J. Symbolic Computation (1993) **15**, 523–545

Origin Tracking

A. VAN DEURSEN, P. KLINT, AND F. TIP
arie@cwi.nl, paulk@cwi.nl, tip@cwi.nl

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We are interested in generating interactive programming environments from formal language specifications and use term rewriting to execute these specifications. Functions defined in a specification operate on the abstract syntax tree of programs and the initial term for the rewriting process will consist of an application of some function (e.g., a type checker, evaluator or translator) to the syntax tree of a program. During the term rewriting process, pieces of the program such as identifiers, expressions, or statements, recur in intermediate terms. We want to formalize these recurrences and use them, for example, for associating positional information with messages in error reports, visualizing program execution, and constructing language-specific debuggers. *Origins* are relations between subterms of intermediate terms and subterms of the initial term. *Origin tracking* is a method for incrementally computing origins during rewriting. We give a formal definition of origins, and present a method for implementing origin tracking.

1 Introduction

We are interested in generating interactive development tools from formal language definitions. Thus far, this has resulted in the design of an algebraic specification formalism, called ASF+SDF [BHK89, HHKR89] supporting modularization, user-definable syntax, associative lists, and conditional equations, and in the implementation of the ASF+SDF Meta-environment [Hen91, Kli93].

Given a specification for a programming (or other) language, the Meta-environment generates an interactive environment for the language in question. More precisely, the Meta-environment is a tool generator which takes a specification in ASF+SDF and derives a lexical analyzer, a parser, a syntax-directed editor and a rewrite engine from it. The Meta-environment provides fully interactive support for writing, checking, and testing specifications—all tools are generated in an incremental fashion and, when the input specification is changed, they are updated incrementally rather than being regenerated

Partial support has been received from the European Communities under ESPRIT project 2177: Generation of Interactive Programming Environments - GIPE II, under ESPRIT project 5399: Compiler Generation for Parallel Machines - COMPARE, and from the Netherlands Organization for Scientific Research - NWO, project *Incremental Program Generators*.

0747–7171/93/5–6523 + 23 $08.00/0

from scratch. A central objective in this research is to maximize the direct use that is made of the formal specification of a language when generating development tools for it.

We use Term Rewriting Systems (TRSs) [Klo91] to execute our specifications. A typical function (such as an evaluator, type checker, or translator) in a specification operates on the abstract syntax tree of a program (which is part of the initial term). During the term rewriting process, pieces of the program such as identifiers, expressions, or statements, recur in intermediate terms. We want to formalize these recurrences and use them, for example, for:

- associating positional information with messages in error reports;
- visualizing program execution;
- constructing language-specific debuggers.

Our approach to formalize recurrences of subterms consists of two stages. First, we define relations for elementary reduction steps $t_i \rightarrow t_{i+1}$; these relations are described in Section 1.3. Then, we extend these relations to complex reduction sequences $t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$. In particular, we are interested in relations between subterms of an intermediate term t_i, and subterms of the initial term t_0. We will call this the *origin relation*. Intuitively, it formalizes from which parts of the initial term a particular subterm originates. The process of incrementally computing origins we will call *origin tracking*.

1.1 Applications of Origin Tracking

In TRSs describing programming languages terms like

```
program(decls(decl(n,natural)), stats(assign(n,34)))
```

are used to represent abstract syntax trees of programs. A typical type check function takes a program and computes a list of error messages. An example of the initial and final term when type checking a simple program is shown in Figure 1.

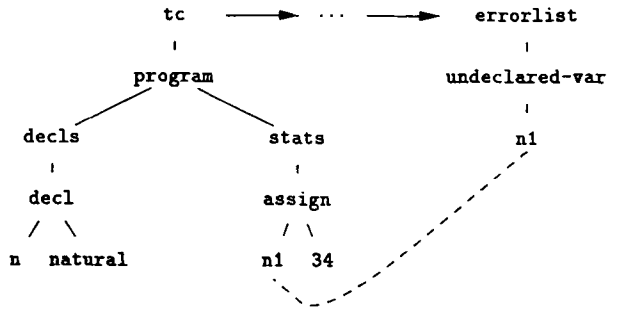


Figure 1: Type checking a simple program

The program uses an undeclared variable `n1`, and the result of the type checker is a term representing this fact, i.e., a term with `undeclared-var` as function symbol and the name `n1` of the undeclared variable as argument. The dashed line represents an origin: it

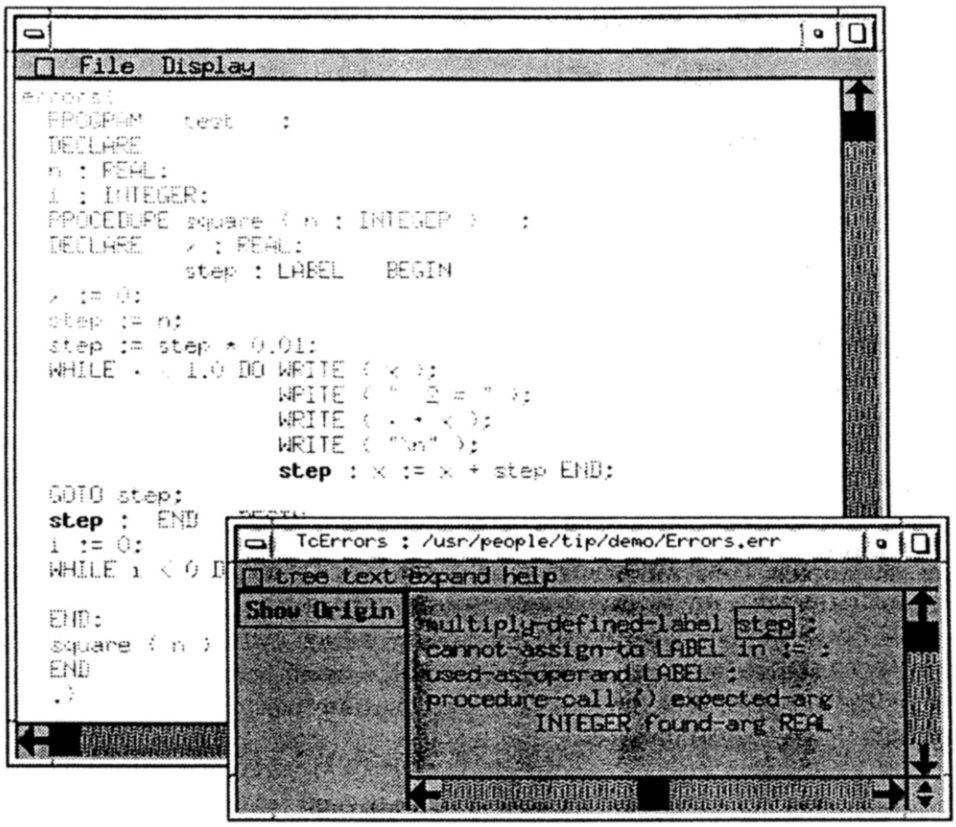


Figure 2: Highlighting occurrences of errors.

relates the occurrence of `n1` in the result to the `n1` in the initial term. One can use this to highlight the exact position in the source program where the error occurred. Figure 2 shows an application of this technique.

Similarly, program evaluators can be defined. Consider for example a rule which evaluates a list of statements by evaluating the first statement followed by the remaining statements:

```
[1] ev-list(cons(Stat,S-list),Env) -> ev-list(S-list, ev-stat(Stat,Env))
```

The variables (*Stat*, *S-list*, and *Env*) are used to pass information from the left to the right-hand side. The origins of these variable occurrences in the right-hand side are shown by dashed lines in Figure 3.

