, unless the user overrides it. The combinator combines the results in an innermost manner. An analyzer is a higher-order function with a combinator as function parameter. This higher-order behavior has to be simulated in the first-order world of ASF+SDF. As a consequence, this solution can result in type-unsafe rewriting.

1.7.2 Relation with Traversal Functions

Traversal functions emerged from our experience in writing program transformations for real-life languages in ASF+SDF. Both Stratego and Transformation Factories are attempts to remedy the problems that we encountered.

Stratego and ELAN extend term rewriting with user-defined strategies, but we are more conservative and extend first-order term rewriting only with a fixed set of traversal primitives. The result is simple, can be statically typed and can be implemented efficiently.

Compared to Transformation Factories (which most directly inspired our traversal functions), we provide a slightly different set of traversal functions and reduce the notational overhead. More important is that we provide a fully typed approach. At the level of the implementation, we do not generate ASF+SDF rules, but we have incorporated traversal functions in the standard evaluator of ASF+SDF. As a result, execution is more efficient and specifications are more readable, since users are not confronted with generated rewrite rules.

1.7.3 Other Related Work

Apart from the directly related work already mentioned, we summarize related work in *functional languages*, *object-oriented languages* and *attribute grammars*.

Functional languages. The prototypical traversal function in the functional setting are the functions `map`, `fold` and relatives. `map` takes a tree and a function as argument and applies the function to each node of the tree. However, problems arise as soon heterogeneous trees have to be traversed. One solution to this problem are fold algebras as described in [23]: based on a language definition traversal functions are generated in Haskell. A tool generates generic folding over algebraic types. The folds can be updated by the user.

Object-oriented languages. The traversal of arbitrary data structures is captured by the *visitor design pattern* described in [13]. Typically, a fixed traversal order is provided as framework with default behavior for each node kind. This default behavior can be overruled for each node kind. A related implementation of the visitor pattern is JJForester [21]: a tool that generates Java class structures from SDF language definitions. The generated classes implement generic tree traversals that can be overridden by the user. The technique is related to generating traversals from language definitions as in Transformation Factories, but is tailored to and profits from the object-oriented programming paradigm.

Attribute grammars [1]. The approaches described so far provide an operational view on tree traversals. Attribute grammars provide a declarative view: they extend a syntax tree with *attributes* and *attribute equations* that define relations between attribute values. Attributes get their values by solving the attribute equations; this is achieved by one or more traversals of the tree. For attribute grammars tree traversal is an issue for the implementation and not for the user. Attribute grammars are convenient for expressing analysis on a tree but they have the limitation that tree transformations cannot be easily expressed. However, higher-order attribute grammars [31] remedy this limitation to a certain extent. A new tree can be constructed in one of the attributes which can then be passed on as an ordinary tree to the next higher-order attribute function.

Combining attribute grammars with object orientation. JastAdd [15] is very recent work in the field of combining reference attribute grammars [14] with visitors and class weaving. The attribute values in reference attributes may be references to other nodes in the tree. The implicit tree traversal mechanism for attribute evaluation is combined with the explicit traversal via visitors. This is convenient for analysis purposes but it does not solve the transformation problem in a nice manner.

2 Traversal functions in ASF+SDF

We want to automate tree traversal in the many-sorted, first-order term rewriting language ASF+SDF [2, 10]. ASF+SDF uses context-free syntax for defining the signature of terms. As a result, terms can be written in arbitrary user-defined notation. The context-free syntax is defined in SDF[1]. Terms are used in rewrite rules defined in ASF[2]. For the purpose of this paper, the following features of ASF are relevant:

- Unconditional and conditional rules. Conditions come in three flavors: equality between terms, inequality between terms, and so-called assignment conditions that introduce new variables.
- Default rules that are tried only if all other rules fail.
- Terms are normalized by leftmost innermost reduction.

The idea of traversal functions is as follows. First, the user has to declare the signature of a visiting function. This is an ordinary declaration but it is explicitly labeled with transformer (`trafo`) or accumulator (`accu`) (or both) to indicate that this is a visiting function. We call such a labeled function a *traversal function* since, from the user's perspective they automatically traverse a term. Next, the user has to give rewrite rules for the nodes that the traversal function will visit.

If, during innermost rewriting, a traversal function appears as outermost function symbol of a redex, then that function will first be used to traverse the redex before further reductions occur.

Conceptually, a traversal function is a shorthand for a possibly large set of rewrite rules. For every traversal function a set of rewrite rules can be calculated that implements both the traversal and the visiting function. This a nice way of defining the semantics of traversal functions, which is explained in Appendix A.

From Section 1.6 we have learned that typing systems cannot easily cope with all generic principles of traversals. So, the question is what traversals we can provide in our fully typed setting. We offer three types of traversal functions (Section 2.1) and two types of visiting strategies (Section 2.2) which we now

[1]Syntax Definition Formalism.
[2]Algebraic Specification Formalism.

discuss in order. In Section 2.3 we present examples of traversal functions. The merits and limitations of this approach are discussed in Section 6.

2.1 Kinds of Traversal Functions

We distinguish three kinds of traversal functions, defined as follows.

Transformer: a sort-preserving transformation that will traverse its first argument. Possible extra arguments may contain additional data that can be used (but not modified) during the traversal. A transformer is declared as follows:

$f(S_1, \ldots, S_n) \rightarrow S_1$ {`traversal(trafo)`}

Because a transformer always returns the same sort, it is type-safe. A transformer is used to transform a tree and implements goal (G1) discussed in Section 1.3.

Accumulator: a mapping of all node types to a single type. It will traverse its first argument, while the second argument keeps the accumulated value. An accumulator is declared as follows:

$f(S_1, S_2, \ldots, S_n) \rightarrow S_2$ {`traversal(accu)`}

After each application of an accumulator, the accumulated argument is updated. The next application of the accumulator, possibly somewhere else in the term, will use the *new* value of the accumulated argument. In other words, the accumulator acts as a global, modifiable, state during the traversal.

An accumulator function never changes the tree, only its accumulated argument. Furthermore, the type of the second argument has to be equal to the result type. The end-result of an accumulator is the value of the accumulated argument. By these restrictions, an accumulator is also type-safe for every instantiation.

An accumulator is meant to be used to extract information from a tree and implements goal (G2) discussed in Section 1.3.

Accumulating transformer: a sort preserving transformation that accumulates information while traversing its first argument. The second argument maintains the accumulated value. The return value of an accumulating transformer is a tuple consisting of the transformed first argument and accumulated value. An accumulating transformer is declared as follows:

$f(S_1, S_2, \ldots, S_n) \rightarrow S_1 \# S_2$ {`traversal(accu, trafo)`}

An accumulating transformer is used to simultaneously extract information from a tree and transform it. It implements goal (G3) discussed in Section 1.3.

Transformers, accumulators, and accumulating transformers may be overloaded to obtain visitors for heterogeneous trees. Their optional extra arguments can carry information down and their defining rewrite rules can extract information from their children by using conditions. So we can express analysis and transformation using non-local information rather easily.

2.2 Visiting strategies

Having these three types of traversals, they must be provided with visiting strategies. Visiting strategies determine the order of traversal and the "depth" of the traversal. We provide the following two strategies for each type of traversal:

Bottom-up: the traversal visits *all* the subtrees of a node where the visiting function applies in an *bottom-up* fashion. The annotation `bottom-up` selects this behavior. A traversal function without an explicit indication of a visiting strategy also uses the bottom-up strategy.

