Term Rewriting with Type-safe Traversal Functions

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

Centrum voor Wiskunde en Informatica
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands

Abstract

Term rewriting is an appealing technique for performing program analysis and program transformation. Tree (term) traversal is frequently used but is not supported by standard term rewriting. In this paper, many-sorted first-order term rewriting is extended with automatic tree traversal by adding two primitive tree traversal strategies and complementing them with three types of traversals. These so-called traversal functions can be either top-down or bottom-up. They can be sort preserving, mapping to a single sort, or a combination of these two. Traversal functions have a simple design, their application is type-safe in a first-order many-sorted setting and can be implemented efficiently. We describe the operational semantics of traversal functions and discuss applications.

1 Introduction

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 a tree: to visit all the nodes of a tree and extract information from some nodes or make changes to certain other nodes.

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

This problem asks for a form of automation that takes care of the tree traversal itself so that the human programmer can concentrate on the few node categories where real work is to be done. Stated differently, we want to offer tree traversal as a built-in, not a burden, to the programmer.

From previous experience [3,5,6] we know that term rewriting is a convenient, scalable technology for expressing analysis, transformation, and renovation of individual programs and complete software systems. In this paper we address therefore the question how tree traversals can be added to the term rewriting paradigm.

One important requirement is to have a typed design of automated tree traversals, such that terms are always well-formed. Another requirement is to have simplicity of design and use. In particular, we prefer to remain inside the domain of first-order specifications.

In the remainder of this introduction we will briefly recapitulate term rewriting, and discuss why traversal functions are necessary in term rewriting. In Section 2 we extend the algebraic specification formalism Asf+Sdf [1,9] with traversal functions and in Section 3 we give some small examples of the use of these traversal functions. The operational semantics of traversal functions is given in Section 4. Section 5 describes the experience with traversal functions and Section 6 gives a discussion and also discusses related work.

A brief recapitulation of term rewriting

Algorithm 1 An interpreter for innermost rewriting.

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

A basic insight in term rewriting is important for understanding the traversal functions described in this paper. Therefore we give a brief and informal recapitulation of innermost term rewriting. For a full account see [11].

A *term* is a prefix expression consisting of constants (e.g., `a` or `12`), variables (e.g., `X`) or function applications (e.g., `f(a, X, 12)`). For simplicity, we will view constants as nullary functions. A *closed term* is a term without variables. 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 [1] (e.g., `f(a, X)` matches `f(a, b)`) and results in a *binding* (e.g., `X` is bound to `b`). The bindings resulting from matching can be used for *substitution*, i.e., replace the variables in a term by the values they are bound to.

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

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

The operation of a rewrite rule interpreter is shown in more detail in Algorithm 1. The functions *match* and *substitute* are not further defined, but have a meaning as just sketched. The functions *decompose* and *compose* manipulate terms and also rules, and *append* appends an element to the end of a list.

Note that the underlying term representation can be either typed or untyped. The *match*, *substitute*, *decompose* and *compose* functions have to take this into account.

Observe how function *innermost* first reduces the children of the current term before attempting to reduce the term itself. This realizes a bottom-up traversal of the term. Also note that if the reduction of the term fails, it returns itself as result. The function *reduce* performs, if possible, one reduction step. It searches all rules for a matching left-hand side and, if found, the bindings resulting from the successful match are substituted in the corresponding right-hand side. This modified right-hand side is then further reduced with innermost rewriting.

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

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 1 displays an SDF grammar [10] 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`. Transformations on these trees can now be defined easily. For instance, if we want to replace all occurrences of `f` by `h`, then the following rule suffices:

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

[1] A matching algorithm has to do some extra work if a specific variable name occurs more than once in a pattern. Usually the different occurrences are renamed to fresh variables and equality checks are added.

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

Figure 1: SDF grammar for simple tree language.

