Term Rewriting with Traversal Functions

MARK G.J. VAN DEN BRAND and PAUL KLINT and JURGEN J. VINJU
Centrum voor Wiskunde en Informatica

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 and can either traverse all nodes in a tree or can stop the traversal at a certain depth as soon as a matching node is found. They can either define sort preserving transformations or mappings to a fixed sort. We give small and somewhat larger examples of traversal functions and describe their operational semantics and implementation. An assessment of various applications and a discussion conclude the paper.

Categories and Subject Descriptors: D1.1 [**Programming Techniques**]: Applicative (functional) programming; D3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*; *Recursion*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Automated Tree Traversal, Term Rewriting, Types

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 nodes of the tree once 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 over thousand grammar productions. 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

Authors' address: Centrum voor Wiskunde en Informatica (CWI), Software Engineering Department, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, Mark.van.den.Brand@cwi.nl, Paul.Klint@cwi.nl and Jurgen.Vinju@cwi.nl

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 [Brand et al. 1996; Brand et al. 1996; 1998; Klint 2003] we know that term rewriting is a convenient, scalable technology for expressing analysis, transformation, and renovation of individual programs and complete software systems. The main reasons for this are:

—Term rewriting provides implicit tree pattern matching that makes it easy to find patterns in program code.
—Programs can easily be manipulated and transformed via term rewriting.
—Term rewriting is rule-based, which makes it easy to combine sets of rules.
—Efficient implementations exist that can apply rewrite rules to millions of lines of code in a matter minutes.

In this paper we aim at further enhancing term rewriting for the analysis and transformation of software systems and address 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 preserve.

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 [Bergstra et al. 1989; Deursen et al. 1996] and give various examples. Some larger examples of traversal functions are presented in Section 3. The operational semantics of traversal functions is given in Section 4 and implementation issues are considered in Section 5. Section 6 describes the experience with traversal functions and Section 7 gives a discussion.

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

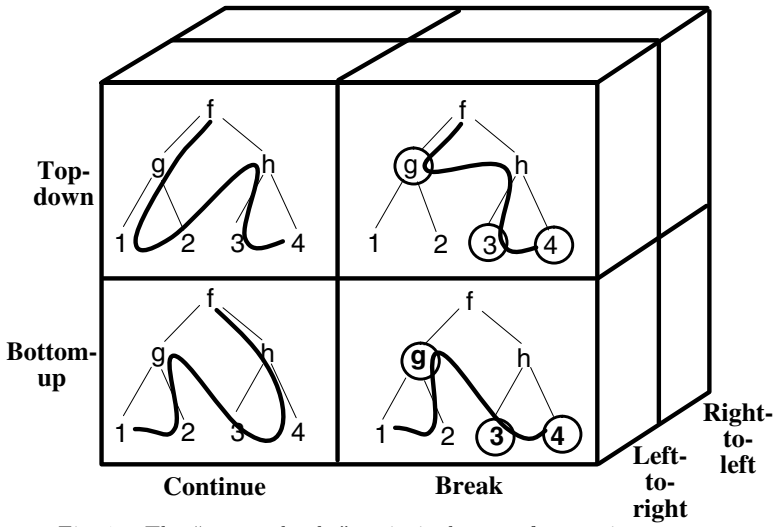

Fig. 1. The "traversal cube": principal ways of traversing a tree.

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

General properties of tree traversal are shown in the "traversal cube" in figure 1. On the first (vertical) axis, we distinguish the standard visiting orders *top-down* (order: root, sub-trees) and *bottom-up* (order: sub-trees, root). Note that for *binary* trees (as shown in the example) there is yet another way of visiting every node once called in-order[1] (order: one sub-tree, root, other sub-tree). In this paper we target arbitrary tree structures and therefore do not further consider this special case.

On the second (horizontal) axis, we distinguish traversals that *break* the recursion at specific nodes and traversals that always *continue* until all nodes have been visited. In the right half of figure 1, these breaks occur at the nodes `g`, `3`, and `4`.

On the third (depth) axis, we distinguish the direction of the traversal: visiting nodes from left-to-right or from right-to-left.

The eight possibilities given in the traversal cube are obvious candidates for abstraction and automation. In this paper we will focus on the front plane of the cube, i.e. left-to-right traversals since they are most prominently used in the application areas we are interested in. An extension to the complete cube is, however, straightforward.

During a tree traversal, a rewrite rule should be applied to some or all nodes to achieve the intended effect of the traversal. The type of the traversal 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

[1]In-order is called post-order in *The Art of Computer Programming, Volume 1* [Knuth 1968], nowadays post-order is used to indicate what is called end-order in that book.

type. This is the case when, for instance, C or Java are used and nodes in the tree are represented by a single "tree-node" data type. As with untyped nodes, homogeneous trees are manipulated easily because every combination 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 when, for instance, C or Java are used and a separate data type is introduced for representing each construct in a grammar (e.g., "declaration_node", "statement_node", "if_node" and so forth).

In this paper we will focus on the traversal of typed, heterogeneous, trees. Various aspects of traversal functions will be discussed:

—What is the type of their result value?

—What is the type of their other arguments?

—Does the result of the traversal 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.

1.4 A Brief Recapitulation of Term Rewriting

A basic insight in term rewriting is important for understanding traversal functions. Therefore we give a brief recapitulation of innermost term rewriting. For a full account see [Terese 2003].

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* (or *ground term*) is a term without variables. A *rewrite rule* is a pair of terms $T_1 \rightarrow T_2$. 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* where `X` is bound to `b`). If a variable occurs more than once in a term, a so-called non-left-linear pattern, the values matched by each occurrence are required to be equal. 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 ground term T and a set of rewrite rules, the purpose of a rewrite rule interpreter is to find a sub-term that can be reduced: the so-called *redex*. If sub-term R of T matches with the left-hand side of a rule $T_1 \rightarrow T_2$, 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. Note that innermost rewriting corresponds to call-by-value in a programming language like C.

Algorithm 1 An interpreter for innermost rewriting.

```
function match(term, term) : bindings or NO-MATCH
function substitute(term, bindings) : term

function innermost(t: term, rules : list_of[rule]) : term
begin
  var children, children' : list-of[term];
  var child, reduct, t'   : term;
  var fn                  : function-symbol;

  decompose term t as fn(children);
  children' := nil;
  foreach child in children
  do children' := append(children', innermost(child, rules)) od;
  t' := compose term fn(children');
  reduct := reduce(t', rules);
  return if reduct = fail then t' else reduct fi
end

function reduce(t : term, rules : list-of[rule]) :  term
begin
  var r : rule;
  var left, right : term;

  foreach r in rules
  do decompose rule r as left -> right;
     var b : bindings;
     b := match(t, left);
     if b != NO-MATCH then return innermost(substitute(right, b), rules) fi
  od
  return fail
end
```

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. We only show their signature. Terms can be *composed* from a top function symbol and a list of children, and they can be *decomposed* into their separate parts too. For example, if `fn` has as value the function-name `f`, and `children` has as value the list of terms `[a,b,c]`, then *compose term* `fn(children)` will yield the term `f(a,b,c)`. Decompose works in a similar fashion and also allows more structured term patterns. For example, *decompose term* `t` *into* `fn(child, children)` will result in the assignments `fn := f; child := a, children := [b, c]`. Rules are *composed* from a left-hand side and a right-hand side. They can also be *decomposed* to obtain these distinct parts. The underlying term representation can be either typed or untyped. The *compose* and *decompose* functionality as well as the functions `match` and `substitute` have to take this aspect into account. We use an `append` function to append 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

```
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
  variables
    N[0-9]*       -> NAT
    T[0-9]*       -> TREE
```

Fig. 2. SDF grammar for a simple tree language.

```
module Tree-trafo1
imports Tree-syntax
equations
[t1] f(T1, T2) = h(T1, T2)
```

Fig. 3. Example equation `[t1]`.