Top-down: the traversal visits the subtrees of a node in an top-down fashion and stops recurring at the first node where the visiting function applies and does not visit the subtrees of that node. The annotation `top-down` selects this behavior.

A transformer with a `bottom-up` strategy resembles standard innermost rewriting; it is sort preserving and bottom-up. It is as if a small rewriting system is defined within the context of a transformer function. The difference is that a transformer function inflicts one reduction on a node, while innermost reduction normalizes a node completely.

The `top-down` strategy is rather powerful because it stops, allowing the user to continue the traversal under certain conditions.

We will discuss the merits and limitations of supplying only these three types in combination with these two strategies in Section 6.

2.3 Examples of transformers

In this and the following subsections, we give examples of transformers, accumulators, and accumulating transformers. All examples use the tree language introduced earlier in Figure 2.

2.3.1 The trafo example from the introduction revised

Recall the definition of the transformations `trafo1` and `trafo2` in the introduction (Figure 3). They looked clumsy and cluttered the intention of the transformation completely. Figure 5 shows how to express the same transformations using two traversal functions.

Observe how these two rules resemble the original rewrite rules. There is, however, one significant difference: these rules are only applicable in the context of the traversal function. This function disappears after each application, so normalization using these equations must be specified explicitly.

```
module Tree-trafo12-trav
imports Tree-syntax
exports
context-free syntax
  trafo1(TREE) -> TREE {traversal(trafo)}
  trafo2(TREE) -> TREE {traversal(trafo)}
equations

[tr1'] trafo1(f(T1, T2))      = h(T1, T2)
[tr2'] trafo2(f(g(T1,T2),T3)) = h(T1, h(T2, T3))
```

Figure 5: `Trafo1` and `trafo2` from Figure 3 revised using traversal functions.

```
module Tree-inc
imports Tree-syntax
exports
  context-free syntax
    inc(TREE) -> TREE {traversal(trafo)}
equations
[1] inc(N) = N + 1
```

Figure 6: The transformer `inc` increments each number in a tree.

2.3.2 Increment the numbers in a tree

The specification in Figure 6 shows the transformer `inc`. Its purpose is to increment all numbers that occur in a tree. The results of applying `inc` to a sample tree are shown in Figure 7. To better understand this example, we follow the rewrite steps when applying `inc` to the tree `f(g(1,2),3)`:

```
inc(f(g(1,2),3)) ->
f(inc(g(1,2)), inc(3)) ->
f(g(inc(1),inc(2)), 4) ->
f(g(2,3),4)
```

In Section A we will give a translation-based semantics for traversal functions. An expanded version of the `inc` specification will be given in Figure 25.

2.3.3 Increment the numbers in a tree (with parameter)

The specification in Figure 8 shows the transformer `incp`. Its purpose is to increment all numbers that occur in a tree with a given parameter value. The results of applying `incp` to a sample tree are shown in Figure 9. Observe that the *first* argument of `incp` is traversed and that the second argument is a value that is carried along during the traversal.

If we follow the rewrite steps for `incp(f(g(1,2),3), 7)`, we get:

term	normal form

`inc(`
`  f( g( f(1,2), 3 ),`
`     g( g(4,5), 6 )`
`   )`
`)`
	
	`  f( g( f(2,3), 4 ),`
	`     g( g(5,6), 7 )`
	`   )`

Figure 7: `inc` applied to sample tree.

```
module Tree-incp
imports Tree-syntax
exports
  context-free syntax
    incp(TREE, NAT) -> TREE {traversal(trafo)}
equations
[1] incp(N1, N2) = N1 + N2
```

Figure 8: The transformer `incp` increments each number in a tree with a parameter value.

```
incp(f(g(1,2),3), 7) ->
f(incp(g(1,2), 7), incp(3, 7)) ->
f(g(incp(1, 7),incp(2, 7)), 10) ->
f(g(8,9),10)
```

2.3.4 Replace function symbols

A common problem in tree manipulation is the replacement of function symbols. In the context of our tree language we want to replace occurrences of symbol `g` by a new symbol `i`. Replacement can be defined in many flavors. Here we only show three of them: full replacement that replaces all occurrences of `g`, shallow replacement that only replaces occurrences of `g` that are closest to the root of tree, and deep replacement that only replaces occurrences that are closest to the leaves of the tree.

term	normal form

`incp(`
`  f( g( f(1,2), 3 ),`
`     g( g(4,5), 6 )`
`   ),`
`   7`
`)`
	
	`  f( g( f( 8, 9), 10 ),`
	`     g( g(11,12), 13 )`
	`   )`

Figure 9: `incp` applied to sample tree.

```
module Tree-frepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)  -> TREE
   frepl(TREE)    -> NAT {traversal(trafo)}
equations

[1] frepl(g(T1, T2)) = i(T1, T2)
```

Figure 10: The transformer `frepl` replaces all occurrences of `g` by `i`.

term	**normal form**

`frepl(`
`  f( g( f(1,2), 3 ),`
`     g( g(4,5), 6 )`
`   )`
`)`
	
	`  f( i( f(1,2), 3 ),`
	`     i( i(4,5), 6 )`
	`   )`

Figure 11: `frepl` applied to sample tree.

Full replacement is defined in Figure 10; an example is shown in Figure 11. Since we did not specify a visiting strategy, bottom-up reduction will be used. This will ensure that all nodes in the tree will be visited.

Shallow replacement is defined in Figure 12; an example is shown in Figure 13. In this case, replacement stops at each outermost occurrence of `g`.

Observe that top-down traversal applies the traversal function at an applicable outermost node and does not visit the subtrees of that node. However, the right-hand side of a defining equation of the traversal function may contain recursive applications of the traversal function itself! In this way, one can even force traversal behavior that visits all nodes in a tree. To illustrate this, we show in Figure 14 a version of full replacement using top-down reduction. Note how

```
module Tree-srepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)  -> TREE
   srepl(TREE)    -> NAT {traversal(trafo, top-down)}
equations
[1] srepl(g(T1, T2)) = i(T1, T2)
```

Figure 12: The transformer `srepl` replaces shallow occurrences (i.e., occurrences close to the root) of `g` by `i`.

term	normal form

`srepl(`
`  f( g( f(1,2), 3 ),`
`     g( g(4,5), 6 )`
`   )`
`)`
	
	`  f( i( f(1,2), 3 ),`
	`     i( g(4,5), 6 )`
	`   )`

Figure 13: `srepl` applied to sample tree.

```
module Tree-frepl2
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)  -> TREE
   frepl2(TREE)    -> NAT {traversal(trafo, top-down)}
equations

[1] frepl2(g(T1, T2)) = i(frepl2(T1), frepl2(T2))
```

Figure 14: The transformer `frepl2` replaces all occurrences of `g` by `i`.

two recursive applications of `frepl2` occur in the right-hand side of equation `[1]`. An example is shown in Figure 15.

We use this combination of a top-down strategy with recursive applications of the traversal function to define deep replacement as shown in Figure 16; an example is shown in Figure 17. Here, recursive applications of `drepl` are used in the *conditions* of the equation to test whether replacements take place in any of the children of the current node. If not, the current node is a candidate for innermost replacement.