A natural application of origin tracking consists of the animation of program execution. By this we mean that—during execution—the statement currently being executed is indicated in some distinctive way in the source text. In the above example this can be achieved by matching every redex with the pattern `ev-stat(`*Stat*`,` *Env*`)` and, whenever a match occurs, highlighting the origin of the first argument of `ev-stat` in the initial program. Applications of this technique can mainly be found in the areas of source-level debugging

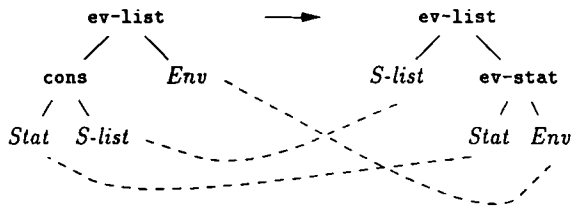


Figure 3: One step in the evaluation of a simple program.

and tutoring. It allows us to animate program execution in a very fine-grained manner: we can visualize the evaluation of any language construct (e.g., expressions, declarations).

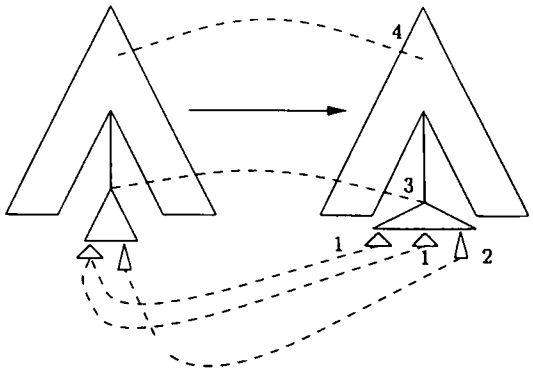
A second usage of origin tracking in the area of debugging is the possibility to define various notions of breakpoints. Source-level debuggers often have a completely fixed notion of a breakpoint, based on line-numbers, procedure calls and machine addresses. The origin relation can be used to define breakpoints in a much more uniform and generic way. For instance, a *positional* breakpoint can be created when the user selects a certain point in the source program. The path from the root to that point is recorded and the breakpoint becomes effective when—in this example—the origin of the first argument of `ev-stat` equals that path. *Position-independent* breakpoints can be defined by using patterns describing statements of a certain form (e.g., an assignment with `x` as left-hand side). The breakpoint becomes effective when the argument of `ev-stat` matches that pattern; its origin shows the position in the original program.

1.2 Points of Departure

Before sketching the notion of origin relation (in the next subsection) we briefly summarize our points of departure:

- No assumptions should be made regarding the choice of a particular reduction strategy.
- No assumptions should be made concerning confluence or termination; origins can be established for arbitrary reductions in any TRS.
- The origin relation should be obtained by a static analysis of the rewrite rules.
- Relations should be established between any intermediate term and the initial term. This implies that relations can be established even if there is no normal form.
- Origins should satisfy the property that if t has an origin t', then t' can be rewritten to t in zero or more steps.
- An efficient implementation should exist.

These requirements do not lead, however, to a unique solution. We will therefore only present one of the possible definitions of origins but we can easily imagine alternative ones.



1. Common Variables
2. Common Subterms
3. Redex - Contractum
4. Contexts

Figure 4: Single-step origin relations.

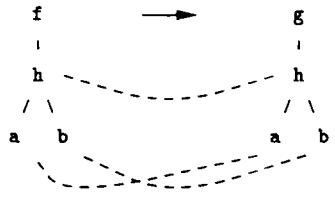


Figure 5: Relations according to a variable occurrence in both sides of a rule.

1.3 Origin Relations

As mentioned above, the origin relation is defined in terms of single-step origin relations for elementary reductions. Put more precisely, the origin relation is based on the transitive and reflexive closure of these single-step origin relations.

To summarize the single-step origin relations, consider Figure 4. Assume a rewrite rule $r : t_1 \to t_2$ is applied in context C with substitution σ, thus giving rise to the elementary reduction $\mathrm{C}[t_1^\sigma] \to_r \mathrm{C}[t_2^\sigma]$. We distinguish four relations:

Common variables: if a variable X appears both in the left-hand side t_1 of the rule and in the right-hand side t_2, then relations are established between each function symbol in the instantiation X^σ of X in t_1 and that same function symbol in each instantiated occurrence of X in t_2.

Figure 5 illustrates how the variable X causes all function symbols in the instantiation to be related when applying the rule `f(X) -> g(X)`. Rules may be non-linear, i.e., some variable X may have multiple occurrences in its left-hand side like in `plus(X,X) -> mul(2, X)`. Since there is no obvious solution to which occurrence of X in the left-hand side the X in the right-hand side should be related, we link it to both occurrences. As a consequence, non-linearity in the left-hand side causes one subterm to have several related subterms.

Common subterms: if a term s is both a subterm of t_1 and of t_2, then these occurrences

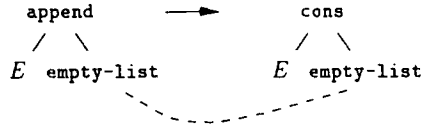


Figure 6: Relations according to common subterms rule.

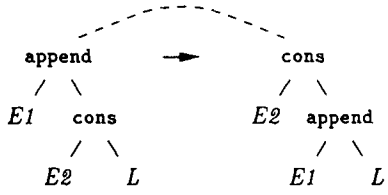


Figure 7: Relations according to redex-contractum rule.

of s are related. Take for example the rule defining how to append an element to the end of a list:

```
[a1] append(E,empty-list) -> cons(E,empty-list)
[a2] append(E1,cons(E2,L)) -> cons(E2, append(E1,L))
```

Using the common variables relation, several useful origin relations will be constructed. But the constant `empty-list` which occurs in both the left-hand side and the right-hand side of `[a1]` will not be related. This is done by the common subterms relation. For equation `[a1]` this is shown in Figure 6. A more elaborate example is the conditional rule for evaluating `while`-statements:

```
[w1] ev-stat(while(Exp,S-list),Env) ->
           ev-stat(while(Exp,S-list),ev-list(S-list,Env))
     when ev-exp(Exp, Env) = true
```

When evaluation of *Exp* yields true, the same while-statement is evaluated in a new environment obtained by evaluating the body of the while-statement (*S-list*) in the initial environment (*Env*). The common subterms relation links the `while`-symbols at both sides of `[w1]`.

Redex-contractum: the top symbol of the redex t_1^q and the top symbol of its contractum t_2^q are related, as shown in Figure 7 for equation `[a2]`. An essential application of this relation can be seen in

```
[r1] real-const(Char-list) -> real-type
```

where a real constant containing a list of characters is rewritten to its type denotation `real-type`. Application of the redex-contractum relation may introduce more than one element in an origin.

Contexts: relations are established between each function symbol in the context C of the left-hand side and its counterpart in the context C of the right-hand side.

It is obvious how in a chain of elementary reductions, the chain of single-step origin relations can be used to find the origins of any subterm in the reduction.

In an alternative, more implementation-oriented view, subterms are annotated with their origins (as sets of paths to the original term). For each reduction, the origins of the redex are propagated to the contractum in accordance with the single-step origin relations.

Origin tracking for conditional TRSs (CTRSs) is an extension of the origin function for unconditional TRSs, but is slightly more complicated. The main complications arise from the fact that we want to be able to determine origins of terms that appear in the (sub)reductions necessary to evaluate conditions.

If evaluation of a condition involves a reduction of a term t, then the origins of the redex are passed to that term t, according to the common-variables and common-subterms rule. These origins are propagated to the normal form of t, by using the normal origin relations. If a condition introduces variables, then these are matched against normal forms that have already obtained an origin. These variables may be re-used in other conditions, or in the right-hand side of the conclusion, thus giving the contractum its origins.

2 Formal Definition

In this section, we present a formal definition of origin tracking. A basic knowledge of term rewriting systems (TRSs), and conditional term rewriting systems (CTRSs) is assumed. For a detailed discussion of these, the reader is referred to [Klo91].

The remainder of this section is organized as follows. First, we introduce basic concepts and rewriting histories for unconditional TRSs. Subsequently, the origin function for unconditional TRSs is defined, and illustrated by way of an example. After discussing basic concepts and rewriting histories for CTRSs, we consider the origin function for CTRSs.