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 4 we will extend Algorithm 1 to cover traversal functions as well.

In the above presentation of term rewriting we have focused on the features that are essential for an understanding of traversal functions. Many other features such as, for instance, conditional rules with various forms of conditions (e.g., equality/inequality, matching conditions), list matching and the like are left undiscussed. In an actual implementation (Section 5) they have, of course, to be taken care of.

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. figure 2 displays an SDF grammar [Heering et al. 1989] for a simple tree language. The leafs are natural numbers and the nodes are constructed with one of the binary constructors `f`, `g` or `h`. Note that numbers (sort `NAT`) are embedded in trees (sort `TREE`) due to the production `NAT -> TREE`. This corresponds to a chain rule in a context-free grammar. The grammar also defines variables over natural numbers (`N`, `N0`, `N1`, ...) and trees (`T`, `T0`, `T1`, ...). Transformations on these trees can now be defined easily. For instance, if we want to replace all occurrences of `f` by `h`, then the single equation `[t1]` shown in figure 3 suffices. 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)
```

```
module Tree-trafo2
imports Tree-syntax
equations
[t2] f(g(T1, T2), T3) = h(T1, h(T2, T3))
```

Fig. 4. Example equation `[t2]`.

Similarly, if we want to replace all sub-trees of the form `f(g(T1, T2), T3)` by `h(T1, h(T2, T3))`, we can achieve this by the single rule `[t2]` shown in figure 4. 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:

—First, 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 `h(h(1,h(2,3)),4)` in two steps, depending on whether `[t1]` or `[t2]` is applied in the first reduction step. Observe, however, that an interpreter like the one shown in Algorithm 1 will always select one rule and produce a single result.

—The second problem is that rewrite rules cannot access any context information other than the term that matches the left-hand side of the rewrite rule. Especially for program transformation this is very restrictive.

—Thirdly, 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. Sub-terms can only be replaced by sub-terms 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`).

A common solution to the above three problems is to introduce new function symbols that eliminate the interference between rules. In our example, if we introduce the functions `trafo1` and `trafo2`, we can explicitly control the outcome of the combined transformation by the order in which we apply `trafo1` and `trafo2` to the initial term. By introducing extra function symbols, we also gain the ability to pass data around using extra parameters of these functions. Finally, the function symbols allow to express non-type-preserving transformations by explicitly typing the function to accept one type and yield another. This proposed change in specification style does not yield a semantically equivalent rewriting system in general. It is used as a practical style for specifications, for the above three reasons.

So by introducing new function symbols, three limitations of rewrite rules are solved. The main down side of this approach is that we loose the built-in facility of innermost rewriting to traverse the input term without an explicit effort of the