2.4 Examples of accumulators

Sofar, we have only shown examples of transformers. In this section we will give two examples of accumulators.

term	normal form

`frepl2(`
`  f( g( f(1,2), 3 ),`
`     g( g(4,5), 6 )`
`   )`
`)`
	
	`  f( i( f(1,2), 3 ),`
	`     i( i(4,5), 6 )`
	`   )`

Figure 15: `frepl2` applied to sample tree.

```
module Tree-drepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)  -> TREE
   drepl(TREE)    -> NAT {traversal(trafo, top-down)}
equations
    drepl(T1) = T1, drepl(T2) = T2
[1] ===========================
    drepl(g(T1, T2)) = i(T1, T2)
```

Figure 16: The transformer `drepl` replaces deep occurrences (i.e., occurrences close to the leaves) of `g` by `i`.

term	normal form

`drepl(`
`  f( g( f(1,2), 3 ),`
`     g( g(4,5), 6 )`
`   )`
`)`
	
	`f( i( f(1,2), 3 ),`
	`   g( i(4,5), 6 )`
	` )`

Figure 17: `drepl` applied to a sample tree.

2.4.1 Add the numbers in a tree

The first problem we want to solve is computing the sum of all numbers that occur in a tree. The accumulator `sum` in Figure 18 solves this problem. Note that in equation `[1]` variable `N1` represents the current node (a number), while variable `N2` represents the sum that has been accumulated so far (also a number). The results of applying `sum` to a sample tree are shown in Figure 19.

2.4.2 Count the nodes in a tree

The second problem is to count the number of nodes that occur in a tree. The accumulator `cnt` shown in Figure 20 does the job. The results of applying `cnt`

```
module Tree-sum
imports Tree-syntax
exports
context-free syntax
   sum(TREE, NAT) -> NAT {traversal(accu)}
equations
[1] sum(N1, N2) = N1 + N2
```

Figure 18: The accumulator `sum` computes the sum of all numbers in a tree.

term	normal form

`sum(`
`  f( g( f(1,2), 3 ),`
`     g( g(4,5), 6 )`
`   ),`
`  0`
`)`
	`21`

Figure 19: `sum` applied to sample tree.

```
module Tree-cnt
imports Tree-syntax
exports
context-free syntax
  cnt(TREE, NAT) -> NAT {traversal(accu)}
equations
[1] cnt(T, N) = N + 1
```

Figure 20: The accumulator `cnt` counts the nodes in a tree.

to a sample tree are shown in Figure 21.

2.5 Examples of accumulating transformers

We conclude our series of examples with one example of an accumulating transformer.

2.5.1 Multiply by position in tree

Our last problem is to determine the position of each number in a preorder traversal of the tree and to multiply each number by its position. This is achieved by the accumulating transformer `pos` shown in Figure 22. The general idea is to accumulate the position of each number during the traversal and to use it as a multiplier to transform numeric nodes. The results of applying `pos` to a sample tree are shown in Figure 23.

term	normal form

`cnt(`
`  f( g( f(1,2), 3 ),`
`     g( g(4,5), 6 )`
`   ),`
`  0`
`)`
	`11`

Figure 21: `cnt` applied to sample tree.

```
module Tree-pos
imports Tree-syntax
exports
context-free syntax
  pos(TREE, NAT) -> NAT {traversal(accu, trafo)}
equations
[1] pos(N1, N2) = <N1 * N2, N2 + 1>
```

Figure 22: The accumulating transformer `pos` multiplies each number by its position in the tree.

term	normal form
`pos( f( g( f(1,2), 3 ), g( g(4,5), 6 ) ), 0 )`	`<f( g( f(0,2), 6 ), g( g(12,20), 30 ) ), 6>`

Figure 23: `pos` applied to sample tree.

3 Operational Semantics

In this section we show an operational semantics for traversal functions. The reader is referred to Appendix A for a different style of semantics, namely expressing traversal functions in terms of normal rewriting systems. The semantics in this section is better suited as a reference for implementation. These two semantics are expected to be equivalent, but we give no formal proof of this.

We start with normal innermost rewriting as depicted earlier in Algorithm 1 (see Section 1.4), but we switch to rewriting with traversals when a traversal function is encountered. The switch statement in Algorithm 2 detects a traversal function and turns over control to a function called *traverse*, instead of calling *reduce*. This function is shown in Algorithm 3. It initiates the traversal with different parameters for each kind of traversal function. Recall that the input term is of the form $trfn(T_1, T_2, ..., T_n)$ $(n >= 1)$ where *trfn* is a traversal function, T_1 is the term to be traversed, T_2 is the (optional) accumulator argument, and $T_3, ..., T_n$ are the (optional) remaining arguments. Actual traversal is done by *td-or-bu* ("top-down or bottom-up") that uses either *top-down* or *bottom-up* depending on the traversal strategy of *trfn*. The arguments of *td-or-bu* are determined by the different kinds of traversals.

We apply the traversal function by reusing the *reduce* function from the basic innermost rewriting algorithm. It is applied either before or after traversing the children, depending on the traversal strategy (`bottom-up` or `top-down`).

The traversal of children in function *visit-children* takes into account that the accumulated value must be passed on between each child. Note that in case

Algorithm 2 An extended interpreter for innermost rewriting.

funct *innermost*(*term*, *rules*) $\equiv$
  (*fn*, *children*) := *decompose*(*term*)
  children$'$:= **nil**
  foreach *child* **in** *children* **do**
    children' := *append*(*children'*, *innermost*(*child*, *rules*))
  od
  term := *compose*(*fn*, *children'*)
  reduct := **switch** *function-type*(*fn*)
      case `traversal` : *traverse*(*term*, *rules*)
      case `normal` : *reduce*(*term*, *rules*);
  return **if** *reduct* = **fail** **then** *term* **else** *reduct* **fi**
.

of a transformer, this accumulated value is ignored by passing always the value **nil**.

After a successful application of a user-defined rule, the function *make-reduct* decides what to do with the reduct depending on the type of traversal:

Transformer	The reduct replaces the redex.
Accumulator	The reduct replaces the accumulated argument of the traversal function while the redex remains unchanged.
Accumulating transformer	The reduct is a tuple. The first element of the tuple replaces the redex, while the second element replaces the accumulated argument of the traversal function.

Finally, when we return from the traversal, the top level function *traverse* returns a different normal form for each type of traversal function:

Transformer	The transformed term.
Accumulator	The accumulated argument.
Accumulating transformer	A tuple of the transformed term and the accumulated argument.

4 Implementation issues

The actual implementation of traversal functions in ASF+SDF consists of three parts:

- Parsing the user-defined rules of a traversal function.
- An interpreter-based implementation of traversal functions.
- A compilation scheme for traversal functions.