We have used the formal definition of the origin relation to obtain an executable specification of origin tracking. The examples that will be given in this section have been verified automatically using that specification.

2.1 Basic concepts for unconditional TRSs

A notion which will frequently recur throughout this paper is that of a *path* (occurrence), consisting of a sequence of natural numbers between brackets. Paths are used to indicate subterms of a term by interpreting numbers as argument positions of function symbols. For instance, (2 1) indicates subterm `b` of term `f(a, g(b, c))`. This is indicated by the / operator: `f(a, g(b, c))`$/(2\ 1) =$ `b`. The associative operator $\cdot$ concatenates paths, e.g., $(2) \cdot (1) = (2\ 1)$. The operators $\prec$, $\preceq$, and $\mid$ define the prefix ordering on paths. The fact that p is a prefix of q is denoted $p \prec q$; $\preceq$ is the reflexive closure of $\prec$. Two paths p and q are disjoint (denoted by $p \mid q$) if neither one is a prefix of the other.

The set of all valid paths in a term t is $O(t)$. The set of variables occurring in t is denoted $Vars(t)$. We use $t' \subset t$ to express that t' appears as a subterm of t; the reflexive closure of $\subset$ is $\subseteq$. The negations of $\subset$ and $\subseteq$ are $\not\subset$ and $\not\subseteq$, respectively. Finally, $Lhs(r)$ and $Rhs(r)$ indicate the left-hand side and the right-hand side of a rewrite rule r.

2.2 A FORMALIZED NOTION OF A REWRITING HISTORY

A basic assumption in the subsequent definitions is that the complete history of the rewriting process is available. This is by no means essential to our definitions, but has the following advantages:

- The origin function for CTRSs can be defined in a declarative, non-recursive manner. We encountered ill-behaved forms of recursion in the definition itself when we experimented with more operational definition methods.
- Uniformity of the origin functions for unconditional TRSs and for CTRSs. The latter can be defined as an extension of the former.

In the case of unconditional TRSs, the *rewriting history* $\mathcal{H}$ is a single *reduction sequence* $\mathcal{S}$. This sequence consists of a list of *sequence elements* $\mathcal{S}_i$ which contain all information regarding the i^{th} rewrite-step. Here, i ranges from 1 to $|\mathcal{S}|$ where $|\mathcal{S}|$ is the length of sequence $\mathcal{S}$.

Each sequence element is a 5-tuple (n, t, r, p, σ) where n is the name of the sequence element (consisting of a sequence name and a number), t denotes the i^{th} term of sequence $\mathcal{S}$, r the i^{th} rewrite rule applied, p the path to the redex in t, and σ the substitution used in the application of r. Access functions $n(s)$, $t(s)$, $r(s)$, $p(s)$, and $\sigma(s)$ are used to obtain the components of s. The last element of a sequence is irregular, because the term associated with this element is in normal form: the rule, path and substitution associated with $\mathcal{S}_{|\mathcal{S}|}$ consist of the special value *undefined*.

Below, s, s', and s'' denote sequence elements. Moreover, it will be useful to have a notion $\mathcal{H}^-$ denoting the history $\mathcal{H}$ from which the last sequence element, $\mathcal{S}_{|\mathcal{S}|}$, is excluded. For our convenience, we introduce $Lhs(s)$ and $Rhs(s)$ to denote the left-hand side and right-hand side of $r(s)$. Finally, $Succ(\mathcal{H}, s)$ denotes the successor of s, for s in $\mathcal{H}^-$, and $Start(\mathcal{H})$ determines the first element of the reduction sequence in $\mathcal{H}$.

2.3 THE ORIGIN FUNCTION FOR UNCONDITIONAL TRSs

2.3.1 AUXILIARY NOTIONS

The auxiliary function *Com* (Definition 2.1) is frequently used in the definitions of the origin functions below, to compute positions of common variables and common subterms. The arguments of *Com* are a substitution σ and two terms t and t'. The result computed by *Com* is a set containing pairs (p, p') such that either a variable X or a common subterm t'' occurs both at path p in t and at path p' in t'.

DEFINITION 2.1 (*Com*)

$$Com(\sigma, t, t') \equiv \{ (p \cdot q, p' \cdot q) \mid t/p \in Vars(t),\ t/p = t'/p',\ q \in O(\sigma(t/p)) \} \cup \{ (p, p') \mid t/p \notin Vars(t),\ t/p = t'/p' \}$$

For one-step reductions, the basic origin relation LR (short for Left-hand side to Right-hand side) relates common subterms of a redex and its contractum, which appear as a result of the presence of a common variable or a common subterm in the applied rewrite rule.

DEFINITION 2.2 (LR) For s in $\mathcal{H}^-$: $LR(s) \equiv Com(\sigma(s), Lhs(s), Rhs(s))$

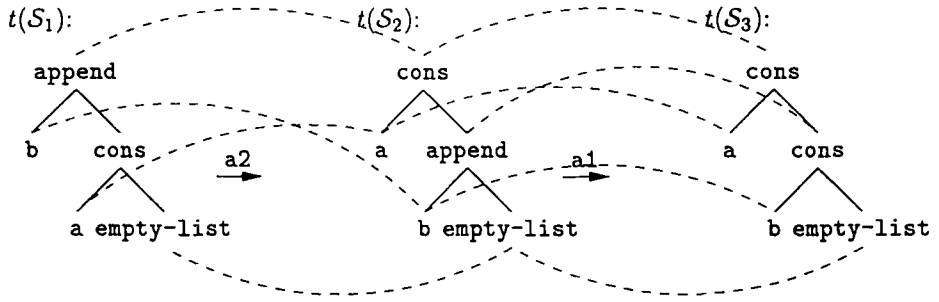


Figure 8: History $\mathcal{H}_{app}$ for `append(b, cons(a, empty-list))`. Dashed lines indicate origin relations.

2.3.2 Definition of *ORG*

The origin function *ORG* for unconditional TRSs is defined using *LR*. Relations are represented by *relate* clauses: a clause $relate(\mathcal{H}, s', p', s, p)$ indicates a relation between the subterm at path p' in $t(s')$ and the subterm at path p in $t(s)$ in history $\mathcal{H}$. In **(u1)**, all relations between symbols in the redex in $t(s)$ and its contractum in $t(Succ(\mathcal{H}, s))$ are defined, excluding the top symbols of the redex and the contractum. The fact that all symbols in the context of the redex remain unchanged is expressed in **(u2)**. In addition, the top symbols of the redex and the contractum are related by **(u2)**.

For s in $\mathcal{H}$ and a path p in $t(s)$, the set of related subterms (according to the transitive and reflexive closure of *relate*) in the initial term, $t(Start(\mathcal{H}))$, is denoted $ORG(\mathcal{H}, s, p)$.

Definition 2.3 (*ORG*) For s in $\mathcal{H}^-$ and s' in $\mathcal{H}$:

$$\begin{array}{lll} \textbf{(u1)} & \forall (q, q') \in LR(s): & relate(\mathcal{H}, s, p(s)\cdot q, Succ(\mathcal{H}, s), p(s)\cdot q') \\ \textbf{(u2)} & \forall p: (p \preceq p(s)) \vee (p \mid p(s)): & relate(\mathcal{H}, s, p, Succ(\mathcal{H}, s), p) \end{array}$$

$$ORG(\mathcal{H}, s, p) \equiv \begin{cases} \{ p \} & \text{when } s = Start(\mathcal{H}) \\ \{ p'' \mid p'' \in ORG(\mathcal{H}, s', p'), relate(\mathcal{H}, s', p', s, p) \} & \text{when } s \neq Start(\mathcal{H}) \end{cases}$$

In principle, the availability of all *relate* clauses allows us to determine relationships between subterms of two arbitrary intermediate terms that occur during the rewriting process. In this paper, we will focus on relations involving the initial term.