```
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))
```

Fig. 5. Definition of `trafo1` and `trafo2`.

programmer. Extra rewrite rules are needed to define the traversal of `trafo1` and `trafo2` over the input term, as shown in figure 5. Observe that equations `[1]` 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. Defining the traversal rules requires explicit knowledge of *all* productions 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.

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 typical tree traversal will then be described by m (m usually less than 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 traversed to get to the point of interest, resulting in tens to hundreds of rules that have to be written for a given large grammar and some specific transformation or analysis.

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

Higher-order term rewriting. One solution is the use of higher-order term rewriting [Huet and Lang 1978; Felty 1992; Heering 1992]. 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 [Heering 1996] for a simple example of higher-order term rewriting.

Higher-order term rewriting is a very powerful mechanism, which can be used to

avoid expressing entire tree traversals. It introduces, however, complex semantics and implementation issues. It does not solve the non-confluence problems discussed earlier (see Section 1.5). Another observation is that the traversal is done during matching, so for every match the sub-terms might be traversed. This might be very expensive.

Generic traversal or strategy primitives One can extend the rewriting language with a set of generic traversal or strategy primitives as basic operators that enable the formulation of arbitrary tree traversals. Such primitives could for instance be the traversal of one, some or all sub-trees of a node, or the sequential composition, choice or repetition of traversals. They can be used to selectively apply a rewrite rule at a location in the term. Generic traversal primitives separate the application of the rewrite rule from the traversal of the tree as advocated in *strategic programming*. See, for instance, [Visser 2001b] for a survey of strategic programming in the area of program transformations.

The genericity provided by generic traversals is hard to handle by conventional typing systems [Visser 2000; Lämmel 2003]. The reason for this is that the type of a traversal primitive is completely independent of the structures that it can traverse. In [Lämmel 2003] a proposal is made for a typing system for generic traversal primitives which we will further discuss in Section 1.7.

Having types is relevant for static type checking, program documentation, and program comprehension. It is also beneficial for efficient implementation and optimization. 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. Sub-terms can only be replaced by sub-terms of the same type, thus enforcing that the complete term remains well-typed. Type-checking a first-order many-sorted term rewriting system simply boils down to checking if both sides of every rewrite rule yield the same type and checking if both sides are well-formed with respect to the signature.

Traversal functions.

Our approach is to allow functions to traverse a tree automatically, according to a set of built-in traversal primitives. In our terminology, such functions are called *traversal functions*. They solve the problem of the extra rules needed for term traversal without loosing the practical abilities of functions to carry data around and having non-sort-preserving transformations.

By extending ordinary term rewriting with traversal functions, the type-system can remain the same. One can provide primitives that allow type-preserving and even a class of non-type-preserving traversals in a type-safe manner without even changing the type-checker of the language.

1.7 Related Work

1.7.1 *Directly Related Work.* We classify directly related approaches in tigure I and discuss them below.

ELAN [Borovanský et al. 1998] 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

Table I. Classification of traversal approaches.

	Untyped	**Typed**
Strategy primitives	Stratego [Visser 2001a]	ELAN [Borovanský et al. 1998]
Built-in strategies	Renovation Factories [Brand et al. 2000]	Traversal Functions, TXL [Cordy et al. 1991]

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 [Visser 2001a] is an untyped term rewriting language that provides user-defined strategies. Among its strategy primitives are rewrite rules and several generic strategy operators (such as, e.g., sequential composition, choice, and repetition) that allow the definition of any tree traversal, such as top-down and bottom-up, in an abstract manner. Therefore, tree traversals are first class objects that can be reused separately from rewrite rules. Stratego provides a library with all kinds of named traversal strategies such as, for instance, `bottomup(s)`, `topdown(s)` and `innermost(s)`.

Transformation Factories [Brand et al. 2000] are an approach in which ASF+SDF rewrite rules are generated from language definitions. After the generation phase, the user instantiates an actual transformation by providing the name of the transformation and by updating default traversal behavior. Note that the generated rewrite rules are well-typed, but very general types have to be used to obtain reusability of the generated rewrite rules.

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 combination function and a default value. The generated traversal function reduces each node to the default value, unless the user overrides it. The combination function combines the results in an innermost manner. The simulation of higher-order behavior again leads to very general types.

TXL [Cordy et al. 1991] TXL is a typed language for transformational programming [Cordy et al. 1991]. Like ASF+SDF it permits the definition of arbitrary grammars as well as rewrite rules to transform parsed programs. Although TXL is based on a form of term rewriting, its terminology and notation deviate from standard term rewriting parlance. TXL has been used in many renovation projects.

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

Stratego extends term rewriting with traversal strategy combinators and user-defined strategies. We are more conservative and extend first-order term rewriting only with a fixed set of traversal primitives. One contribution of traversal functions is that they provide a simple *type-safe* approach for tree traversals in first-order specifications. The result is simple, can be statically type-checked in a trivial manner and can be implemented efficiently. On the down-side, our approach does not allow adding new traversal orders: they have to be simulated with the given,

built-in, traversal orders. See [Klint 2001] for a further discussion of the relative merits of these two approaches.

Recently, in [Lämmel 2003] another type system for tree traversals was proposed. It is based on traversal combinators as found in Stratego. While this typing system is attractive in many ways, it is more complicated than our approach. Two generic types are added to a first-order type system: type-preserving (`TP`) and type-unifying (`TU(`τ`)`) strategies. To mediate between these generic types and normal types an extra combinator is offered that combines both a type-guard and a type lifting operator. Extending the type system is not needed in our traversal function approach, because the tree traversal is joined with the functional effect in a single traversal function. This allows the interpreter or compiler to deal with type-safe traversal without user intervention. As is the case with traversal functions, in [Lämmel 2003] traversal types are divided into type-preserving effects and mappings to a single type. The tupled combination is not offered.

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 interpreter and compiler of ASF+SDF. As a result, execution is more efficient and specifications are more readable, since users are not confronted with generated rewrite rules.

Although developed completely independently, our approach has much in common with TXL, which also provides type-safe term traversal. TXL rules always apply a pre-order search over a term looking for a given pattern. For matching subterms a replacement is performed. TXL rules are thus comparable with top-down transformers. A difference is that transformers perform only one pass over the term and do not visit already transformed subtrees. TXL rules, however, also visit the transformed subtrees. In some cases, e.g., renaming all variables in a program, special measures are needed to avoid undesired, repeated, transformations. In TXL jargon, traversal functions are all *one-pass* rules. Although TXL does not support accumulators, it has a notion of global variables that can be used to collect information during a traversal. A useful TXL feature that we do not support is the ability to skip subterms of certain types during the traversal.

1.7.3 *Other Related Work.* Apart from the directly related work already mentioned, we briefly mention 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 as heterogeneous trees have to be traversed. One solution to this problem are fold algebras as described in [Lämmel et al. 2000]: 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. Another way of introducing generic traversals in a functional setting is described in [Lämmel and Visser 2002].

Object-oriented languages. The traversal of arbitrary data structures is captured by the *visitor design pattern* described in [Gamma et al. 1994]. 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. An implementation of the visitor pattern is JJForester [Kuipers and Visser 2001]: 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. In [Visser 2001c] this approach is further generalized to traversal combinators.

Attribute grammars. The approaches described so far provide an operational view on tree traversals. Attribute grammars [Alblas 1991] 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 [Vogt et al. 1989] 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 [Hedin and Magnusson 2001] is recent work in the field of combining reference attribute grammars [Hedin 1992] 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 problems posed by program transformations.

2. TRAVERSAL FUNCTIONS IN ASF+SDF

We want to automate tree traversal in many-sorted, first-order term rewriting. We present traversal functions in the context of the language ASF+SDF [Bergstra et al. 1989; Deursen et al. 1996], but our approach can be applied to any term rewriting language. No prior knowledge of ASF+SDF is required and we will explain the language when the need arises.

The reason for choosing ASF+SDF is that it is a well-known language in the term rewriting community and that it is supported by an interactive development environment that is widely available[2]. In addition, there are many industrial applications that can be used as test cases for language extensions.

ASF+SDF uses context-free syntax for defining the signature of terms. As a result, terms can be written in arbitrary user-defined notation. This means that functions can have free notation (e.g., `move ... to ...` rather than `move(..., ...)`) and that the complete text of programs can be represented as well. The

[2] `www.cwi.nl/projects/MetaEnv`

context-free syntax is defined in SDF[3]. Terms are used in rewrite rules defined in ASF[4]. For the purpose of this paper, the following features of ASF are relevant:

—Many-sorted (typed) terms.

—Unconditional and conditional rules. Conditions are comparisons between two terms which come in three flavors: equality between terms, inequality between terms, and so-called assignment conditions that introduce new variables. In the first two flavors, no new variables may be introduced on either side. In the last form only one side of the condition may contain new variables, which are bound while matching the pattern with the other side of the condition.

—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. The programmer defines functions as usual by providing a signature and defining rewrite rules. The signature of a traversal function has to be defined as well. This is an ordinary declaration but it is explicitly labeled with the attribute `traversal`. We call such a labeled function a traversal function since from the user's perspective it automatically traverses a term: the rewrite rules for term traversal do not have to be specified anymore since they are provided automatically by the `traversal` attribute. The specification writer only has to give rewrite rules for the nodes that the traversal function will actually visit.

The rewrite rules provided by the `traversal` attribute thus define the *traversal* behavior while rewrite rules provided by the user define the *visit* behavior for nodes. 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 actual rewriting of sub-terms. Expanding a traversal function to this set of rewrite rules is a possible way of defining the *semantics* of traversal functions, which we do not further pursue here (but see [Brand et al. 2001]).

We continue our discussion in Section 1.6 on how to type generic traversals. The question is what built-in 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 discuss in order. In Section 2.3 we present examples of traversal functions. The merits and limitations of this approach are discussed in Section 7.

2.1 Kinds of Traversal Functions

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

Transformer A sort-preserving transformation, declared as:
$f(S_1, \ldots, S_n) \rightarrow S_1$ {`traversal(trafo)`}

[3]Syntax Definition Formalism.
[4]Algebraic Specification Formalism.

Accumulator A mapping to a single type, declared as:
$f(S_1, S_2, \ldots, S_n) \rightarrow S_2$ {`traversal(accu)`}

Accumulating Transformer A sort preserving transformation that accumulates information simultaneously, declared as:
$f(S_1, S_2, \ldots, S_n) \rightarrow S_1 \# S_2$ {`traversal(accu, trafo)`}

A Transformer will traverse its first argument. Possible extra arguments may contain additional data that can be used (but not modified) during the traversal. 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.

An Accumulator will traverse its first argument, while the second argument keeps the accumulated value. 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, it only changes 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.

An Accumulating Transformer is 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 the accumulated value.

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 (recall figure 1). Visiting strategies determine the order of traversal. We provide the following two strategies for each type of traversal:

Bottom-up First recur down to the children, then try to visit the current node. The annotation `bottom-up` selects this behavior.

Top-down First try to visit the current node and then traverse to the children. The annotation `top-down` selects this behavior.

Without an extra attribute, these strategies define traversals that visit all nodes in a tree. We add two attributes that select what should happen after a successful

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

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

in	trafo1(trafo2(f(f(g(1,2),3),4)))	out	h(h(1,h(2,3)),4)

Fig. 6. `trafo1` and `trafo2` from figure 5 now using `top-down` traversal functions.

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

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

Fig. 7. Transformer `inc` increments each number in a tree.

visit.

Break Stop visiting nodes on the current branch after a successful visit. The corresponding annotation is `break`.

Continue Continue the traversal after a successful visit. The corresponding annotation is `continue`.

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.

To be able to `break` a traversal is a powerful feature. For example, it allows the user to continue the traversal under certain conditions.

2.3 Examples of Transformers

In the following subsections, we give some trivial examples of transformers, accumulators, and accumulating transformers. All examples use the tree language introduced earlier in figure 2. In Section 3 we show some more elaborate examples.

2.3.1 *The trafo example from the introduction revised.* Recall the definition of the transformations `trafo1` and `trafo2` in the introduction (figure 5). They looked clumsy and cluttered the intention of the transformation completely. Figure 6 shows how to express the same transformations using two traversal functions.

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

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

Fig. 8. Transformer `incp` increments each number in a tree with a given value.