Algorithm 3 An interpreter for traversal functions.

```
funct traverse(term, rules) ≡
  (trfn, args) := decompose(term); term := head(args); args := tail(args)
  switch traversal-kind(trfn)
    case "trafo" :
          (reduct, nil) := td-or-bu(trfn, term, nil, args, rules)
          return reduct
    case "accu" :
          (reduct, accu) := td-or-bu(trfn, term, head(args), tail(args), rules)
          return accu
    case "accu, trafo" :
          return td-or-bu(trfn, term, head(args), tail(args), rules);
.
funct td-or-bu(trfn, term, accu, args, rules) ≡
  return switch traversal-strategy(trfn)
              case "top-down" : top-down(trfn, term, accu, args, rules)
              case "bottom-up" : bottom-up(trfn, term, accu, args, rules);
.
funct top-down(trfn, term, accu, args, rules) ≡
  reduct := reduce(compose(trfn, [term, accu, args]), rules)
  return if reduct = fail then visit-children(trfn, term, accu, args, rules)
         else make-reduct(trfn, term, reduct)
         fi
.
funct bottom-up(trfn, term, accu, args, rules) ≡
  (term, accu) := visit-children(trfn, term, accu, args, rules)
  reduct := reduce(compose(trfn, [term, accu, args]), rules)
  return if reduct = fail then (term, accu)
         else make-reduct(trfn, term, reduct)
         fi
.
funct visit-children(trfn, term, accu, args, rules) ≡
  (fn, children) := decompose(term)
  children' := nil
  foreach child in children do
    (reduct, accu) := td-or-bu(trfn, child, accu, args, rules)
    children' := append(children', reduct)
  od
  return (compose(fn, children'), accu)
.
funct make-reduct(trfn, term, reduct) ≡
  return switch traversal-kind(trfn)
              case "trafo" : (reduct, nil)
              case "accu" : (term, reduct)
              case "accu, trafo" : reduct;
.
```

4.1 Parsing traversal functions

The terms used in the rewrite rules of ASF+SDF have completely user-defined syntax. In order to parse a specification, the user-defined term syntax is combined with the standard equation syntax of ASF. This combined syntax is used to generate a parser that can parse the specification.

In order to parse the rewrite rules of a traversal function we need grammar rules that define them.

A first approach is to use Algorithm 4 (Appendix A) to generate the syntax for any possible application of a traversal function[3] This simple approach relieves the programmer from typing in the trivial productions himself. In practice, this solution has two drawbacks:

- The parsetables tend to grow by a factor equal to the number of traversal functions. As a result, scalability is lost since the turnaround time for parsetable generation will grow accordingly.
- Such generated grammars are possibly ambiguous. Disambiguating grammars is an involved process, for which the user needs complete control over the grammar. This control is lost if generated productions can interfere with user-defined productions.

An alternative approach is to let the user define the grammar for each construct that he wants to use in the rewrite rules of a traversal function.

For example, for the specification in Figure 6 this means that the following rule should be added to the syntax by hand: `inc(NAT) -> NAT {trafo}`. The other production, `inc(TREE) -> TREE {trafo}`, allows to parse the input term in Figure 7.

The amount of work for defining or changing a traversal function increases by this approach, but it is still proportional to the number of node types that are actually visited. Now the parsetable also grows proportionally to the number of visited node types and scalability is regained.

We have opted for a combination of both solutions. By default, the second approach is used and the user must specify all necessary declarations. However, if the traversal function is annotated with `generate-syntax`, syntax is generated for all reachable sorts. Now we can have the comfort of syntax generation, but we can use explicit specification when needed.

4.2 Interpretation of traversal functions

Remember that the syntax of terms in ASF+SDF is completely user-defined. The ASF interpreter handles user defined syntax in the following manner; it rewrites *parse trees* directly instead of abstract terms. The parse trees of rewrite rules are simply matched with the parse trees of terms during rewriting. A reduction is done by substituting the parse tree of the right-hand side of a rule at the location of a redex in the term.

[3]Note that this collection of generated functions can be viewed as one *overloaded* function.

The ASF+SDF interpreter follows the reduction strategy as presented in Algorithms 2 and 3.

4.3 Compilation of traversal functions

In order to have better performance of rewriting systems, compiling them to C has proven to be very beneficial. The ASF+SDF compiler [27] translates rewrite rules to C functions. After compilation, the runtime behavior of a rewriting system is:

1. The applicative parse tree of a term is translated to an abstract tree. During this process a dictionary is constructed. It contains a mapping from abstract trees to applicative parse trees and from abstract trees to C functions.

2. A specific C function is called for each node in this tree. This function contains a dedicated matching automaton for the left-hand sides of all rules that have this node as outermost node. It also contains an automaton for checking the conditions. Finally there are C function calls to other similarly compiled rewrite rules for evaluation of the right-hand sides.

3. Every failing application of a C function means that this node is in normal form. It results in the actual construction of the node in memory, the normal form. Note that for the nodes that have no rewrite rule, including the constructors, this happens always.

4. Finally, the resulting normal form in abstract tree format is translated back to parse tree format using the dictionary.

We are planning to add traversal functions to the compiler. They are easily fitted into the runtime scheme just described. A compiled traversal function will contain the matching automata for the left-hand sides of each defining rewrite rule. These can be computed as usual.

Depending on the traversal type and strategy, different generic traversal code can be added to the compiled function. For example, a `bottom-up` traversal can be created by adding this code before the matching code, and a `top-down` traversal by adding it just after.

This generic traversal code will be very similar to the code in the ASF interpreter. For that matter, the traversal itself is still rather interpreted than compiled. As an optimization, the compiler can analyze the user-defined syntax and the visiting rules and sometimes decide at compile time which branches can never contain a successful visit. In this case, the recursion is not generic anymore and we obtain a solution more specific for a rewrite system, which can be expected to be faster.

5 Experience

5.1 COBOL transformations

In a joint project of the Software Improvement Group (SIG), Centrum voor Wiskunde en Informatica (CWI) and Vrije Universiteit (VU) traversal functions have been applied to the conversion of COBOL programs [32]. This is based on earlier work described in [25]. The purpose was to migrate VS Cobol II to Cobol/390. An existing tool (CCCA from IBM) was used to carry out the basic, technically necessary, conversions. However, this leaves many constructions unchanged that will obtain the status "archaic" or "obsolete" in the next COBOL standard. In addition, compiler-specific COBOL extensions remain in the code and several outdated run-time utilities can be replaced by standard COBOL features.

Ten transformation rules were formalized to replace all these undesired language features and to achieve code improvements. Examples of rules are:

- Adding `END-IF` keywords to close `IF`-statements.
- Replace nested `IF`-statements to `EVALUATE`-statements.
- Replace outdated `CALL` utilities by standard COBOL statements.
- Reduce `GO-TO` statements: a goto-elimination algorithm that itself consists of over 20 different transformation rules that are applied iteratively.

After formalization of these ten rules in ASF+SDF with traversal functions, and applying them to a test base of 582 programs containing 440000 lines of code, the following results were obtained:

- 17000 `END-IF`s were added.
- 4000 lines were changed in order to eliminate `CALL`-utilities.
- 1000 `GO-TO`s have been eliminated (about 65% of all `GO-TO`s).

Each transformation rule is implemented by means of a traversal function defined by only a few equations. These results prove that this automatic transformation technology is at least superior to manual conversion.

5.2 SDF refactoring

In [24] a Framework for SDF Transformations (FST) is described that is intended to support *grammar recovery* (i.e., the process of recovering grammars from manuals and source code) as well as *grammar re-engineering* (transforming and improving grammars to serve new purposes such as information extraction from legacy systems and dialect conversions). The techniques are applied to a VS COBOL II grammar. The experience with traversal functions is positive. To cite the authors:

> "At the time of writing FST is described by 24 traversal functions with only a few rewrite rules per function. The SDF grammar itself has about 100 relevant productions. This is a remarkable indication for the usefulness of the support for traversal functions. In worst case, we would have to deal with about 2400 rewrite rules otherwise."