2.3.3 Example

As an example, we consider the TRS consisting of the two rewrite rules `[a1]` and `[a2]` of section 1.3. Figure 8 shows a history $\mathcal{H}_{app}$, consisting of a sequence $\mathcal{S}$, as obtained by rewriting the term `append(b, cons(a, empty))`.

Below, we argue how the origin relations shown in Figure 8 are derived from Definition 2.3. For the first sequence element, $\mathcal{S}_1$, we have $p(\mathcal{S}_1) = ()$, $r(\mathcal{S}_1) =$ `a2`, and $\sigma(\mathcal{S}_1) = \{ E1 \mapsto$ `b`, $E2 \mapsto$ `a`, $L \mapsto$ `empty-list` $\}$. As all variable bindings are constants here, we have: $O(E1^{\sigma(\mathcal{S}_1)}) = O(E2^{\sigma(\mathcal{S}_1)}) = O(L^{\sigma(\mathcal{S}_1)}) = \{ () \}$. From this, we obtain:

$$LR(\mathcal{S}_1) = Com(\sigma(\mathcal{S}_1), Lhs(\mathcal{S}_1), Rhs(\mathcal{S}_1)) = \{ ((1), (2\,1)), ((2\,1), (1)), ((2\,2), (2\,2)) \}$$

In a similar way, we compute:

$$LR(\mathcal{S}_2) = Com(\sigma(\mathcal{S}_2),\ Lhs(\mathcal{S}_2),\ Rhs(\mathcal{S}_2)) = \{\ ((1),\ (1)),\ ((2),\ (2))\ \}$$

From Definition 2.3 we now derive the following *relate* relationships. (Note that the last three relationships are generated according to **(u1)** of Definition 2.3.)

$$\begin{array}{ll}
relate(\mathcal{H}_{app},\ \mathcal{S}_1,\ (1),\ \mathcal{S}_2,\ (2\ 1)) & relate(\mathcal{H}_{app},\ \mathcal{S}_1,\ (2\ 1),\ \mathcal{S}_2,\ (1)) \\
relate(\mathcal{H}_{app},\ \mathcal{S}_1,\ (2\ 2),\ \mathcal{S}_2,\ (2\ 2)) & relate(\mathcal{H}_{app},\ \mathcal{S}_1,\ (),\ \mathcal{S}_2,\ ()) \\
relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (2\ 1),\ \mathcal{S}_3,\ (2\ 1)) & relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (2\ 2),\ \mathcal{S}_3,\ (2\ 2)) \\
relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (),\ \mathcal{S}_3,\ ()) & relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (1),\ \mathcal{S}_3,\ (1)) \\
relate(\mathcal{H}_{app},\ \mathcal{S}_2,\ (2),\ \mathcal{S}_3,\ (2)) &
\end{array}$$

As an example, we compute the subterms related to the constant a at path (1) in $t(\mathcal{S}_3)$:

$$\begin{array}{c}
ORG(\mathcal{H}_{app},\ \mathcal{S}_3,\ (1)) = \{\ p'' \mid p'' \in ORG(\mathcal{H}_{app},\ s',\ p'),\ relate(\mathcal{H}_{app},\ s',\ p',\ \mathcal{S}_3, (1))\ \} = \\
\{\ p'' \mid p'' \in ORG(\mathcal{H}_{app},\ \mathcal{S}_2,\ (1))\ \} = ORG(\mathcal{H}_{app},\ \mathcal{S}_2,\ (1)) = \\
\{\ p'' \mid p'' \in ORG(\mathcal{H}_{app},\ s',\ p'),\ relate(\mathcal{H}_{app},\ s',\ p',\ \mathcal{S}_2, (1))\ \} = \\
\{\ p'' \mid p'' \in ORG(\mathcal{H}_{app},\ \mathcal{S}_1,\ (2\ 1))\ \} = ORG(\mathcal{H}_{app},\ \mathcal{S}_1,\ (2\ 1)) = \{\ (2\ 1)\ \}
\end{array}$$

Hence, the constant a at path (1) in $t(\mathcal{S}_3)$ is related to the constant a at path (1 2) in the initial term.

We conclude this example with a few brief remarks. First, some symbols in $t(\mathcal{S}_3)$ are not related to any symbol of $t(\mathcal{S}_1)$. For instance, symbol cons at path (2) in $t(\mathcal{S}_3)$ is only related to symbol append in $t(\mathcal{S}_2)$; this symbol, in turn, is not related to any symbol in $t(\mathcal{S}_1)$. Second, we have chosen a trivial example where no origins occur that contain more than one path. Such a situation may arise when a rewrite rule is not left-linear, or when the right-hand side of a rewrite rule consists of a common variable or a common subterm.

2.4 Basic concepts for CTRSs

A conditional rewrite-rule takes the form:

$$lhs \rightarrow rhs \quad \text{when} \quad l_1 = r_1,\ \cdots,\ l_n = r_n$$

We assume that CTRSs are executed as join systems [Klo91]: both sides of a condition are instantiated and normalized. A condition succeeds if the resulting normal forms are syntactically equal. It is assumed that the conditions of a rule are evaluated in left-to-right order. As an extension, we allow *one* side of a condition to introduce variables; we will refer to such variables as *new* variables (as opposed to *old* variables which were bound during the matching of the left-hand side, or during the evaluation of a previous condition). To avoid complications in our definitions, we impose the non-essential restriction that no condition side may contain old as well as new variables. New variables may occur in subsequent conditions as well as in the right-hand side. Variable-introducing condition sides are *not* normalized, but matched against the normal form of the non-variable-introducing side (for details, see [Wal91]).

Let $|r|$ be the number of conditions of r. For $1 \leq j \leq |r|$, the left-hand side and the right-hand side of the j^{th} condition of r are denoted $Side(r,\ j,\ left)$ and $Side(r,\ j,\ right)$,

respectively. Moreover, let $\overline{left} = right$ and $\overline{right} = left$. The function *VarIntro* (Definition 2.4) indicates where new variables occur; tuples $(h, side)$ are computed, indicating that $Side(r, h, side)$ is variable-introducing.

DEFINITION 2.4 (*VarIntro*)

$$VarIntro(r) \equiv \{ (h, side) \mid X \not\subseteq Lhs(r),\ X \subseteq Side(r, h, side),\ \forall j\,(j < h)\,\forall side' : X \not\subseteq Side(r, j, side') \}$$

For convenience, we also define a function *NonVarIntro* (Definition 2.5) which computes tuples $(h, side)$ for all non-variable-introducing condition sides.

DEFINITION 2.5 (*NonVarIntro*)

$$NonVarIntro(r) \equiv \{ (h, side) \mid 1 \leq h \leq |r|,\ side \in \{ left, right \},\ (h, side) \notin VarIntro(r) \}$$

2.5 REWRITING HISTORIES FOR CTRSs

Conditional term rewriting can be regarded as the following cyclic 3-phase process:

1. Find a match between a subterm t and the left-hand side of a rule r.
2. Evaluate the conditions of r: instantiate and normalize non-variable-introducing condition sides.
3. If all conditions of r succeed: replace t by the instantiated right-hand side of r.

In phase 2, each normalization of an instantiated condition side is a situation similar to the normalization of the original term, involving the same 3-phase process. Thus, we can model the rewriting of a term as a tree of reduction sequences. The *initial reduction sequence* named $\mathcal{S}^{init}$ starts with the initial term and contains sequence elements $\mathcal{S}_i^{init}$ that describe successive transformations of the initial term. In addition, $\mathcal{H}$ now contains a sequence for every condition side that is normalized in the course of the rewriting process. Two sequences appear for non-variable-introducing conditions, but for variable-introducing conditions only one sequence occurs in $\mathcal{H}$ (for the non-variable-introducing side).