```
module Tree-frepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)  -> TREE
   frepl(TREE)    -> TREE {traversal(trafo,bottom-up,continue)}
equations
[1] frepl(g(T1, T2)) = i(T1, T2)
```

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

Fig. 9. Transformer `frepl` replaces all occurrences of `g` by `i`.

Observe how these two rules resemble the original rewrite rules. There is, however, one significant difference: these rules can only be used when the corresponding function is actually applied to a term.

2.3.2 *Increment the numbers in a tree.* The specification in figure 7 shows the transformer `inc`. Its purpose is to increment all numbers that occur in a tree. To better understand this example, we follow the traversal and rewrite steps when applying `inc` to the tree `f(g(1,2),3)`:

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

We start by the application of inc to the outermost node, then each node is visited in a left-to-right bottom-up fashion. If no rewrite rule is activated, the identity transformation is applied. So, in this example only naturals are transformed and the other nodes are left unchanged.

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. Observe that the *first* argument of `incp` is traversed and that the second argument is a value that is carried along during the

```
module Tree-frepl2
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)  -> TREE
   frepl2(TREE)    -> TREE {traversal(trafo, top-down, continue)}
equations
[1] frepl2(g(T1, T2)) = i(T1, T2)
```

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

Fig. 10. Transformer `frepl2` replaces all occurrences of `g` by `i`.

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

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

Fig. 11. Transformer `srepl` replaces shallow occurrences of `g` by `i`.

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

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

Fig. 12. Transformer `drepl` replaces deep occurrences of `g` by `i`.

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

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

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

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

Fig. 13. Accumulator `sum` computes the sum of all numbers in a tree.

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 the tree, and deep replacement that only replaces occurrences that are closest to the leafs of the tree.

Full replacement is defined in figure 9. We specified a `bottom-up` traversal that continues traversing after a reduction. This will ensure that all nodes in the tree will be visited. Note that in this case we could also have used a top-down strategy and get the same result as is shown in figure 10.

Shallow replacement is defined in figure 11. In this case, traversal stops at each outermost occurrence of `g` because `break` was given as an attribute. In this case, the top-down strategy is essential. Observe that a top-down traversal with the `break` attribute applies the traversal function at an applicable outermost node and does not visit the sub-trees 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 traverse certain sub-trees recursively while avoiding others explicitly.

We use the combination of a bottom-up strategy with the `break` attribute to define deep replacement as shown in figure 12. As soon as the rewrite rule applies to a certain node, the traversal visits no more nodes on the path from the reduced node to the root. In this case, the bottom-up strategy is essential.

2.4 Examples of Accumulators

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

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 13 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).

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 14 does the job.

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

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

Fig. 14. Accumulator `cnt` counts the nodes in a tree.

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

in	out
`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>`

Fig. 15. Accumulating transformer `pos` multiplies numbers by their tree position.

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 top-down traversal of the tree and to multiply each number by its position. This is achieved by the accumulating transformer `pos` shown in figure 15. 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.

3. LARGER EXAMPLES

Now we give some less trivial applications of traversal functions. They all use the small imperative language PICO whose syntax is shown in figure 16. The toy language PICO was originally introduced in [Bergstra et al. 1989] and has been used as running example since then. A PICO program consists of declarations followed by statements. Variables should be declared before use and can have two types: natural number and string. There are three kinds of statements: assignment, if-statement and while-statement. In an assignment, the types of the left-hand side and the right-hand side should be equal. In if-statement and while-statement the condition should be of type natural. The arguments of the numeric operators `+` and `-` are natural. Both arguments of the string-valued operator `||` are strings.

```
module Pico-syntax
imports Pico-whitespace
exports
  sorts PROGRAM DECLS ID-TYPE ID DECLS STAT STATS EXP
  sorts NAT-CON STR-CON
  lexical syntax
    [a-z] [a-z0-9]*                          -> ID
    [0-9]+                                   -> NAT-CON
    "\"" ~[\"\n]* "\""                       -> STR-CON
  context-free syntax
    "begin" DECLS STATS "end"                -> PROGRAM
    "declare" ID-TYPES ";"                   -> DECLS
    ID ":" TYPE                              -> ID-TYPE
    "natural" | "string"                     -> 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
```

Fig. 16. SDF grammar for the small imperative language PICO.

3.1 Type-checking

The example in figure 17 defines a type-checker for PICO in a style described in [Heering 1996]. The general idea is to reduce type-correct programs to the empty program and to reduce programs containing type errors to a program that only contains the erroneous statements. This is achieved by using the information from declarations of variables to replace all variable occurrences by their declared type and by replacing all constants by their implicit type. After that, all type-correct statements are removed from the program. As a result, only type-correct programs are normalized to the empty program.

This approach is interesting from the perspective of error reporting when rewriting is augmented with *origin tracking*, a technique that links back sub-terms of the normal form to sub-terms of the initial term [Deursen et al. 1993]. In this way, the residuals of the type-incorrect statements in the normal form can be traced back to their source. See [Tip and Dinesh 2001] for applications of this and similar techniques.

The example in figure 17 works as follows. First, it is necessary to accommodate

```
module Pico-typecheck
imports Pico-syntax
exports
context-free syntax
 type(TYPE)              -> ID
 replace(STATS, ID-TYPE) -> STATS {traversal(trafo,bottom-up,break)}
 replace(EXP  , ID-TYPE) -> STATS {traversal(trafo,bottom-up,break)}
equations
[0] begin declare Id-type, Decl*; Stat* end =
      begin declare Decl*; 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
```

in	out

`begin declare x : natural,`
`  s : string;`
`x := 10; s := "abc";`
`if x then x := x + 1`
`  else s := x + 2`
`fi;`
`y := x + 2;`
`end`
	`begin`
	`  declare;`
	`  type(string) :=`
	`  type(natural);`
	`end`

Fig. 17. A type-checker for PICO.

the replacement of variables by their type, in other words, we want to replace `x := y` by `type(natural) := type(natural)`, assuming that `x` and `y` have been declared as `natural`. This is achieved by extending the syntax of PICO with the context-free syntax rule

```
type(TYPE) -> ID
```

The actual replacement of variables by their declared type is done by the transformer `replace`. It has to be declared for all sorts for which equations for `replace` are defined, in this case `STATS` and `EXP`. It is a bottom-up, breaking, transformer. The second argument of `replace` is an (identifier, type) pair as it appears in a variable declaration.

Note that for more complex languages a bottom-up breaking transformer might not be sufficient. For example, when dealing with *nested scopes* it is imperative that the type-environment can be updated before going into a new scope. A top-down breaking transformer is used in such a case which stops at the entrance of a new

scope and explicitly recurs into the scope after updating the type-environment.

In equation `[0]` a program containing a non-empty declaration section is replaced by a new program with one declaration less. In the statements all occurrences of the variable that was declared in the removed declaration are replaced by its declared type. `replace` is specified in equations `[1]`, `[2]` and `[3]`. It simply replaces identifiers, natural constants and string constants by their type.

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 a type incorrect program will reduce to a simplified program that precisely contains the incorrect statements. The example that is also given in figure 17 shows how the incorrect statement `s := x + 2` (both sides of an assignment should have the same type) is reduced to `type(string) := type(natural)`.

The traversal order could be both top-down and bottom-up, since `replace` only matches leafs in `[1]`, `[2]` and `[3]`. However, `bottom-up` and `break` make this traversal more efficient because once a leaf has been visited none of its ancestors is visited anymore. This example shows that traversal functions can be used for this style of type-checking and that they make this approach feasible for much larger languages.

Equations `[7]` through `[9]` use associative matching (called list matching in ASF+SDF) to concisely express operations on lists of statements. For instance, in `[8]`, the list variables `Stat*1` and `Stat*3` represent the statements surrounding a while statement and `Stat*2` represents the list of statements in the body of the while statement. On the right-hand side of the equation these three lists of statements are concatenated thus effectively merging the body of the while statement with its surroundings.

3.2 Inferring Variable Usage

The second example in figure 18 computes an equivalence relation for PICO variables based on their usage in a program. This technique is known as *type-inference* [Cardelli 1997] and can be used for compiler optimization and reverse engineering. Examples are statically inferring variable types in a dynamically typed language such as Smalltalk or in a weakly typed language such as COBOL ([Deursen and Moonen 1998]).

The analysis starts with the assumption that the input program is correct. Based on their usage in the program variables are related to each other by putting them in the same equivalence class. Finally, the equivalence classes are completed by taking their transitive closure. Variables of the same type that are used for different purposes will thus appear in different classes. In this way one can, for example, distinguish integer variables used for dates and integer variables used for account numbers.

In the specification, notation is introduced for sets of expressions (`SET`) and sets of such sets (`SETS`). The accumulator `infer-type` is then declared that collects identifier declarations, expressions and assignments and puts them in separate equivalence classes represented by `SETS`. This is expressed by equations `[0]` and `[1]`. In `[0]` an assignment statement generates a new set consisting of both sides of the assignment. In `[1]` an expression generates a new set on its own. In equations `[2]`