5.3 CASL Transformations

CASL (Common Algebraic Specification Language) [9] is a recently developed specification formalism. The design of this language has been performed in a number of steps. First the abstract syntax was defined together with the formal semantics. Later on a concrete syntax was designed.

Due to parsing problems and insights gained while developing sample CASL specifications, the concrete syntax has been modified several times.

The most recent modification in the concrete syntax dealt, among others, with the location where priority and associativity relations of operators had to be defined in the specification. In the latest version these definitions must be grouped at the beginning of a structural specification instead of being scattered over the entire specification. Using traversal functions a specification has been developed to move these relations to the appropriate place in the specification.

5.4 Miscellaneous projects

Various other experiments have been carried out ranging from an SDF checker, Java refactoring, and others.

6 Discussion

Traversal functions are based on a minimalistic design that tries to combine type safety with expressive power. In this section we discuss the consequences and the limitations of this approach.

6.1 Declarative *versus* Operational Specifications

Traversal functions are expressed by annotation of the function declaration. Understanding the meaning of the rules requires understanding which function is a traversal function and what visiting order it uses.

In pure algebraic specification, it is considered bad practice to depend on the rewriting strategy (i.e. the operational semantics) when writing specifications. By extending the operational semantics of our rewrite system with traversal functions, we effectively encourage using operational semantics.

However, if term rewriting is viewed as a programming paradigm, traversal functions enhance the declarative nature of specifications. That is, without traversal functions a simple transformation must be coded using a lot of "operational style" rewrite rules. With traversal functions, only the essential rules have to be defined. The effort for understanding and checking a specification decreases significantly.

In Appendix A we show how traversal functions in ASF+SDF can be translated to specifications without traversal functions in a relatively straightforward manner. So, traversal functions can be seen as an abbreviation mechanism.

6.2 Limited Visiting Orders

Recall from Figure 1 the main left-to-right visiting orders for trees: preorder (top-down), postorder and endorder (bottom-up) combined with two stop criteria: stop at first node or visit all nodes.

The visiting orders that can be expressed by traversal functions are preorder/first and endorder/all. In other words, we have a limited set of primitives that is not complete with respect to the generic visiting orders as discussed in Section 1.3.

However, most of the other visiting orders can be expressed in terms of our two primitives in a relatively simple manner. Focusing on transformers, we propose the following simulations:

Preorder/first: Is a `bottom-up` traversal.

Preorder/all: Start with a `top-down` traversal and use recursive calls in the right-hand side of each rule.

Postorder/first: This cannot be simulated in a simple manner. However, it is possible to use a `top-down` accumulating transformer to keep track whether we have already reduced a term. We leave the details to the reader.

Postorder/all: Start with a `top-down` traversal and use recursive calls in the conditions of each rule to simulate the postorder behavior.

Endorder/first: In other words, we only want to reduce the deepest match. Apply a `top-down` transformer and use a condition in each visiting rule to check if the traversal function is not applicable to any of the children.

Remember that conditions are checked before a rule is applied. If the traversal is applicable to any of the children, such condition fails. After that, the recursion tries to match deeper in the term. See `drepl` in Figure 16 for an example of this technique.

Endorder/all: Is a `top-down` traversal.

Using conditional term rewriting one can, eventually, express any traversal scheme. However, the fact remains that we only provide primitives for preorder/first and endorder/all and any other visiting order is the responsibility of the user, who must *encode* it.

To remedy this situation, we could supply all primitives from Figure 1. We have opted not to do this in order to simplify the understanding of sets of rules.

6.3 Limited Types of Traversal Functions

Finally, we discuss the limitation that accumulators can only map subtrees to a *single* sort and transformers can only do sort preserving transformations.

One can argue that general non-sort-preserving transformations cannot be expressed conveniently with this restriction. Such transformations typically occur when translating from one language to another and they will completely change the type of every subterm.

However, in the case of *full* translations the usefulness of *any* traversal scheme is debatable, since translation rules have to be given for any language construct anyway.

An interesting case are *partial* translations as occur when, for instance, embedded language statements are being translated while all surrounding language constructs remain untouched. In this case, the number of rules will be proportional to the number of *translated* constructs only and not to the total number of grammatical constructs. Most of such partial transformations can be seen as the combination of a sort-preserving transformation for the constructs where the transformation is not defined and a non-sort-preserving transformation for the defined parts. If the sort-preserving part is expressed as a transformer, we have again a number of rewrite rules proportional to the number of translated constructs.

To summarize, it is difficult to see how a generic non-sort-preserving traversal primitive could really make specifications of translations more concise.

6.4 Abstraction and reuse

There is one goal of specification though, were genericity is often required: software reuse. As was explained above, a traversal function is the combination of a visiting order and a visiting function. These two can be separated, as in Stratego, to allow the visiting function to be reused in different visiting orders.

We do not separate them because their *combination* allows correct typing. For example, one can specify a set of rules that map a certain tree to a scalar value, When applied in some specific order this will always yield well-formed terms, but when applied in a different order this can yield not well-formed terms. ASF ensures that every (intermediate) term is always well-formed. In other words, we have chosen for correctness above reusability.

6.5 Conclusions

We have described term rewriting with traversal functions as an extension to ASF+SDF. The advantages of our approach are:

- The most frequently used traversal orders are provided as built-in primitives.
- The approach is fully type-safe.
- Traversal functions can be implemented efficiently.

To summarize, traversal functions are a nice compromise between simplicity and expressive power.

The main disadvantage of our approach manifests itself when dealing with traversal orders that are not provided by the built-in primitives. Two escapes are possible: such traversals can either be simulated as a modification of one of the built-in strategies (by adding conditions or auxiliary functions), or one can fall back to the tedious specification of the traversal by enumerating traversal rules for all constructors of the grammar.

Experience with traversal functions shows that they are also applicable to real-life problems such as the transformation of COBOL and JAVA programs of significant sizes.

Acknowledgments

We received indispensable feedback from the users of traversal functions. Steven Klusener (Software improvement Group) and Hans Zaadnoordijk (University of Amsterdam) used them for COBOL transformations, and Ralf Lämmel (CWI and Vrije Universiteit Amsterdam) and Guido Wachsmuth (University of Rostock) applied them in SDF refactoring. Ralf Lämmel, Eelco Visser and Joost Visser commented on drafts of this paper.

References

[1] H. Alblas. Introduction to attribute grammars. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 1–15, Berlin Heidelberg New York, 1991. Springer Verlag.

[2] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[3] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[4] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[5] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[6] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications, WRLA 98*, 1998.

[7] 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*, 36:209–266, 2000.

[8] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by HTTP[4] and FTP[5], 1998.

[9] CoFI-LD. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0. Documents/CASL/Summary-v1.0, in [8], 1998.

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

[11] A. van Deursen and L. Moonen. Type inference for COBOL systems. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proc. 5th Working Conf. on Reverse Engineering*, pages 220–230. IEEE Computer Society, 1998.

[12] A. Felty. A logic programming approach to implementing higher-order term rewriting. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Extensions of Logic Programming (ELP '91)*, volume 596 of *Lecture Notes in Artifial Intelligence*, pages 135–158. Springer-Verlag, 1992.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusets, 1994.