Formally, we define the history as a flat representation of this tree of reduction sequences. A history now consists of two parts:

- A set of uniquely named reduction sequences. Besides the initial sequence, $\mathcal{S}^{init}$, there is a sequence $\mathcal{S}^k$ (with k an integer) for every condition side that is normalized in the course of the rewriting process.

  As before, a sequence consists of one or more sequence elements, and each sequence element is a 5-tuple (n, t, r, p, σ), denoting the name, term, rule, path, and substitution involved. As in the unconditional case, access functions are provided to obtain the components of s. A name of a sequence element is composed of a sequence name and a number, permitting us to find out to what sequence an element belongs.

- A mechanism indicating the connections between the various reduction sequences. This mechanism takes the form of a relation which determines a sequence name given a name of a sequence element s, a condition number j, and a condition side *side*, for all $(j, side) \in NonVarIntro(s)$. E.g., a tuple $(n(s), j, side, sn)$ indicates that a sequence named sn occurred as a result of the normalization of $Side(s, j, side)$.

Two functions *First* and *Last* are defined, both taking four arguments: the history $\mathcal{H}$, a sequence element s, a condition number j, and a condition side *side*. $First(\mathcal{H}, s, j, side)$ retrieves the name of s, determines the name of the sequence associated with side *side* of condition j of $r(s)$, looks up this sequence in $\mathcal{H}$, and returns the first element of this sequence. $Last(\mathcal{H}, s, j, side)$ is similar: it determines the last element of the sequence associated with side *side* of condition j of $r(s)$.

Furthermore, $\mathcal{H}^-$ now denotes the history $\mathcal{H}$ from which all last elements of sequences are excluded, $Succ(\mathcal{H}, s)$ now denotes the successor of s in the same sequence, for s in $\mathcal{H}^-$, and $Start(\mathcal{H})$ determines the first element of the initial sequence in $\mathcal{H}$. Finally, we introduce the shorthands $Side(s, j, side)$, $VarIntro(s)$, and $NonVarIntro(s)$ for $Side(r(s), j, side)$, $VarIntro(r(s))$, and $NonVarIntro(r(s))$, respectively.

2.6 The origin function for CTRSs

2.6.1 Basic origin relations

The basic origin relation LR (Definition 2.2) defines relations between consecutive elements s and $Succ(\mathcal{H}, s)$ of the same sequence. The basic origin relations LC, CR, and CC define relations between elements of different sequences. Each of these relations reflects the following principle: common subterms are only related when a common variable or a common subterm appears at corresponding places in the left-hand side, right-hand side and condition side of the rewrite rule involved.

Definition 2.6, LC (Left-hand side to Condition side), defines relations which result from common variables and common subterms of the left-hand side and a condition side of a rule. An LC-relation connects a sequence element s to the first element s' of a sequence for the normalization of a condition side of $r(s)$. The relation consists of triples (q, q', s') indicating a relation between the subterm at path q in the redex and the subterm at path q' in $t(s')$.

We do not establish LC-relations for variable-introducing condition sides, because such relations are always *redundant*. To understand this, consider the fact that we disallow instantiated variables in variable-introducing condition sides. Thus, LC relations would always correspond to a common *subterm* t of the left-hand side and a variable-introducing condition side. Then, only if t also occurs in a subsequent condition side, or in the right-hand side of the rule can the relation be relevant for the remainder of the rewriting history. But if this is the case, this other occurrence of t will be involved in an LC-relation anyway.

Definition 2.6 (LC) For s in $\mathcal{H}^-$:

$$LC(\mathcal{H}, s) \equiv \{ (q, q', s') \mid s' = First(\mathcal{H}, s, j, side), (j, side) \in NonVarIntro(s), (q, q') \in Com(\sigma(s), Lhs(s), Side(s, j, side)) \}$$

In Definition 2.7 and Definition 2.8 below, the final two basic origin relations, CR (Condition side to Right-hand side) and CC (Condition side to Condition side) are presented. These relations are concerned with common variables and common subterms in variable-introducing condition sides. In addition to a variable-introducing condition side, these relations involve the right-hand side, and a non-variable-introducing condition side, respectively. The following technical issues arise here:

- There are no CR and CC relations for non-variable-introducing conditions, because both condition sides are normalized in this case, and no obvious correspondence with the syntactical form of the rewrite rule remains.
- As mentioned earlier, no reduction sequence appears in $\mathcal{H}$ for a variable-introducing condition side. To deal with this issue, the variable-introducing side $Side(s, j, side)$ is used to indicate relations with the term $t(Last(\mathcal{H}, s, j, \overline{side}))$ it is matched against.

CR-relations are triples (q, q', s') indicating that the subterm at path q in $t(s')$ is related to the subterm at path q' in the contractum; CC-relations are quadruples (q, q', s', s'') which express a relation between the subterm at path q in $t(s')$ and the subterm at path q' in $t(s'')$.

DEFINITION 2.7 (CR) For s in $\mathcal{H}^{-}$:

$$CR(\mathcal{H}, s) \equiv \{(q, q', s') \mid (j, side) \in VarIntro(s),\ s' = Last(\mathcal{H}, s, j, \overline{side}),\ (q, q') \in Com(\sigma(s), Side(s, j, side), Rhs(s))\}$$

DEFINITION 2.8 (CC) For s in $\mathcal{H}^{-}$:

$$CC(\mathcal{H}, s) \equiv \{(q, q', s', s'') \mid (j, side) \in VarIntro(s),\ (h, side') \in NonVarIntro(s),\ j < h,\ s' = Last(\mathcal{H}, s, j, \overline{side}),\ s'' = First(\mathcal{H}, s, h, side'),\ (q, q') \in Com(\sigma(s), Side(s, j, side), Side(s, h, side'))\}$$

2.6.2 DEFINITION OF $CORG$

The origin function $CORG$ for CTRSs (Definition 2.9) is basically an extension of ORG. Using the basic origin relations LC, CR, and CC, relations between elements of different reduction sequences are established in **(c1)**, **(c2)**, and **(c3)**. Again, the origin function computes a set of paths in the initial term according to the transitive and reflexive closure of *relate*. For any sequence element s in $\mathcal{H}$, and any path p in $t(s)$, $CORG$ computes a set of paths to related subterms in $t(Start(\mathcal{H}))$.

DEFINITION 2.9 ($CORG$) For s in $\mathcal{H}^{-}$ and s' in $\mathcal{H}$:

(u1)	$\forall (q, q') \in LR(s)$:	$relate(\mathcal{H}, s, p(s) \cdot q, Succ(\mathcal{H}, s), p(s) \cdot q')$
(u2)	$\forall p : (p \preceq p(s)) \vee (p \mid p(s))$:	$relate(\mathcal{H}, s, p, Succ(\mathcal{H}, s), p)$
(c1)	$\forall (q, q', s') \in LC(\mathcal{H}, s)$:	$relate(\mathcal{H}, s, p(s) \cdot q, s', q')$
(c2)	$\forall (q, q', s') \in CR(\mathcal{H}, s)$:	$relate(\mathcal{H}, s', q, Succ(\mathcal{H}, s), p(s) \cdot q')$
(c3)	$\forall (q, q', s', s'') \in CC(\mathcal{H}, s)$:	$relate(\mathcal{H}, s', q, s'', q')$

$$CORG(\mathcal{H}, s, p) \equiv \begin{cases} \{p\} & \text{when } s' = Start(\mathcal{H}) \\ \{p'' \mid p'' \in CORG(\mathcal{H}, s', p'),\ relate(\mathcal{H}, s', p', s, p)\} & \text{when } s' \neq Start(\mathcal{H}) \end{cases}$$

2.6.3 EXAMPLE

We extend the example of section 2.3.3 with the following conditional rewrite rules for a function `rev` to reverse lists.