```
module Pico-usage-inference
imports Pico-syntax
exports
sorts SET SETS
context-free syntax
 "{" EXP* "}"            -> SET
 "[" SET* "]"            -> SETS
 infer-use(PROGRAM,SETS) -> SETS {traversal(accu,top-down,break)}
 infer-use(STAT   ,SETS) -> SETS {traversal(accu,top-down,break)}
 infer-use(EXP    ,SETS) -> SETS {traversal(accu,top-down,break)}
variables
 "Set"[0-9]*  -> SET
 "Set*"[0-9]* -> SET*
 "Exp*"[0-9]* -> EXP*
equations
[0] infer-use(Id := Exp, [ Set* ] ) = [ { Id Exp } Set* ]
[1] infer-use(Exp      , [ Set* ] ) = [ { Exp }    Set* ]

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

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

in	**out**

`infer-use(`
`  begin declare x : natural,`
`  y : natural,`
`  z : natural;`
`  x := 0;`
`  if x then y := 1`
`  else y := 2 fi;`
`  z := x + 3; y := 4`
`  end, [])`
	`[ { y 4 2 1 } { z x 3 0 } ]`

Fig. 18. Inferring variable usage for PICO programs.

through `[5]`, equivalence sets are simplified by breaking down complex expressions into their constituting operands. Finally, equation `[6]` computes the transitive closure of the equivalence relation.

Note that equations `[2]` through `[6]` use list matching to concisely express operations on sets. For instance, in `[2]` the list variables `Exp*1`, `Exp*2` and `Exp*3`, are used to match elements that surround the two occurrences of the same expression `Exp`. On the right-hand side of the equation, they are used to construct a new list of expressions that contains only a single occurrence of `Exp`. In fact, this equation defines that `SET` actually defines sets! figure 18 also shows an example of applying `infer-use` to a small program.

3.3 Examples of Accumulating Transformers

We leave examples of accumulating transformers to the reader. They can be found in two directions. Either transformation with side-effects or a transformations with

state. A trivial example of the first is to generate a logfile of a transformation. Log entries are added to the accumulated argument while the traversed argument is transformed. This functionality can sometimes be split into first generating the logfile and then doing the transformation, but that inevitably leads to code duplication and degregation of performance.

An instance of the second scenario is a transformer that assigns a unique identification to some language constructs. The accumulated argument is used to keep track of the identifications that were already used. It is impossible to split this behaviour into a separate transformer and accumulator.

Algorithm 2 An interpreter for transformers, Part 1.

```
function traverse-trafo(t : term, rules : list-of[rule]) : term
begin
  var trfn    : function-symbol;
  var subject : term;
  var args    : list-of[term];

  decompose term t as trfn(subject,args)
  return visit(trfn, subject, args, rules);
end

function visit(trfn : function-symbol, subject : term, args : list-of[term],
               rules : list-of[rule]) : term
begin
  var subject', reduct : term;

  if traversal-strategy(trfn) = TOP-DOWN
  then subject' := reduce(typed-compose(trfn, subject, args), rules);
     if subject' = fail
     then return visit-children(trfn, subject, args, rules)
     else if traversal-continuation(trfn) = BREAK
        then return subject'
        else reduct := visit-children(trfn, subject', args, rules)
            return if reduct = fail then subject' else reduct fi
        fi
     fi
  else /* BOTTOM-UP */
     subject' := visit-children(trfn, subject, args, rules);
     if subject' = fail
     then reduct = reduce(typed-compose(trfn, subject, args), rules)
     else if traversal-continuation(trfn) = BREAK
        then return subject'
        else reduct = reduce(typed-compose(trfn, subject',args), rules)
            return if reduct = fail then subject' else reduct fi
        fi
     fi
  fi
end
```

Algorithm 3 An interpreter for transformers, Part 2.

```
function visit-children(trfn : function-symbol, subject : term,
                        args : list-of[term], rules : list-of[rule]) : term
begin
  var children, children' : list-of[term];
  var child, reduct       : term;
  var fn                  : id;
  var success             : bool;
  decompose term subject as fn(children);
  success := false;
  foreach child in children
  do reduct := visit(trfn, child, args, rules);
     if reduct != fail
     then children' := append(children', reduct);
          success   := true;
     else children' := append(children', child)
     fi
  od;
  return if success = true then compose the term fn(children') else fail fi
end
function typed-compose(trfn : function-symbol, subject : term,
                       args : list-of[term]) : term
begin
  var τ1, τ2, ..., τn, τsubject : type;
  var rsym : function-symbol;
  var fn   : id;
  τsubject := result-type-of(subject);
  decompose function-symbol trfn as fn:τ1 × τ2 × ... × τn -> τ1;
  rsym := compose function-symbol fn: τsubject × τ2 × ... × τn -> τsubject
  return compose term rsym(subject, args);
end
```

4. OPERATIONAL SEMANTICS

Now we will describe an operational semantics for traversal functions. We assume that we have a *fully typed* term representation.

This means that with every function name a first order type can be associated. For example, a function with name f could have type $f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau_r$. If $n = 0$, f is a constant of type $f :\rightarrow \tau_r$. If $n > 0$, f is either a constructor or a function with its arguments typed by $\tau_1, ..., \tau_n$ respectively. We will call this fully typed version of a function name a *function symbol* and assume that terms only contain function symbols. Of course, the term construction and destruction and matching functionality should be adapted to this term representation.

Note that the typed-term representation is an operational detail of traversal functions. It is needed to match the correct nodes while traversing a tree. However, a definition of a traversal function can be statically type-checked (Section 2) to ensure that its execution never leads to an ill-formed term.

4.1 Extending Innermost

We start with normal innermost rewriting as depicted earlier in Algorithm 1 (see Section 1.4). The original algorithm first normalizes the children of a term and relies on `reduce` to reduce the term at the outermost level.

In the modified algorithm, the call to the function `reduce` is replaced by a case distinction depending on the kind of function: a normal function (i.e., not a traversal function), a transformer, an accumulator, or an accumulating transformer. For these cases calls are made to the respective functions `reduce`, `traverse-trafo`, `traverse-accu`, or `traverse-accu-trafo`. Note that we describe the three kinds of traversal functions here by means of three different functions. This is only done for expository purposes (also see the discussion in Section 4.5).

4.2 Transformer

The function `traverse-trafo` and its auxiliary functions are shown in Algorithms 2 and 3. Function `traverse-trafo` mainly decomposes the input term into a function symbol (the traversal function), the subject term to be traversed and optional arguments. It then delegates actual work to the function `visit`.

Function `visit` distinguishes two major cases: top-down and bottom-up traversal. In both cases the break/continue behavior of the traversal function has to be modeled. If an application of a traversal function has not failed the recursion either continues or breaks, depending on the annotation of the traversal function. If the application has failed it always continues the recursion.