[14] G. Hedin. *Incremental Semantic Analysis*. PhD thesis, Lund University, 1992.

[15] Gorel Hedin and Eva Magnusson. JastAdd - a Java-based system for implementing frontends. In Mark van den Brand and Didier Parigot, editors, *Proc. LDTA'01*, volume 44-2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[16] J. Heering. Implementing higher-order algebraic specifications. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 141–157. University of Pennsylvania, Philadelphia, 1992. Published as Technical Report MS-CIS-92-86.

[17] J. Heering. Second-order term rewriting specification of static semantics: An exercise. In A. van Deursen, J. Heering, and P. Klint, editors, *Language*

[4] `http://www.brics.dk/Projects/CoFI/`
[5] `ftp://ftp.brics.dk/Projects/CoFI/`

Prototyping, volume 5 of *AMAST Series in Computing*, pages 295–305. World Scientific, 1996.

[18] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[19] J.W. Klop. Term rewriting systems. In D. Gabbay, S.Abramski, and T. Maibaum, editors, *Handbook of Logic and Computer Science*, volume 1. Oxford University Press, 1992.

[20] D.E. Knuth. *The art of computer programming, volume 1.* Addison-Wesley, 1968.

[21] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Submitted for publication*, 2000.

[22] R. Lämmel. Typed generic traversals. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, 2001.

[23] Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report, Universiteit Utrecht.

[24] Ralf Lämmel and Guido Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In Mark van den Brand and Didier Parigot, editors, *Proc. LDTA'01*, volume 44-2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[25] A. Sellink, H. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In *Proceedings of Conference on Maintenance and Reengineering (CSMR'99)*, Amsterdam, March 1999.

[26] J.J. Thiel. Stop losing sleep over incomplete data type specifications. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 76–82, 1984.

[27] M.G.J. van den Brand, P. Klint, and P.A. Olivier. Compilation and memory management for ASF+SDF. In *Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.

[28] E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming*, pages 86–104, Ponte de Lima, July 2000.

[29] E. Visser. Stratego: A language for program transformation based on rewriting strategies. system description for stratego 0.5. *Submitted for publication*, 2000.

[30] Eelco 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)*, Lecture Notes in Computer Science. Springer-Verlag, May 2001. To appear.

[31] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *SIGPLAN Notices*, 24(7):131–145, 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[32] H. Zaadnoordijk. Source code transformations using the new ASF+SDF meta-environment. Master's thesis, University of Amsterdam, Programming Research Group, 2001.

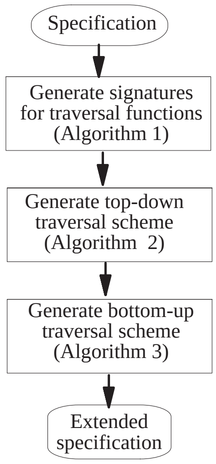

Figure 24: Organization of semantic definitions.

A Translational Semantics of Traversal Functions

We want to get a better understanding of the relation between rewriting systems with and without traversal functions. Therefore, we will now define a translation from rewriting systems with traversal functions to rewriting systems without traversal functions. The overall approach is shown in Figure 24:

1. Generate the necessary signature for the traversal functions. Starting with the signature definition of a user-defined traversal function, new instances of this definition are generated for all sorts that are reachable from the sort of the first argument of the traversal function. See Section A.1.

2. Generate the `top-down` traversal scheme. This is a set of default rewrite rules that encodes a traversal of the first argument of a traversal function such that the traversal function is applied to that first argument and all its subtrees. In the absence of user-defined rules, a traversal is performed without performing any operation on any subtree. In the presence of user-defined rules, a user-defined rule is applied to the current node when this is possible. See Section A.2.

3. For `bottom-up` traversal functions, extend the `top-down` traversal scheme in such a way that the traversal function is applied in an `bottom-up` fashion. We do this by applying a transformation to the user-defined rules. See Section A.3.

Although conceptually the `bottom-up` traversal scheme is closer to standard leftmost innermost rewriting, for this definition it is more convenient to start with defining the semantics of the `top-down` scheme.

We start with a rewriting system (S, Sig, R, D), defined by a collection of sorts S, a signature Sig, rules R and default rules D. The rules in D will only be applied when none of the rules in R are applicable. We allow rules to have a number of conditions which assign terms to new variables by matching.

We now define an extended rewriting system $(S, Sig, R, D, Sig_t, R_t, D_t)$ by adding a signature Sig_t, rules R_t and default rules D_t that contain definitions for traversal functions. The contents of Sig_t, R_t and D_t are defined below.

A.1 Generating a signature for traversal functions

We require that the ordinary signature and the signature of traversal functions do not overlap, i.e. $Sig \cap Sig_t = \emptyset$. Furthermore, the elements of Sig_t may only be one of the following three forms (where S_i represents a declared sort in the signature, and the # operator means creates tuples of its arguments):

Transformer: $f(S_1, \ldots, S_n) \to S_1$ $(n \geq 1)$

Accumulator: $f(S_1, S_2, \ldots, S_n) \to S_2$ $(n \geq 2)$

Accumulating transformer: $f(S_1, S_2, \ldots, S_n) \to S_1 \# S_2$ $(n \geq 2)$

For each traversal function in Sig_t we compute a new set of productions. We start by calculating the set of reachable sorts from the first argument of a traversal function (S_1). For each of these reachable sorts we create a new production. Algorithm 4 describes the process in detail.

Now that we have generated the new signature Sig'_t we define the rules in R_t to be over the combined signature $Sig \cup Sig'_t$. This allows the user to define rewrite rules for every reachable sort for a single traversal function.

A.2 The semantics of `top-down` traversals

Next, we generate rewrite rules that describe the `top-down` traversal over a tree. For simplicity's sake, we assume that D_t is empty for the moment. So there are no user-defined default rules. We will remove this restriction later on in Section A.5.

For a single traversal function the generation process, as summarized by Algorithm 5, is as follows:

- It handles all sorts that the traversal can reach, which have been computed by Algorithm 4. Each of these sorts is produced by a number of productions in the original signature (Sig). To be able to traverse down to the children of applications of such productions, we compute a specific rewrite rule for each of them. This new rewrite rule is different depending on the type of traversal (transformer, accumulator or accumulating transformer).
- All of these generated rewrite rules are added to the set of default rules D_t. By doing this the generated rules are automatically tried after all

Algorithm 4 Generating signatures for traversal functions

S_i, T_i, U are sort names and X, Y are either simple sort names or tuples (#) of sort names

```
begin
  Sig'_t := Sig_t;
  foreach f(S_1, ..., S_n) → X ∈ Sig_t do
    R := {S_1};
    Size := 0;
    while ||R|| ≠ Size do        (Get reachable sorts by fixed point calculation)
        Size := ||R||;
        foreach p(T_1, ..., T_m) → Y ∈ Sig do
          if Y ∈ R then R := R ∪ {T_1, ..., T_m} fi;
        od;
    od;
    foreach U ∈ R do
      if X = S_1 then                                          (transformer)
        Sig'_t = Sig'_t ∪ {f(U, S_2, ..., S_n) → U};
      elsif X = S_2 then                                       (accumulator)
          Sig'_t = Sig'_t ∪ {f(U, S_2, ..., S_n) → S_2};
      elsif X = S_1#S_2 then                      (accumulating transformer)
          Sig'_t = Sig'_t ∪ {f(U, S_2, ..., S_n) → U#S_2};
      fi
    od;
  od;
```

Algorithm 5 Generating traversal rules.