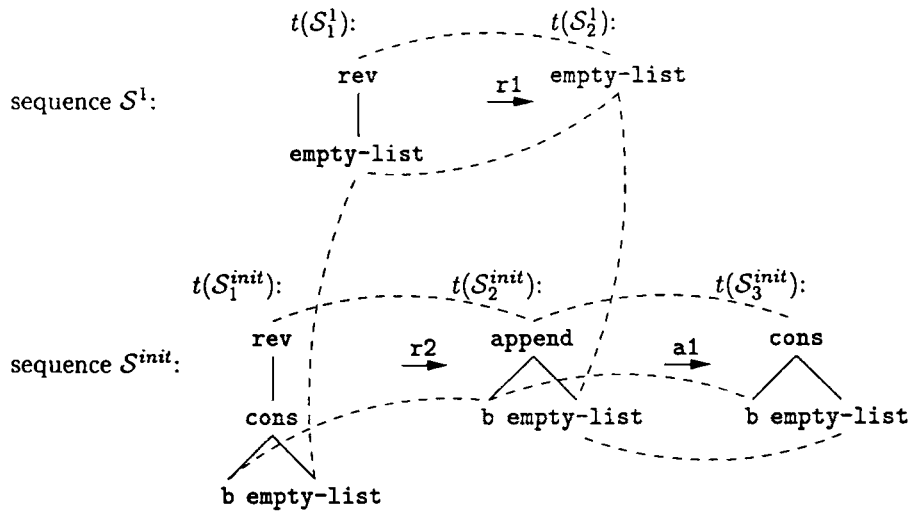


Figure 9: History $\mathcal{H}_{rev}$ for `rev(cons(b, empty-list))`. Dashed lines indicate origin relations.

```
[r1] rev(empty-list) -> empty-list
[r2] rev(cons(E, L1)) -> append(E, L2) when L2 = rev(L1)
```

In rule `r2`, a variable *L2* is introduced in the left-hand side of the condition. Actually, the use of a new variable is not necessary in this case: we may alternatively write `append(E, rev(L1))` for the right-hand side of `r2`. The new variable is used solely for the sake of illustration. Figure 9 shows the rewriting history $\mathcal{H}_{rev}$ for the term `rev(cons(b, empty-list))`. Note that besides the initial sequence, S^{init}, only one sequence, S^1, appears for the normalization of the condition of `r2`, because it is variable-introducing.

For sequence element S_1^{init} we have $p(S_1^{init}) = ()$, $\sigma(S_1^{init}) = \{ E \mapsto \texttt{b},\ L1 \mapsto \texttt{empty-list},\ L2 \mapsto \texttt{empty-list} \}$. It follows that $O(E^{\sigma(S_1^{init})}) = O(L1^{\sigma(S_1^{init})}) = O(L2^{\sigma(S_1^{init})}) = \{ () \}$. Moreover, $VarIntro(S_1^{init}) = \{ (1, left) \}$. Consequently, we obtain:

$$LR(S_1^{init}) = \{ ((1\,1), (1)) \}, \qquad LC(\mathcal{H}_{rev}, S_1^{init}) = \{ ((1\,2), (1), S_1^1) \}$$
$$CR(\mathcal{H}_{rev}, S_1^{init}) = \{ ((), (1), S_2^1) \}, \quad CC(\mathcal{H}_{rev}, S_1^{init}) = \emptyset$$

As a result, the following relationships are generated for S_1^{init}:

$$relate(\mathcal{H}_{rev}, S_1^{init}, (), S_2^{init}, ()) \qquad relate(\mathcal{H}_{rev}, S_1^{init}, (1\,1), S_2^{init}, (1))$$
$$relate(\mathcal{H}_{rev}, S_1^{init}, (1\,2), S_1^1, (1)) \qquad relate(\mathcal{H}_{rev}, S_2^1, (), S_2^{init}, (2))$$

In a similar way, the following *relate* relationships are computed for S_2^{init} and S_1^1:

$$relate(\mathcal{H}_{rev}, S_2^{init}, (), S_3^{init}, ()) \qquad relate(\mathcal{H}_{rev}, S_2^{init}, (1), S_3^{init}, (1))$$
$$relate(\mathcal{H}_{rev}, S_2^{init}, (2), S_3^{init}, (2)) \qquad relate(\mathcal{H}_{rev}, S_1^1, (), S_2^1, ())$$
$$relate(\mathcal{H}_{rev}, S_1^1, (1), S_2^1, ())$$

Finally, we compute the subterms related to `empty-list` at path (2) in $t(S_3^{init})$:

$$CORG(\mathcal{H}_{rev}, S_3^{init}, (2)) = \{ p'' \mid p'' \in CORG(\mathcal{H}_{rev}, s', p'), relate(\mathcal{H}_{rev}, s', p', S_3^{init}, (2)) \} =$$
$$\{ p'' \mid p'' \in CORG(\mathcal{H}_{rev}, S_2^{init}, (2)) \} = CORG(\mathcal{H}_{rev}, S_2^{init}, (2)) = \cdots = \{ (1\,2) \}$$

Consequently, the constant `empty-list` in $t(\mathcal{S}_3^{init})$ is related to the constant `empty-list` in $t(\mathcal{S}_1^{init})$.

3 Properties

The origins defined by *CORG* have the following property: whenever an intermediate term t_{mid} has a path to initial subterm t_{org} in its origin, then t_{org} can be rewritten to t_{mid} in zero or more reduction steps. This property gives a good intuition of the origin relations which are established in applications such as error handling or debugging.

To see why this property holds, we first consider one reduction step:

PROPERTY 3.1 Let $\mathcal{H}$ be a rewriting history, s, s' arbitrary sequence elements in $\mathcal{H}$, and p, p' paths. For any $relate(\mathcal{H}, s, p, s', p')$ we have $t(s) \equiv t(s')$ or $t(s) \rightarrow t(s')$.

Informally stated, directly *related* terms are either syntactically equal or one can be reduced to the other in exactly one step. This holds because the context, common-variables, and common-subterm relations all relate identical terms. Only the redex-contractum relation links non-identical terms, but these can be rewritten in one step. Because the origin relation *CORG* is defined as the transitive and reflexive closure of *relate*, we now have the desired property:

PROPERTY 3.2 Let $\mathcal{H}$ be a history. For every term $t(s)$ occurring in some sequence element s in history $\mathcal{H}$, and for every path $p \in O(t(s))$, we have:

$$q \in CORG(\mathcal{H}, s, p) \Rightarrow t(Start(\mathcal{H}))/q \rightarrow^* t(s)/p$$

When using origins, one may be interested in the *number of paths* in an origin. To that end, we introduce:

DEFINITION 3.1 Let o be an origin, and let $|o|$ denote the number of paths in o. Then: o is *empty* iff $|o| = 0$, *non-empty* iff $|o| \geq 1$, *precise* iff $|o| \leq 1$, and *unitary* iff $|o| = 1$.

For some applications, unitary origins are desirable. In animators for sequential program execution, one wants origins which refer to exactly one statement. On the other hand, when error-positioning is the application, it can be desirable to have non-unitary origins, as for instance in errors dealing with multiple declarations of the same variable (see, e.g., the label declaration in Figure 2).

The properties below indicate how non-empty, precise and unitary origins can be detected through static analysis of the CTRS. In the sequel r denotes an arbitrary rule, j is a number of some condition in r, and $side \in \{left, right\}$ denotes an arbitrary side.

PROPERTY 3.3 (Non-empty origins) Terms with top symbol f have non-empty origins if for all open terms u with top function symbol f:

$$\begin{array}{ll} (1) & u \subseteq Rhs(r) \Rightarrow u \subseteq Lhs(r) \\ (2) & u \subseteq Side(r,\ j,\ side) \Rightarrow u \subseteq Lhs(r) \end{array}$$

This can be proved by induction over all *relate* clauses, after introducing an ordering on all sequence elements. Informally, all terms with top symbol f will have non-empty origins

if no f is introduced that is not *related* to a "previous" f. Note that relations according to variables have no effect on origins being (non-)empty.