We apply the traversal function by reusing the `reduce` function from the basic innermost rewriting algorithm (see Algorithm 1). It is applied either before or after traversing the children, depending on the traversal strategy (`bottom-up` or `top-down`). `visit` depends on `visit-children` for recurring over all the children of the current node. If none of the children are reduced `visit-children` returns `fail`, otherwise it returns the list of new children.

In order to be type-safe, the type of the traversal function follows the type of the term is being traversed. Its type always matches the type of the node that is currently being visited. This behavior is encoded by the `typed-compose` function. Transformers are type-preserving, therefore the type of the first argument and the result are adapted to the type of the node that is currently being visited. Note that using this algorithm this we can reuse the existing matching functionality.

The following auxiliary functions are used but not defined in these algorithms:

—`traversal-strategy(fn : function-symbol)` returns the traversal strategy of the given function symbol `fn`, i.e., `top-down` or `bottom-up`.

—`traversal-continuation(fn : function-symbol)` returns the continuation style of the given function symbol `fn`, i.e., `break` or `continue`.

—`result-type-of(t : term)` returns the result type of the outermost function symbol of the given term `t`.

4.3 Accumulator

The function `traverse-accu` and its auxiliary functions are shown in Algorithms 4 and 5. The definitions largely follow the same pattern as for transformers, with the following exceptions:

—`traverse-accu` not only separates the traversed subject from the arguments of the traversal function. It also identifies the second argument as the initial value of the accumulator.

Algorithm 4 An interpreter for accumulators, Part 1.

```
function traverse-accu(t : term, rules : list-of[rule]) : term
begin
  var trfn    : function-symbol;
  var subject : term;
  var args    : list-of[term];

  decompose term t as trfn(subject, accu, args)
  return visit(trfn, subject, accu, args, rules);
end

function visit(trfn : function-symbol, subject : term, accu : term,
               args : list-of[term], rules : list-of[rule]) : term
begin
  var reduct, accu' : term;

  if traversal-strategy(trfn) = TOP-DOWN
  then accu' := reduce(typed-compose(trfn, subject, accu, args), rules);
     if accu' = fail
     then return visit-children(trfn, subject, accu, args, rules)
     else if traversal-continuation(trfn) = BREAK
        then return accu'
        else reduct = visit-children(trfn, accu', reduct, args, rules)
           return if reduct = fail then accu' else reduct fi
        fi
     fi
  else /* BOTTOM-UP */
     accu'  := visit-children(trfn, subject, accu, args, rules);
     if accu' = fail
     then reduct := reduce(typed-compose(trfn, subject, accu, args), rules);
     else if traversal-continuation(trfn) = BREAK
        then return accu'
        else reduct := reduce(typed-compose(trfn, subject, accu', args), rules);
           return if reduct = fail then accu' else reduct fi
        fi
     fi
  fi
end
```

—Both `visit` and `visit-children` have an extra argument for the accumulator.

—In `typed-compose` only the type of the first argument is changed while the type of the accumulator argument remains the same.

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

4.4 Accumulating Transformer

We do not give the details of the algorithms for the accumulating transformer since they are essentially a fusion of the algorithms for accumulators and transformers. Since an accumulating transformer has two input and output values (the initial term and the current accumulator value, respectively, the transformed term and the updated accumulator value), the types of `visit`, `visit-children` and `typed-compose` have to be adjusted to manipulate a pair of terms rather than a

Algorithm 5 An interpreter for accumulators, Part 2.

```
function visit-children(trfn : function-symbol, subject : term, accu : term,
                        args : list-of[term], rules : list-of[rule]) : term
begin
  var children                : list-of[term];
  var child, accu', reduct    : term;
  var fn                      : id;
  var success                 : bool;

  decompose term subject as fn(children);
  accu' := accu; success := false;
  foreach child in children
  do reduct  := visit(trfn, child, accu', args, rules);
     if reduct != fail then success = true; accu' := reduct fi
  od;
  return if success = true then accu' else fail fi
end

function typed-compose(trfn : function-symbol, subject : term, accu : term,
                       args : list-of[term]) : term
begin
  var τ1, τ2, ..., τn, τsubject : type;
  var rsym : function-symbol;
  var fn   : id;

  τsubject := result-type-of(subject);
  decompose function-symbol trfn as fn: τ1 × τ2 × ... × τn -> τ2 ;
  rsym := compose function-symbol fn: τsubject × τ2 × ... × τn -> τ2;
  return compose term rsym(subject, accu, args);
end
```

single term.

4.5 Discussion

In the above presentation we have separated the three cases transformer, accumulator and accumulating transformer. In an actual implementation, these three cases can be implemented by a single function that uses pairs of terms (to accommodate accumulating transformers).

The algorithms become slightly more involved since the algorithms for transformer and accumulator now have to deal with term pairs and in several places case distinctions have to be made to cater for the specific behavior of one of the three algorithms.

5. IMPLEMENTATION ISSUES

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

—Parsing the user-defined rules of a traversal function (Section 5.1).

—An interpreter-based implementation of traversal functions (Section 5.2).

—A compilation scheme for traversal functions (Section 5.3).

5.1 Parsing Traversal Functions

The terms used in the rewrite rules of ASF+SDF have 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 (described in [Brand et al. 2001]) was to generate the syntax for any possible application of a traversal function. This collection of generated functions could be viewed as one *overloaded* function. This simple approach relieved the programmer from typing in the trivial productions himself. In practice, this solution had two drawbacks:

—The parse tables tended to grow by a factor equal to the number of traversal functions. As a result, interactive development became unfeasible because the parse table generation time was growing accordingly.

—Such generated grammars were possibly ambiguous. Disambiguating grammars is a delicate 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 that we finally adopted is to let the user specify the grammar rule for each sort that is used as *argument* of the traversal function: this amounts to *rewrite rules* defining the traversal function and *applications* of the traversal function in other rules. 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 being visited. The parse table will now only grow proportionally to the number of visited node types. As a result the parse table generation time will be acceptable for interactive development.

We have opted for the latter solution since we are targeting industrial size problems with traversal functions and solutions that work only for small examples are not acceptable. The above considerations are only relevant for term rewriting with concrete syntax. Systems that have fixed term syntax can generate the complete signature without introducing any significant overhead.

5.2 Interpretation of Traversal Functions

The ASF interpreter 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.

The ASF+SDF interpreter implements the algorithms as presented in Section 4.

5.3 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 [Brand et al. 1999; Brand et al. 2002] translates rewrite rules to C functions. The compiled specification takes a parse tree as input and produces a parse tree as result. Internally, a more

dense abstract term format is used. After compilation, the run-time behavior of a rewriting system is as follows:

(1) In a bottom-up fashion, each node in the input parse tree is visited and the corresponding C function is retrieved and called immediately. This retrieval is implemented by way of a pre-compiled dictionary that maps function symbols to the corresponding C function. During this step the conversion from parse tree to abstract term takes place. The called function contains a dedicated matching automaton for the left-hand sides of all rules that have the function symbol of this node as outermost symbol. 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.

(2) When an application of a C function fails, this means that this node is in normal form. As a result, the normal form is explicitly constructed in memory. Nodes for which no rewrite rules apply, including the constructors, have this as standard behavior.

(3) Finally, the resulting normal form in abstract term format is translated back to parse tree format using the dictionary.

Traversal functions can be fitted in this run-time behavior in the following manner. For every defining rewrite rule of a traversal function and for every call to a traversal function the type of the overloaded argument and optionally the result type is turned into a single universal type. The result is a collection of rewrite rules that all share the same outermost traversal function, which can be compiled using the existing compilation scheme to obtain a matching automaton for the entire traversal function.