S_i, T_i are sort names and V_i, W_i, A_i are sorted variable names.

begin

$D_t := \emptyset$;

foreach $f(S_1, \ldots, S_n) \to X \in Sig'_t$ **do**

foreach $p(T_1, \ldots, T_m) \to S_1 \in Sig$ **do**

if $X = S_1$ **then** (transformer)

$D_t := D_t \cup \{\textit{transformer-rule}\}$;

elsif $X = S_2$ **then** (accumulator)

$D_t := D_t \cup \{\textit{accumulator-rule}\}$;

elsif $X = S_1 \# S_2$ **then** (accumulating transformer)

$D_t := D_t \cup \{\textit{combination-rule}\}$;

fi;

od;

od;

where

$$\textit{transformer-rule} \equiv \frac{\begin{array}{c} V'_1 = f(V_1, W_2, \ldots, W_n), \\ \ldots, \\ V'_m = f(V_m, W_2, \ldots, W_n) \end{array}}{f(p(V_1, \ldots, V_m), W_2, \ldots, W_n) \to p(V'_1, \ldots, V'_m)}$$

$$\textit{accumulator-rule} \equiv \frac{\begin{array}{c} A_2 = f(V_1, A_1, W_3, \ldots, W_n), \\ A_3 = f(V_2, A_2, W_3, \ldots, W_n), \\ \ldots, \\ A_{m+1} = f(V_m, A_m, W_3, \ldots, W_n) \end{array}}{f(p(V_1, \ldots, V_m), A_1, W_3, \ldots, W_n) \to A_{m+1}}$$

$$\textit{combination-rule} \equiv \frac{\begin{array}{c} (V'_1, A_2) = f(V_1, A_1, W_3, \ldots, W_n), \\ (V'_2, A_3) = f(V_2, A_2, W_3, \ldots, W_n), \\ \ldots, \\ (V'_m, A_{m+1}) = f(V_m, A_m, W_3, \ldots, W_n) \end{array}}{f(p(V_1, \ldots, V_m), A_1, W_3, \ldots, W_n) \to (p(V'_1, \ldots, V'_m), A_{m+1})}$$

end

user-defined rules have been tried. Therefor, the generated traversal stops as soon as a user-defined rule fires. Because the user-defined rules are always tried *before* the generated rules, we effectively obtain a topdown traversal.

As an aside, the generated rules for transformers and accumulators could have been written as simple unconditional rules. But this is not true for the accumulating transformer. For uniformity, we use conditional rules in all three cases. Note that D_t was initially empty in order to prevent overlapping of user-defined and generated rules.

A.3 The semantics of `bottom-up` traversals

Now we define the meaning of traversal functions with the `bottom-up` attribute. To do this, we first apply the exact same Algorithm 5, obtaining a `top-down` traversal scheme. After that we need two more algorithms to obtain a set of rewrite rules that specify the behavior of a `bottom-up` traversal. We will design two transformations on the *user-defined* rules in R_t to do this.

First Algorithm 6 divides the user-defined rules into two kinds. There are rules that have only a variable as the traversed argument. We call such rules *not-production-specific*. The second kind of rules do match a particular production in their first argument. These are the production-specific rules. Algorithm 6 replaces each not-production-specific rule by a collection of production-specific rules. Namely, for each production that produces the sort of the traversed argument a rule is added. Now we are able to address the children of all nodes properly.

Next we transform all rules in R_t by adding recursive calls to the conditions using Algorithm 7. Because the conditions are evaluated before a rule is applicable, we obtain a bottom-up traversal.

It is important that a transformed user-defined rule does not fire if the generated recursive calls have changed the original arguments, such that the original user-defined rule would not have fired. Therefor, we use *matching conditions* to ensure that the original patterns still occur after recursion. Suppose these matching conditions fail, then the rules fails. This allows a generated default rule to traverse down to the children to do the rest of the computation.

Note that implementing traversal functions by using this scheme will have inefficient run-time behavior, because these children are now visited more than once.

A.4 Constructing the conventional rewrite system

After application of the above algorithms, the semantics of the original extended rewriting system $(S, Sig, R, D, Sig_t, R_t, D_t)$ is defined by the conventional rewriting system $(S, Sig \cup Sig'_t, R \cup R'_t, D \cup D_t)$.

For example, if we apply these algorithms to the `inc` example from Figure 6, we get as result the expanded specification shown in Figure 25.

Algorithm 6 Removing not-production-specific rules.

T_i and S are sorts, Z is a variable, W_i are term patterns and V_i are fresh variables.

begin
  $R'_t := \emptyset$;
  foreach $r \in R_t$ **do**
    if $r = \dfrac{Conditions}{f(Z, W_2, \ldots, W_n) \to P}$ **then** (a not-production-specific rule)
      $S :=$ *sort-of-variable*(Z)
      foreach $g(T_1, \ldots, T_m) \to S \in Sig$ **do**

$$r' := \frac{\begin{array}{c} Z = g(V_1, \ldots, V_m), \\ Conditions \end{array}}{f(g(V_1, \ldots, V_m), W_2, \ldots, W_n) \to P}$$

        $R'_t := R'_t \cup \{r'\}$
      od
    else
        $R'_t := R'_t \cup \{r\}$
    fi
  od
  $R_t = R'_t$
end

Algorithm 7 Generating conditions to obtain `bottom-up` traversal.

S_i are sorts, P, P_i, W_i are term patterns and A_i, V_i are fresh variables.

begin
  $R'_t := \emptyset$
  foreach $\dfrac{\mathit{Conditions}}{f(g(P_1,\ldots,P_m),W_2,\ldots,W_n)\to P} \in R_t$ **do**

    if $f(S_1,\ldots,S_n)\to S_1 \in \mathit{Sig}_t$ **then** (transformer)
      $R'_t := R'_t \cup \{\mathit{transformer\text{-}rule}\}$
    elsif $f(S_1,S_2,\ldots,S_n)\to S_2 \in \mathit{Sig}_t$ **then** (accumulator)
      $R'_t := R'_t \cup \{\mathit{accumulator\text{-}rule}\}$
    elsif $f(S_1,S_2,\ldots,S_n)\to S_1\#S_2 \in \mathit{Sig}_t$ **then** (accumulating transformer)
      $R'_t := R'_t \cup \{\mathit{combination\text{-}rule}\}$
    fi
  od
where

$$\mathit{transformer\text{-}rule} \equiv \dfrac{\begin{array}{c}P_1 = f(V_1,W_2,\ldots,W_n),\\ \ldots\\ P_m = f(V_m,W_2,\ldots,W_n),\\ \mathit{Conditions}\end{array}}{f(g(V_1,\ldots,V_m),W_2,\ldots,W_n)\to P}$$

$$\mathit{accumulator\text{-}rule} \equiv \dfrac{\begin{array}{c}A_2 = f(P_1,A_1,W_3,\ldots,W_n),\\ \ldots\\ A_{m+1} = f(P_m,A_m,W_3,\ldots,W_n),\\ W_2 = A_{m+1},\\ \mathit{Conditions}\end{array}}{f(g(P_1,\ldots,P_m),A_1,W_3,\ldots,W_n)\to P}$$