To characterize sufficient conditions for precise and unitary origins, we first need some definitions:

DEFINITION 3.2 Let r be a conditional rewrite rule and u an open term. Then r is an *u-collapse* rule if $Rhs(r) \equiv u$, and $u \subseteq Lhs(r)$.

DEFINITION 3.3 For open terms t and u, t is *linear in* u if u occurs at most once in t.

DEFINITION 3.4 The predicate $LinearIntro(u, r)$ holds if u has at most one occurrence in either the left-hand side or any variable-introducing condition side. Formally: $LinearIntro(u, r) \Leftrightarrow$ there is (1) at most one $t \in \{Lhs(r)\} \cup \{Side(r, h, side) \mid (h, side) \in VarIntro(r)\}$ such that $u \subseteq t$, and (2) this t, if it exists, is linear in u.

PROPERTY 3.4 (Precise origins) Terms with top symbol f have precise origins if the following holds for all open terms u having either f as top symbol or solely consisting of a variable:

(1) The CTRS does not contain u-collapse rules
(2) $u \subseteq Rhs(r) \Rightarrow LinearIntro(u, r)$
(3) $u \subseteq Side(r, j, side) \Rightarrow LinearIntro(u, r)$

Again, this property can be proved by induction over all *relates*. The crux is that no term with top function symbol f is introduced in a way that it is *related* to more than one "previous" term.

PROPERTY 3.5 (Unitary origins) Since "non-empty" and "precise" implies "unitary", combining the premises of Properties 3.3 and 3.4 yields sufficient conditions for unitary origins

For many-sorted CTRSs, some special properties hold. We assume CTRSs to be sort-preserving, i.e., the redex and the contractum belong to the same sort. Hence, *CORG* is sort-preserving. Thus, we have the following property (which in the implementation allows for an optimization—Section 4):

PROPERTY 3.6 *relate* can be partitioned into subrelations for each sort.

One may be interested whether all terms *of some particular sort* S have non-empty, precise, or unitary origins. This happens under circumstances very similar to those formulated for the single-sorted case (Properties 3.3 to 3.5). For precise and unitary origins, however, it is not sufficient to consider only terms of sort S; one also needs to consider sorts T which can have subterms of sort S (since duplication of T-terms may imply duplication of S-terms). Hence, we define:

DEFINITION 3.5 For two sorts S and T, we write $S \sqsubseteq T$ if terms of sort T can contain subterms of sort S.

Using this, we can formulate when terms of sort S have precise or unitary origins. This is similar to the single-sorted case (see Property 3.4), but in (1) u must be of sort S, and (2) and (3) must hold for all u of sort T such that $S \sqsubseteq T$. Unitary origins of sort S are obtained by combining the premises for the non-empty origins and precise origins.

We refer to [DKT92] for more elaborate discussions of the above results.

4 Implementation Aspects

An efficient implementation of origin tracking has been completed, and is currently being integrated in the ASF+SDF system. In this section, we briefly address the principal aspects of implementing origin tracking.

4.1 The basic algorithm

In our implementation, each symbol is *annotated* with its origin, during rewriting. Two issues had to be resolved:

- annotation of the initial term
- propagation of origins during rewriting

The first issue is a trivial matter because—by definition—the origin of the symbol at path p is $\{p\}$. The second issue is addressed by copying origins from the redex to the contractum according to the basic origin relation LR. In a similar way, propagations occur for the basic origin relations LC, CC, and CR. Observe that no propagations are necessary for the origins in the context of the redex, as the origins of these symbols remain unaltered.

4.2 Optimizations

Several optimizations of the basic algorithm have been implemented:

- All positional information (i.e., the positions of common variables and common subterms) is computed in advance, and stored as annotations of rewrite rules.
- The rewriting engine of the ASF+SDF system explicitly constructs a list of variable bindings. Origin propagations which are the result of common variables can be implemented as propagations to these bindings. When a right-hand side or condition side is instantiated, all common variable propagations are handled as a result of the instantiation. The advantage of this approach is that the number of propagations decreases, because we always propagate to only one subterm for each variable.
- Origins are implemented as a set of pointers to function symbols of the initial term. The advantages are twofold: less space is needed to represent origins, and set union becomes a much cheaper operation.

4.3 Associative lists

In order to implement origin tracking in the ASF+SDF system, provisions had to be made for *associative lists* [Hen91, Wal91]. Associative lists can be regarded as functions with a variable arity. Allowing list functions in CTRSs introduces two minor complications:

- A variable that matches a sublist causes relations between arrays of adjacent subterms. In the implementation, we distinguish between ordinary variables and list variables, and perform propagations accordingly.

- Argument positions below list functions depend on the actual bindings. Therefore, when computing the positions of common variables and common subterms, positions below lists are marked as *relative*. The corresponding *absolute* positions are determined during rewriting.

Consider the following example, where `l` is a list function, and X^* is a list variable which matches sublists of any length:

```
[r1] f(l(X*, a)) → g(l(X*, a))
```

When we rewrite the redex `f(l(b, c, a))` according to `r1`, the contractum is `g(l(b, c, a))`. Variable X^* gives rise to both a relation between the constants `b` in the redex and the contractum, and a relation between the constants `c` in the redex and the contractum. Moreover, constant `a` appears at path (1 2) in the left-hand side of `r1`, but at path (1 3) in the redex.

4.4 Sharing of subterms

For reasons of efficiency, implementations of CTRSs allow sharing of subtrees, thus giving rise to DAGs (Directed Acyclic Graphs) instead of trees. The initial term is represented as a tree, and sharing is introduced by instantiating non-linear right-hand sides and condition sides. For every variable, the list of bindings contains a *pointer* to one of the subterms it was matched against. Instantiating a right-hand side or condition side is done by copying these pointers (instead of copying the terms in the list of bindings). Sharing has the following repercussions for origin tracking:

- No propagations are needed for variables which occur exactly once in the left-hand side (and for new variables which occur exactly once in the introducing condition). This results in a radical reduction of the number of propagations.
- Variables which occur non-linearly in the left-hand side of a rule (and new variables which occur non-linearly in the introducing condition) present a problem. When sharing is allowed in this case, inconsistent origins with respect to the definition may arise. A solution to this problem consists of using a pointer to a *copy* of the term matched against such a variable in the list of bindings. This corresponds to disallowing sharing in a limited number of situations.

4.5 Restricting origin tracking to selected sorts

Often, one is only interested in the origins of subterms of a particular sort. A straightforward result of Property 3.6 is the following: to compute the origins of subterms of sort `S`, only propagations for common subterms of sort `S`, and for common variables of sorts `T` such that $\mathtt{S} \sqsubseteq \mathtt{T}$ are necessary.

4.6 Time and space overhead of origin tracking

Origins are represented by sets of pointers to symbols of the initial term, and associated with every symbol is exactly one such set. The size of these sets is bounded by the number of function symbols in the initial term because, in the worst case, a set contains a pointer to every symbol in the initial term. Thus, the space overhead of origin tracking is linear in the size of the initial term. In practice, only small sets arise, resulting in little space overhead. The use of efficient set representations would reduce this overhead even further.

We have measured the time overhead caused by origin tracking. In all measurements, the run-time overhead lies between 10% and 100%, excluding the costs of pre-computing positional information.

5 Related Work

In TRS theory, the notion of *descendant* [Klo91] (or *residual* [O'D77, HL79]) is used to study properties like confluence or termination, and to find optimal orders for contracting redexes (see [Mar92] for some recent results). For a reduction $t \to t'$ contracting a redex $s \subseteq t$, a different redex $s' \subseteq t$ may reappear in the resulting term t'. The occurrences of this s' in t' are called the descendants of s'.

Descendants are similar to origins, but more restricted. Only relations according to contexts and common variables are established (Bergstra and Klop [BK86] also use *quasi-descendants* linking the redex and contractum as well). Moreover, descendants are defined for a smaller class of TRSs; only orthogonal (left-linear and non-overlapping) TRSs without conditional equations are allowed.