Figure 19 clarifies this scheme using a small example. The first phase shows a module containing a traversal function that visits two types `A` and `B`. This module is parsed, type-checked and then translated to the next module (pretty-printed here for readability). In this phase all variants of the traversal function are collapsed under a single function symbol. The `"_"` denotes the universally quantified type.

The traversal function in this new module is type-unsafe. In `[2]`, the application of the traversal function is guarded by the `b` constructor. Therefore, this rule is only applicable to such terms of type `B`. The other rule `[1]` is not guarded by a constructor. By turning the type of the first argument of the traversal function universal, this rule now matches terms of *any* type, which is not faithful to the semantics of ASF+SDF.

The solution is to add a run-time type-check in cases where the first argument of a traversal function is not guarded. For this we can use the dictionary that was described above to look up the types of symbols. The new module is shown in the third pane of figure 19. A condition is added to the rewrite rule, stipulating that the rule may only succeed when the type of the first argument is equal to the expected type. The `type-of` function encapsulates a lookup in the dictionary that was described above. It takes the top symbol of the term that the variable matched and returns its type. This module can now be compiled using the conventional compiler to obtain a type-safe matching automaton for all defining rules of the traversal function.

```
module Example
exports
  context-free syntax
    a          -> A
    b ( A )    -> B
    example(A) -> A {traversal(trafo,bottom-up,continue)}
    example(B) -> B {traversal(trafo,bottom-up,continue)}
variables
    "VarA" -> A
equations
[1] example(VarA)    = ...
[2] example(b(VarA)) = ...
```

```
module Example
exports
  context-free syntax
    a         -> A
    b ( A )   -> B
    example(_) -> _ {traversal(trafo,bottom-up,continue)}
variables
    "VarA" -> A
equations
[1] example(VarA)    = ...
[2] example(b(VarA)) = ...
```

```
module Example
exports
  context-free syntax
    a         -> A
    b ( A )   -> B
    example(_) -> _ {traversal(trafo,bottom-up,continue)}
equations
[1] type-of(VarA) = A  ===> example(VarA) = ...
[2] example(b(VarA)) = ...
```

```
ATerm example(ATerm arg0)
{
  ATerm tmp0 = call_kids_trafo(example, arg0, NO_EXTRA_ARGS);

  if (check_symbol(tmp0, b_symbol)) { /* [2] */
    return ...;
  }
  if (term_equal(get_type(tmp0), type("A"))) { /* [1] */
    return ...;
  }

  return tmp0;
}
```

Fig. 19. Selected phases in the compilation of a traversal function.

To obtain the tree traversal behavior this automaton is now combined with calls to a small run-time library. It contains functions that take care of actually traversing the tree and optionally passing along the accumulated argument. The fourth pane of figure 19 shows the C code for the running example.

Depending on the traversal type there is a different run-time procedure. In this case it is a transformer, so `call_kids_trafo` is used. For a transformer the function is applied to the children, and a new node is created after the children are reduced. For an accumulator the library procedure, `call_kids_accu`, also takes care of passing along the accumulated value between the children. Depending on the traversal order the calls to this library are simply made either before or after the generated matching automaton. The `break` and `continue` primitives are implemented by inserting extra calls to the run-time library procedures surrounded by conditionals that check the successful application or the failure of the traversal function.

6. EXPERIENCE

Traversal functions have been applied in a variety of projects. We highlight some representative ones.

6.1 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 [Zaadnoordijk 2001; Veerman 2003]. This is based on earlier work described in [Sellink et al. 1999]. 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 deprecated language features and to achieve code improvements. Examples of rules are:

—Adding `END-IF` keywords to close `IF`-statements.

—Replace nested `IF`-statements with `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).

```
module End-If-Trafo
imports Cobol
exports
context-free syntax
  addEndIf(Program)  -> Program {traversal(trafo,continue,top-down)}
variables
 "Stats"[0-9]*    -> StatsOptIfNotClosed
 "Expr"[0-9]*     -> L-exp
 "OptThen"[0-9]*  -> OptThen
equations
[1] addEndIf(IF Expr OptThen Stats)  =
             IF Expr OptThen Stats END-IF

[2] addEndIf(IF Expr OptThen Stats1 ELSE Stats2) =
             IF Expr OptThen Stats1 ELSE Stats2 END-IF
```

Fig. 20. Definition of rules to add END-IFs.

Each transformation rule is implemented by means of a traversal function defined by only a few equations. Figure 20 shows two rewrite rules which add the missing `END-IF` keywords to the COBOL conditionals.

The complete transformation took two and a half hours using the ASF interpreter.[5] The compiled version of traversal functions was not yet ready at the time this experiment was done but it would reduce the time by a factor of at least 30–40 (see Section 6.3). The estimated compiled execution time would therefore be under 5 minutes. These results show that traversal functions can be used effectively to solve problems of a realistic size.

6.2 SDF Re-factoring

In [Lämmel and Wachsmuth 2001] 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."

6.3 SDF Well-formedness Checker

SDF is supported by a tool-set[6] containing among others a parse table generator and a well-formedness checker. A considerable part of the parse table generator is specified in ASF+SDF. The well-formedness checker is entirely specified in ASF+SDF

[5] On a 333 MHz PC with 192 Mb of memory running Linux.
[6] `www.cwi.nl/projects/MetaEnv`

Table II. Performance of the SDF checker.

Grammar	# of productions	Interpreted (seconds)	Compiled (seconds)	Ratio
SDF	200	35	0.85	42
Java	352	215	1.47	146
Action Semantics	249	212	2.00	106
COBOL	1251	1586	5.16	307

and makes extensive use of traversal functions. The well-formedness checker analyses a collection of SDF modules and checks, among others, for completeness of the specification, sort declarations (missing, unused, and double), uniqueness of constructors, and uniqueness of labels. The SDF grammar consists of about 200 production rules, the ASF+SDF specification consists of 150 functions and 186 equations, 66 of these functions are traversal functions and 67 of the equations have a traversal function as the outermost function symbol in the left-hand side and can thus be considered as "traversal" equations.

An indication of the resulting performance is shown in Table II.[7] It shows results for SDF, Java, Action Semantics and COBOL. For each grammar, the number of grammar rules is given as well as execution times (interpreted and compiled) for the SDF checker. The last column gives the interpreted/compiled ration. These figures show that traversal functions have a completely acceptable performance. They also show that compilation gives a speed-up of at least a factor 40.

7. DISCUSSION

Traversal functions are based on a minimalist design that tries to combine type safety with expressive power. We will now discuss the consequences and the limitations of this approach.

7.1 Declarative *versus* Operational Specifications

Traversal functions are expressed by annotating function declarations. 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 [Brand et al. 2001] 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.

[7] On a 333 MHz PC with 192 Mb of memory running Linux.

7.2 Expressivity

Recall from figure 1 the main left-to-right visiting orders for trees: top-down and bottom-up combined with two stop criteria: stop after first application or visit all nodes. All of these orders can be expressed by traversal functions using combinations of `bottom-up`, `top-down`, `break` and `continue`. We have opted for a solution that precisely covers all these possible visiting orders.

One may wonder how concepts like *repetition* and *conditional evaluation*, as used in strategic programming (see Section 1.7), fit in. In that case, *all* control structures are moved to the strategy language and the base language (rewrite rules, functions) remains relatively simple. In our case, we use a base language (ASF+SDF) that is already able to express these concepts and there is no need for them to be added to the set of traversal primitives.

7.3 Limited Types of Traversal Functions

Accumulators can only map sub-trees to a *single* sort and transformers can only do sort preserving transformations. Is that a serious limitation?