$$\mathit{combination\text{-}rule} \equiv \dfrac{\begin{array}{c}(P_1,A_2) = f(V_1,A_1,W_3,\ldots,W_n),\\ \ldots\\ (P_m,A_{m+1}) = f(V_m,A_m,W_3,\ldots,W_n),\\ W_2 = A_{m+1},\\ \mathit{Conditions}\end{array}}{f(g(V_1,\ldots,V_m),A_1,W_3,\ldots,W_n)\to P}$$

end

```
module Tree-inc
imports Tree-syntax
exports
  context-free syntax
    inc(TREE) -> TREE
    inc(NAT)  -> TREE

equations

[default-1] T1' = inc(T1), T2' = inc(T2)
            ===========================
            inc(f(T1, T2)) = f(T1', T2')

[default-2] T1' = inc(T1), T2' = inc(T2)
            ===========================
            inc(g(T1, T2)) = g(T1', T2')

[default-3] T1' = inc(T1), T2' = inc(T2)
            ===========================
            inc(h(T1, T2)) = h(T1', T2')

[default-4] inc(N) = N

[5]         inc(N) = N + 1
```

Figure 25: Expanded specification for `inc`.

A.5 Removing the assumption $D_t = \emptyset$

So far, we have assumed that $D_t = \emptyset$. By doing this and putting every generated rule into the set of default rules we have separated the user-defined rules from the generated rules, preventing any possible overlap. This restriction can be solved by applying another algorithm first.

Rewriting systems with default rules can be transformed into rewriting systems without default rules [10]. Such a transformation, which is based on calculating complement sets of constructors [26], can be applied to the rules in $R_t \cup D_t$ before Algorithm 5 is executed in order to ensure that $D_t = \emptyset$. We do need to know what the constructors are for the user defined rules in $R_t \cup D_t$. Since $Sig_t \cap Sig = \emptyset$ we can assume all productions in Sig to be constructors for the user defined traversal rules.

```
module Pico-syntax
imports Pico-Identifiers Pico-Integers Pico-Strings Types
exports
  sorts PROGRAM
  context-free syntax
    "begin" DECLS STATS "end"                  -> PROGRAM
    "declare" ID-TYPES ";"                     -> DECLS
    ID ":" TYPE                                -> ID-TYPE
    {ID-TYPE ","}*                             -> ID-TYPES

    ID ":=" EXP                                -> STAT
    "if" EXP "then" STATS "else" STATS "fi"    -> STAT
    "while" EXP "do" STATS "od"                -> STAT
    {STAT ";"}*                                -> STATS

    ID                                         -> EXP
    NAT-CON                                    -> EXP
    STR-CON                                    -> EXP
    EXP "+" EXP                                -> EXP {left}
    EXP "-" EXP                                -> EXP {left}
    EXP "||" EXP                               -> EXP {left}
    "(" EXP ")"                                -> EXP {bracket}
  context-free priorities
    EXP "||" EXP -> EXP >
    EXP "-" EXP  -> EXP >
    EXP "+" EXP  -> EXP
```

Figure 26: SDF grammar for the small imperative language PICO.

B Further Examples of Traversal Functions

In this section we give some non-trivial examples of traversal functions. They all use the small imperative language PICO whose syntax is given in Figure 26.

B.1 Type Checking

The first example in Figure 27 defines a typechecker in a style described in [17]. The general idea is to use the declaration information to replace all variables and constants in the program by their type. This is done by the transformer `replace` specified in equations `[1]`, `[2]` and `[3]`.

Next, all type correct expressions are simplified (equations `[4]`, `[5]` and `[6]`).

Finally, type-correct statements are removed from the program (equations `[7]`, `[8]` and `[9]`).

As a result, a type correct program will reduce to the empty program and

```
module Pico-Typecheck
imports Pico-syntax
exports
context-free syntax
 type(TYPE)              -> ID
 replace(STATS, ID-TYPE) -> STATS   {traversal(trafo),
                                      generate-syntax}
equations
[0] begin declare Decl*1, Id-type, Decl*2; Stat* end =
      begin declare Decl*1, Decl*2; replace(Stat*, Id-type) end

[1] replace(Id     , Id : Type) = type(Type)
[2] replace(Nat-con, Id : Type) = type(natural)
[3] replace(Str-con, Id : Type) = type(string)

[4] type(string) || type(string)  = type(string)
[5] type(natural) + type(natural) = type(natural)
[6] type(natural) - type(natural) = type(natural)

[7] Stat*1; if type(natural) then Stat*2 else Stat*3 fi ; Stat*4
     = Stat*1; Stat*2; Stat*3; Stat*4

[8] Stat*1; while type(natural) do Stat*2 od; Stat*3
     = Stat*1; Stat*2; Stat*3

[9] Stat*1; type(Type) := type(Type); Stat*2
     = Stat*1; Stat*2
```

Figure 27: A typechecker for PICO which uses a transformer.

a type incorrect program will reduce to a simplified program that precisely contains the incorrect statements.

B.2 Type inference

The second example in Figure 28 shows the use of an accumulator. This specification computes an equivalence relation for PICO variables based on their use in a program, also known as type-inferencing [11]. The accumulator `type-inference` collects identifier declarations, expressions and assignments and puts them in separate equivalence sets. This is expressed by equations `[0]`,`[1]` and `[2]`.

In equations `[3]` through `[6]`, equivalence sets are simplified, while equation `[7]` computes the transitive closure of the equivalence relation. Note that these normal rules use associative matching to concisely express operations on sets.

```
module Pico-type-inference
imports Pico-syntax
exports
sorts SET SETS
context-free syntax
 "{" EXP* "}"  -> SET
 "[" SET* "]"  -> SETS

 type-inference(PROGRAM, SETS) -> SETS {traversal(accu),
                                        generate-syntax}

variables
 "Set"[0-9]* -> SET
 "Set*"[0-9]* -> SET*
 "Sets"[0-9]* -> SETS
 "Exp*"[0-9]* -> EXP*
equations
[0] type-inference(Id : Type, [ Set* ] ) = [ { Id }     Set* ]
[1] type-inference(Id := Exp, [ Set* ] ) = [ { Id Exp } Set* ]
[2] type-inference(Exp      , [ Set* ] ) = [ { Exp }    Set* ]

[3] { Exp*1 Exp Exp*2 Exp Exp*3 } = { Exp*1 Exp Exp*2 Exp*3 }
[4] { Exp*1 Exp1 + Exp2 Exp*2 }   = { Exp*1 Exp1 Exp2 Exp*2 }
[5] { Exp*1 Exp1 - Exp2 Exp*2 }   = { Exp*1 Exp1 Exp2 Exp*2 }
[6] { Exp*1 Exp1 || Exp2 Exp*3 }  = { Exp*1 Exp1 Exp2 Exp*3 }

[7] [ Set*1 { Exp*1 Exp Exp*2 } Set*2
            { Exp*3 Exp Exp*4 } Set*3 ] =
    [ Set*1 { Exp*1 Exp Exp*2 Exp*3 Exp*4 } Set*2 Set*3 ]
```

Figure 28: Type inferencing for PICO programs using an accumulator.