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

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

Similarly, if we want to replace all 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] f(g(T1, T2), T3) = h(T1, h(T2, T3))
```

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

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

Note how in both cases the standard (innermost) reduction order of the rewriting system takes care of the complete traversal of the term. Such elegant specifications have, 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.
- The second problem is that rewrite rules cannot access any context information. In other words a rewrite rule has only one parameter, which is the term that matches the left-hand side. Especially for program transformation this is very limiting.
- 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.

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

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

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

Figure 2: Definition of `trafo1` and `trafo2`.

`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. This adds *data-flow* to a specification. Finally, the function symbols allow to express non-type-preserving transformations by explicitly typing the function to accept one type and yield another.

So by introducing first-order functions, 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 programmer. Extra rewrite rules are needed to define the traversal of `trafo1` and `trafo2` over the input term, as shown in Figure 2. Observe that equations `[2]` and `[5]` in the figure correspond to the original equations `[t1]` and `[t2]`, respectively. The other equations are just needed to define the tree traversal. 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 language shown in Figure 1. For large grammars this is clearly undesirable.

The traversal functions presented in this paper, solve the problem of the extra rules needed for term traversal without loosing the practical abilities of first-order functions to carry data around and having non-sort-preserving transformations.

2 Traversal functions in Asf+Sdf

We want to automate tree traversal in the many-sorted, first-order term rewriting language Asf+Sdf [1,9]. Asf+Sdf uses context-free syntax for defining the signature of terms. As a result, terms can be written in arbitrary user-defined notation. The context-free syntax is defined in Sdf[2]. Terms are used in

[2] Syntax Definition Formalism.

rewrite rules defined in ASF [3]. For the purpose of this paper, the following features of ASF are relevant:

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

The idea of traversal functions is as follows. 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 functionality provided by the `traversal` attribute thus defines 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 some sub-terms. This is a nice way of defining the semantics of traversal functions. More details can be found in [7].

The question is what sort of traversals we can provide in our fully typed setting. We allow three *types* of functions for the visiting behavior and two types of traversal *strategies* which we now discuss in order. The merits and limitations of this approach are discussed in Section 6. For extensive examples we refer the reader to [7].

Traversal functions are partitioned into three *types*, defined as follows:

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

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

Because a transformer always returns the same sort, it is type-safe. A transformer is used to transform a tree.

[3] Algebraic Specification Formalism.

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

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

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

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

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

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

An accumulating transformer is used to simultaneously extract information from a tree and transform it.

Transformers, accumulators, and accumulating transformers may be overloaded in their first argument to obtain visitors for heterogeneous trees. This means that a single traversal function can visit different types of nodes in a single traversal.

The optional extra arguments of traversal functions 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.

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

Bottom-up: the function traverses *all* the sub-trees of a node while its rewrite rules are used to visit the nodes in a *bottom-up* fashion. The annotation `bottom-up` selects this behavior. A traversal function without an explicit indication of a visiting strategy also uses the bottom-up strategy.

Top-down: the function traverses the sub-trees of a node in a top-down fashion and stops recurring at the first node where one of its rewrite rules applies and does not go to any sub-trees of that node. The annotation `top-down` selects this behavior.

A transformer with a `bottom-up` strategy resembles standard innermost rewriting; it is sort preserving and bottom-up. It is as if a small rewriting

```
module Tree-inc
imports Tree-syntax
exports
context-free syntax
  inc(TREE) -> TREE {traversal(trafo)}
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 ))`

Figure 3: The transformer `inc` increments each number in a tree.

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

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

Figure 4: The accumulator `sum` computes the sum of all numbers in a tree.

system is defined within the context of a transformer function. The difference is that a transformer function inflicts one reduction on a node, while innermost reduction normalizes a node completely.

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

Note that the horizontal ordering (left-to-right or right-to-left), might also be a parameter. In this work we have focused on left-to-right traversals.

3 Examples

After this general description of traversal functions, it is time to illustrate these concepts. We will give a very simple example of each type of traversal function just described. More realistic examples are presented in Section 5.

Increment the numbers in a tree. The specification in Figure 3 shows the transformer `inc`. Its purpose is to increment all numbers that occur in a tree. The results of applying `inc` to a sample tree are also shown in Figure 3.

Add the numbers in a tree. The second example shows the use of the accumulator. The problem we want to solve is computing the sum of all numbers that occur in a tree. The accumulator `sum` in Figure 4 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).

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

in	pos(f(g(f(1,2), 3),
	     g(g(4,5), 6)),
	   0)
		out	<f(g(f(0,2), 6),
			   g(g(12,20), 30)),
			 6>

Figure 5: The accumulating transformer pos multiplies each number by its position in the tree.

Multiply by position in tree. The last example shows the use of the accumulating transformer. The traversal functions is to determine the position of each number in a bottom-up traversal of the tree and to multiply each number by its position. This is achieved by the accumulating transformer pos shown in Figure 5. 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.

4 Operational Semantics

Algorithm 2 An extended interpreter for innermost rewriting.

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

In this section we show an operational semantics for traversal functions. The reader is referred to [7] for a different style of semantics, namely expressing traversal functions in terms of normal rewriting systems. The semantics in this section is better suited as a reference for implementation.

We start with normal innermost rewriting as depicted earlier in Algorithm 1. In this algorithm we do assume we have a typed term representation, because we want to have *type-safe* traversal functions. This means that with every constructor and with every function symbol a first order type can be associated. For example, a function f could have type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau_r$. If $n = 0$, f is a constant of type τ_r. If $n > 0$, f is either a constructor or a function with its arguments typed by $\tau_1, ..., \tau_n$ respectively. If we allow tupled terms, we should also define the types of tuples. A tuple type is simply denoted by $(\tau_1 \times \tau_2)$. Of course, the *match* function should take care to match only terms that have matching types. Our *compose* function gets an extra argument denoting which type of function it should use to construct a term.