Bertot [Ber91c, Ber91b] studies residuals in TRSs and λ-calculus, and introduces *marking functions* to represent the residual relation. He provides a formal language to describe computations on these marking functions, and shows how the marking functions can be integrated in formalisms for the specification of programming language semantics (*viz.* term rewriting systems and collections of inference rules). Bertot works in the realm of left-linear, unconditional TRSs and only considers precise origins.

The ideas of Bertot concerning origins in inference rules have been used in the framework of Typol [Des88], a formalism to specify programming languages, based on natural semantics [Kah87]. For compositional definitions of evaluators or type checkers (in which the meaning of a language construct is expressed in terms of its substructures), the implementation of Typol keeps track of the construct currently processed (the *subject*). A pointer to the subject is available in tools derived from the specification, particularly debuggers or error handlers. In addition to automatic subject tracking, Typol has been equipped with special language constructs to manipulate origins explicitly. This contrasts with our approach, where origin tracking is invisible at the specification level.

Berry [Ber91a] aims at deriving animators from relational rules (similar to operational semantics). He defines a *focus* which is either equal to the *subject* (as in Typol) or to the *result* of the evaluation of some subexpression. The theory he develops uses the concept of an *inference tree*, a notion similar to our rewriting histories.

In the context of the PSG system [BMS87], a generator for language-specific debuggers was described. Debuggers are generated from a specification of the denotational semantics of a language and some additional debugging functions. Bahlke et al. insist that programs

are explicitly annotated with their position in the initial syntactic structure before running their semantic tool.

6 Concluding Remarks

6.1 Achievements

Summarizing the results described in this paper, we have:

- A definition of origins that does not depend on a particular rewrite strategy, nor on the confluence or strong-normalization of the underlying CTRS. It establishes only relations which can be derived from the syntactic structure of the rewrite rules.
- The property that whenever a term t_{mid} has a subterm t_{org} in the initial term as origin, this term t_{org} can be rewritten to t_{mid}.
- Sufficient criteria that a specification should satisfy to guarantee that an origin consisting of at least one, or exactly one path is associated with each subterm of a given sort.
- An efficient implementation method for origin tracking.
- A notion of sort-dependent "filtering" of origins, when only the origins of terms of certain sorts are needed.
- A prospect of applying origin tracking to the generation of interactive language-based environments from formal language definitions. In particular, generic techniques for debugging and error reporting have been discussed.

6.2 Limitations

The current method for origin tracking has limitations, most of which are related to the introduction of new function symbols. Some typical problem cases are:

- In the context of translating arithmetic expressions to a sequence of stack machine instructions, one may encounter an equation of the form

  `trans(plus(`E_1`,`E_2`)) -> seq(trans(`E_1`), seq(trans(`E_2`), add))`

  The `plus` of the expression language is translated to the `add` stack-instruction. It seems intuitive to relate both `seq` function symbols to the `plus` symbol at the left-hand side. However, the current origin mechanism do not establish this relation.

- In specifications of evaluators it frequently occurs that the evaluation of one construct is defined by reducing it to another construct, like in

  `eval(repeat(`S`, `Exp`), `Env`) -> eval(seq(`S`, while(not(`Exp`), `S`)), `Env`)`

where the evaluation of the `repeat`-statement is defined in terms of the `while`-statement. In this example, `seq` is a constructor for statement sequences. Here again, the `while`-statement on the right-hand side does not get an origin but the `repeat`-statement on the left-hand side would be a good candidate for this.

These examples have the flavor of translating terms from one representation to another and they illustrate that *more* origin relations between both sides of the equations have to be established. But which? We have already started exploring several extensions. These include linking equal function symbols, distinguishing constructor functions and defined functions, using the additional structure of primitive recursive schemes (as in [Meu90]) to detect origins of interest, looking at father or child nodes to obtain additional origins, allowing user-provided annotations in specifications describing additional origin information, detection of the minimal set of symbols in the initial term to obtain a particular function symbol, and so on. Some of these options are more promising than others; more research is needed to find out which solution meets our needs best.

6.3 Future Work

Directions in which the current results can be extended or applied are:

Generic debugging techniques and animation. Although we have promising experience with the use of origin tracking for animation [DT92], more work is needed to generate language-specific debuggers that are smoothly integrated in the ASF+SDF Meta-environment. We intend to develop generic methods to specify the desired behavior and features of animators and debuggers.

Error reporting. Further work is needed to determine whether and how our current notion of origins has to be extended for the benefit of error reports containing precise error locations. In addition to origin tracking, a way is needed to compose an error *message*. A first result in this direction is discussed in [DT92].

Translation, Code Optimization. We intend to study if an extended notion of origins can be used for the automatic construction of bi-directional mappings between source programs and generated code. This is useful for code-level debugging, where links are required between assembly instructions and statements in the source program.

This is of particular importance when debugging highly optimized code. We expect that the origin information attached to individual assembly instructions will largely survive the complex reorderings involved in the optimization.

Program slicing. A recently introduced notion in the area of debugging and testing is that of a *dynamic program slice* [AH90, KSF92]. A dynamic program slice is that part of the program that actually determines the value of a variable at a particular moment during execution. Origins describe a similar notion. We will examine whether it is possible to achieve dynamic slicing by means of an (extended) version of origin tracking.

Incremental computation. Continuing in the same spirit, origins resemble the information needed to reduce the amount of recomputation necessary for incremental

computations on programs like type checking and translation [Meu90, Fie93]. We will investigate the commonalities and differences between these two forms of information.

ACKNOWLEDGEMENTS

We thank Y. Bertot, J. Field, J. Heering, and J.W. Klop for their comments on drafts of this paper.

References

[AH90] H. Agrawal and J.R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 246–256, 1990. Appeared as *SIGPLAN Notices* 25(6).

[Ber91a] D. Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, University of Edinburgh, 1991.

[Ber91b] Y. Bertot. Occurrences in debugger specifications. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 327–337, 1991. Appeared as *SIGPLAN Notices* 26(6).

[Ber91c] Y. Bertot. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle.* PhD thesis, INRIA, Sophia-Antipolis, 1991. In French.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BK86] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.

[BMS87] R. Bahlke, B. Moritz, and G. Snelting. A generator of language-specific debugging systems. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 92–101, 1987. Appeared as SIGPLAN Notices 22(7).

[Des88] T. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, 1988.

[DKT92] A. van Deursen, P. Klint, and F. Tip. Origin tracking. Report CS-R9230, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[DT92] T.B. Dinesh and F. Tip. Animators and error reporters for generated programming environments. Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[Fie93] J. Field. A graph reduction approach to incremental term rewriting. Technical report, IBM T.J. Watson Center, 1993. To appear.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[HHKR89] 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.

[HL79] G. Huet and J.-J. Lévy. Call by need computations in non-ambiguous linear term rewriting systems. Rapports de Recherche 359, INRIA, 1979. To appear as: Computations in Orthogonal Rewriting Systems, Part I and II, in J.L. Lassez and G. Plotkin, editors, *Computational Logic, essays in honour of Alan Robinson*, MIT Press, 1991.

[Kah87] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 1993. To appear. Preliminary version in J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, LNCS 490, 1991.

[Klo91] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol II.* Oxford University Press, 1991. Also CWI report CS-R9073.

[KSF92] M. Kamkar, N. Shahmehri, and P. Fritzson. Interprocedural dynamic slicing and its application to generalized algorithmic debugging. In *Proceedings of the International Conference on Programming Language Implementation and Logic Programming, PLILP '92*, 1992.

[Mar92] L. Maranget. *La strategie paresseuse.* PhD thesis, INRIA Rocquencourt, 1992. In French.

[Meu90] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. Report CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990. Extended abstract to appear in *AMAST'91: Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, Springer-Verlag.

[O'D77] M.J. O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science.* Springer-Verlag, 1977.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.