One might 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 sub-term. However, in the case of *full* translations the advantage of *any* generic traversal scheme is debatable, since translation rules have to be given for any language construct anyway. A more 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. It is therefore difficult to see how a generic non-sort-preserving traversal primitive could really make specifications of translations more concise.

7.4 Reuse *versus* Type-safety

We do not separate the traversal strategy from the rewrite rules to be applied. By doing so, we loose the potential advantage of reusing the same set of rewrite rules under different visiting orders. However, precisely the *combination* of traversal strategy and rewrite rules allows for a simple typing mechanism. The reason is that the generic traversal attributes are not separate operators that need to be type-checked. It allows us to ensure well-formedness in both type-preserving transformations and in type-unifying computations without extending the typing mechanisms of our first-order specification language.

7.5 Conclusions

We have described term rewriting with traversal functions as an extension of 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.

Traversal functions are thus a nice compromise between simplicity and expressive power.

The main disadvantage of our approach might manifest itself when dealing with visiting orders that go beyond our basic model of tree traversal. Two escapes would be possible in these cases: such traversals could either be simulated as a modification of one of the built-in strategies (by adding conditions or auxiliary functions), or one could fall back to the tedious specification of the traversal by enumerating traversal rules for all constructors of the grammar.

In practice, these scenario's have not occurred and experience with traversal functions shows that they are extremely versatile when solving real-life problems.

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 re-factoring. Ralf Lämmel, Eelco Visser and Joost Visser commented on drafts of this paper. The feedback by the anonymous referees greatly improved the presentation of the paper.

REFERENCES

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

BERGSTRA, J. A., HEERING, J., AND KLINT, P., Eds. 1989. *Algebraic specification*. ACM Press/Addison-Wesley.

BOROVANSKÝ, P., KIRCHNER, C., KIRCHNER, H., MOREAU, P.-E., AND RINGEISSEN, C. 1998. An overview of ELAN. In *International Workshop on Rewriting Logic and its Applications*, C. Kirchner and H. Kirchner, Eds. Electronic Notes in Theoretical Computer Science, vol. 15. Elsevier.

BRAND, M. V. D., DEURSEN, A. V., KLINT, P., KLUSENER, S., AND MEULEN, E. V. D. 1996. Industrial applications of ASF+SDF. In *Algebraic Methodology and Software Technology (AMAST '96)*, M. Wirsing and M. Nivat, Eds. LNCS, vol. 1101. Springer-Verlag, 9–18.

BRAND, M. V. D., HEERING, J., KLINT, P., AND OLIVIER, P. 2002. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems 24,* 4, 334–368.

BRAND, M. V. D., KLINT, P., AND OLIVIER, P. 1999. Compilation and memory management for ASF+SDF. In *Compiler Construction*. Lecture Notes in Computer Science, vol. 1575. Springer-Verlag, 198–213.

BRAND, M. V. D., KLINT, P., AND VERHOEF, C. 1996. Core technologies for system renovation. In *SOFSEM'96: Theory and Practice of Informatics*, K. Jeffery, J. Král, and M. Bartošek, Eds. LNCS, vol. 1175. Springer-Verlag, 235–255.

Brand, M. v. d., Klint, P., and Verhoef, C. 1998. Term rewriting for sale. In *Second International Workshop on Rewriting Logic and its Applications, WRLA 98*, C. Kirchner and H. Kirchner, Eds.

Brand, M. v. d., Klint, P., and Vinju, J. 2001. Term rewriting with traversal functions. Tech. Rep. SEN-R0121, Centrum voor Wiskunde en Informatica.

Brand, M. v. d., Sellink, M., and Verhoef, C. 2000. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming 36*, 209–266.

Cardelli, L. 1997. Type systems. In *Handbook of Computer Science and Engineering*. CRC Press.

Cordy, J., Halpern-Hamu, C., and Promislow, E. 1991. TXL: A rapid prototyping system for programming language dialects. *Computer Languages 16,* 1, 97–107.

Deursen, A. v., Heering, J., and Klint, P., Eds. 1996. *Language prototyping: an algebraic specification approach*. AMAST Series in Computing, vol. 5. World Scientific.

Deursen, A. v., Klint, P., and Tip, F. 1993. Origin tracking. *Journal of Symbolic Computation 15*, 523–545.

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

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

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusets.

Hedin, G. 1992. Incremental semantic analysis. Ph.D. thesis, Lund University.

Hedin, G. and Magnusson, E. 2001. JastAdd - a Java-based system for implementing frontends. In *Proc. LDTA'01*, M. van den Brand and D. Parigot, Eds. Electronic Notes in Theoretical Computer Science, vol. 44-2. Elsevier Science.

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

Heering, J. 1996. Second-order term rewriting specification of static semantics: An exercise. In *Language Prototyping*, A. van Deursen, J. Heering, and P. Klint, Eds. AMAST Series in Computing, vol. 5. World Scientific, 295–305.

Heering, J., Hendriks, P., Klint, P., and Rekers, J. 1989. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices 24,* 11, 43–75.

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

Klint, P. 2001. Is strategic programming a viable paradigm? In *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, B. Gramlich and S. Lucas, Eds. Electronic Notes in Theoretical Computer Science, vol. 57/2. Elsevier Science Publishers.

Klint, P. 2003. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*. IEEE Computer Society, 2–12.

Knuth, D. 1968. *The Art of Computer Programming, Volume 1*. Addison-Wesley.

Kuipers, T. and Visser, J. 2001. Object-oriented tree traversal with JJForester. In *Electronic Notes in Theoretical Computer Science*, M. van den Brand and D. Parigot, Eds. Vol. 44. Elsevier Science Publishers. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

Lämmel, R. 2003. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming 54*, 1–64.

Lämmel, R. and Visser, J. 2002. Typed combinators for generic traversal. In *PADL 2002: Practical Aspects of Declarative Languages*. Lecture Notes in Computer Science (LNCS), vol. 2257. Springer.

Lämmel, R., Visser, J., and Kort, J. 2000. Dealing with large bananas. In *Workshop on Generic Programming*, J. Jeuring, Ed. Ponte de Lima. Published as Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.

Lämmel, R. and Wachsmuth, G. 2001. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In *Proc. LDTA'01*, M. van den Brand and D. Parigot, Eds. Electronic Notes in Theoretical Computer Science, vol. 44-2. Elsevier Science.

Sellink, M., Sneed, H., and Verhoef, C. 1999. Restructuring of COBOL/CICS legacy systems. In *Proceedings of Conference on Maintenance and Reengineering (CSMR'99)*. Amsterdam, 72–82.

Terese. 2003. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Tip, F. and Dinesh, T. 2001. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology 10*, 5–55.

Veerman, N. 2003. Revitalizing modifiability of legacy assets. In *7th European Conference on Software Maintenance and Reengineering*, M. van den Brand, G. Canfora, and T. Gymóthy, Eds. IEEE Computer Society Press, 19–29.

Visser, E. 2000. Language independent traversals for program transformation. In *Workshop on Generic Programming*, J. Jeuring, Ed. Ponte de Lima, 86–104. Published as Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.

Visser, E. 2001a. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01)*, A. Middeldorp, Ed. Lecture Notes in Computer Science. Springer-Verlag.

Visser, E. 2001b. A survey of strategies in program transformation systems. In *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, B. Gramlich and S. Lucas, Eds. Electronic Notes in Theoretical Computer Science, vol. 57/2. Elsevier Science Publishers.

Visser, J. 2001c. Visitor combination and traversal control. *ACM SIGPLAN Notices 36,* 11 (Nov.), 270–282. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.

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

Zaadnoordijk, H. 2001. Source code transformations using the new ASF+SDF meta-environment. M.S. thesis, University of Amsterdam, Programming Research Group.

Received Month Year; revised Month Year; accepted Month Year;