The main algorithm consists of the normal innermost reduction strategy, but we switch to a different mode when a traversal function is encountered. The switch statement in Algorithm 2 detects a traversal function and turns over control to a function called *traverse*, instead of calling *reduce*.

This function is shown in Algorithm 3. It initiates the traversal with different parameters for each kind of traversal function. Recall that the input term is of the form $\mathit{trfn}(T_1, T_2, ..., T_n)$ $(n >= 1)$ where *trfn* is a traversal function, T_1 is the term to be traversed, T_2 is the (optional) accumulator argument, and $T_3, ..., T_n$ are the (optional) remaining arguments. Actual traversal is done by *td-or-bu* ("top-down or bottom-up") that uses either *top-down* or *bottom-up* depending on the traversal strategy of *trfn*. The arguments of *td-or-bu* are determined by the different kinds of traversals.

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

In order to be type-safe, the type of the traversal function changes while the term is traversed. Its type always matches the type of the node that is currently visited. This behavior is encoded by the *type-compose* function. For *type-preserving* transformers, the type of the first argument and the resulting argument are adapted to the type of the node that is currently visited. Similarly, for the accumulator only the type of the first argument is changed while the type of the accumulated argument remains equal. Finally, for the combination the type of the first argument and the resulting tuple type are updated.

The traversal of children in function *visit-children* takes into account that the accumulated value must be passed on between each child. Note that in case of a transformer, this accumulated value is ignored by passing always the value **nil**.

After a successful application of a user-defined rule, the function *make-reduct* decides what to do with the reduct depending on the type of traversal. If it is a transformer, the reduct replaces the redex. In case of an accumulator, the reduct replaces the accumulated argument. If it is an accumulating transformer, the first element of the tuple replaces the redex, while the second element replaces the accumulated argument.

Finally, when we return from the traversal, the top level function *traverse* returns a different normal form for each type of traversal function. In case of a transformer, the transformed term is simply returned. When it is an accumulator, we return the accumulated argument. An accumulating transformer returns a tuple of the transformed term and the final value of the accumulated argument.

Algorithm 3 An interpreter for traversal functions (part 1/2).

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

5 Experience

Various experiments have been carried out including transformations on COBOL, an SDF checker, SDF re-factoring, and Java re-factoring. We discuss here the COBOL example in more detail.

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 [18]. This is based on earlier work described in [15]. 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

Algorithm 3 An interpreter for traversal functions (part 2/2).

funct *type-compose*(*fn*, *term*, *accu*, *args*) $\equiv$
  $\tau_t := \textit{result-type-of}(\textit{term})$
  switch *traversal-kind*(*fn*)
    case "`trafo`" :
      $\tau_1 \times \cdots \times \tau_n \to \tau_1 := \textit{type-of}(\textit{fn})$
      $\textit{type} := \tau_t \times \cdots \times \tau_n \to \tau_t$
    case "`accu`" :
      $\tau_1 \times \tau_2 \times \cdots \times \tau_n \to \tau_2 := \textit{type-of}(\textit{fn})$
      $\textit{type} := \tau_t \times \tau_2 \times \cdots \times \tau_n \to \tau_2$
    case "`accu, trafo`" :
      $\tau_1 \times \tau_2 \times \cdots \times \tau_n \to (\tau_1 \times \tau_2) := \textit{type-of}(\textit{fn})$
      $\textit{type} := \tau_t \times \tau_2 \times \cdots \times \tau_n \to (\tau_t \times \tau_2)$;
  return *compose*(*fn*, *type*, [*term*, *accu*, *args*]).

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.

This collection of source-to-source transformations were formalized as a number of traversal functions. Every function performs a tiny sub-task. Examples of such sub-tasks 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 in a fixed-point computation.

Each of these sub-tasks (21 in total) only consists of a few rewrite rules that define the effect of a traversal function on specific statements.

There are several ways of combining these sub-tasks to obtain the COBOL transformation tool that we wanted. Firstly, the sub-tasks can be combined using normal ASF+SDF rules using functional application.

Secondly, each sub-task is an ASF+SDF specification in itself, which can be run on the command-line separately. Combined with a generated parser and a generic pretty printer, such tools can be used to build complete source-to-source transformations tools using simple shell scripts.

In an experiment the transformation tools were applied to a test base of 582 COBOL programs containing 440,000 lines of code to obtain the following results:

- 17,000 `END-IF`s were added.
- 4,000 lines were changed in order to eliminate `CALL`-utilities.
- 1,000 `GO-TO`s have been eliminated (about 65% of all `GO-TO`s).
- The complete transformation took two and a half hours using the ASF

interpreter. 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 15. The estimated compiled execution time would therefore be circa 10 minutes.

The number of productions in the COBOL grammar that has been used in this renovation factory was 600 and the number of transformations performed was 21 as we have seen above. Per transformation step only one traversal over the tree is needed. We can then conclude that the maximum number [4] of rewrite rules that would be needed is $21 \times 600 = 12.600$! In reality the actual number of rewrite was less than 100 thanks to the use of traversal functions.

In other projects the experience with traversal functions seems to be very positive as well. To quote [14]:

> "At the time of writing, the Framework for SDF Transformations (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 Discussion

Traversal functions vs. hand-written code The approach sketched in this paper has substantial advantages over manually writing functions that traverse the tree explicitly.

In contrast with hand-written traversal code, the traversal functions approach is more language-independent, type safe and concise. Language-independence is obtained since the implementation of traversal functions is independent of the source language to which they are being applied. The approach is type safe since only type preserving transformations and limited sort changing transformations can be expressed. It is concise since only transformation rules have to be written for node types that actually require transformation (as opposed to nodes that only have to be visited).

In the case of hand-written code, the programmer has to traverse all nodes in the tree explicitly and the amount of code will become bulky. In addition, this code is dependent on the source language being transformed. Type safety depends on the data structure being used to represent trees. One can either use a generic tree data-type and loose type safety, or one can use distinct data-types for each node type. In the latter case a substantial amount of interfacing and traversal code has to be written. That code lends itself, in principle, to automatic generation. For JAVA, such an approach is described in [12].

Related Work We distinguish four directly related approaches in Figure 6

[4] In practice, this number could be smaller since a hand-written specification could explicitly avoid visits to certain sub-trees.

	Untyped	Typed
Strategy primitives	Stratego [16]	ELAN [2]
Built-in strategies	Renovation Factories [8]	Traversal Functions

Figure 6: Classification of traversal approaches.

and discuss them below. For a more extensive coverage of related work we refer to [7].

ELAN [2] is a language of many-sorted, first-order, rewrite rules extended with a strategy language that controls the application of individual rewrite rules. Its strategy primitives (e.g., don't know choice, don't care choice) allow formulating non-deterministic computations. Currently, ELAN does not support generic tree traversals since they are not easily fitted in with ELAN's type system.

Stratego [17] is an untyped term rewriting language that provides user-defined strategies. Among its strategy primitives are rewrite rules and several generic traversal operators that allow the definition of any tree traversal, such as bottom-up and top-down 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)`. In [13], a proposal is made for a Stratego-like language that allows typed generic traversals for type preserving and type-unifying strategies.

Transformation Factories [8] 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 unspecific types are 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 unspecific types.

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

Stratego extends term rewriting with traversal strategy combinators and user-defined strategies, but 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.

Recently, in [13] 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 a bit more complicated for the user. Two generic types are added to a first-order type system: Type-Preserving and Type-Unifying 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. In the case of traversal functions, extending the type system is not needed because the tree traversal is combined with function application in a single traversal function. This allows the interpreter or compiler to create type-guards automatically. In similar way as for traversal functions, in [13] traversal types are also 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. We have also removed the need for higher order arguments, obtaining precise and simple types. 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 or simulated higher-order arguments.

Compilation of Traversal Functions We have extended the ASF+SDF compiler [4] with traversal functions in a very simple manner. We only briefly sketch the approach here. 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. In a first approximation, the result is one big type-unsafe function which can be compiled using the existing compilation scheme.

Type-safety is achieved as follows. If the first argument of a defining rewrite rule is guarded by a constructor, this will automatically enforce type-safety. If it is not guarded (i.e., the argument is a single variable), we add a condition to the rewrite rule that checks the type of the matched tree at run-time. The compiled code will thus be type-safe.

The resulting compiled code is extended with calls to a small run-time library. They take care of actually traversing the tree and passing along the accumulated argument before or after trying to apply the compiled traversal function.

Conclusions We have described term rewriting with traversal functions as an extension to innermost term rewriting and we have shown how to incorporate this in 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 and easily type-checked.
- Traversal functions can be implemented efficiently.

To summarize, traversal functions are a nice compromise between simplicity and expressive power. For implementation issues and extensive examples we refer to [7].

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

Acknowledgement

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.

References

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

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1998.

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

[4] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 2002. To appear.

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

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

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

[8] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36:209–266, 2000.

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

[10] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

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

[12] T. Kuipers and J. Visser. Object-oriented tree traversal with jjforester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[13] R. Lämmel. Typed Generic Traversals in S'_γ. Technical Report SEN-R0122, CWI, August 2001.

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

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

[16] E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, Lecture Notes in Computer Science. Springer-Verlag, May 2001.

[17] E. Visser, Zine-el-Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

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