Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Origin Tracking

A. van Deursen, P. Klint, F. Tip

Computer Science/Department of Software Technology

CS-R9230 1992

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

Abstract

In the framework of conditional, not necessarily orthogonal, term rewriting systems, we introduce the notion of an *origin*. Origins are relations between subterms of intermediate terms which occur during rewriting, and subterms of the initial term. Origin tracking is a method for incrementally computing origins during rewriting. Origins are a generalization of the well-known concept of residuals (also called descendants). We give a formal definition of origins, and present a method for implementing them.

Origin tracking is a highly versatile technique when applied to the prototyping of algebraic specifications of programming languages. For example, origin tracking allows program execution to be visualized in a semi-automatic way, given an algebraic specification of the dynamic semantics of the programming language. Furthermore, various notions of breakpoints for generic debuggers can be defined without difficulty. Finally, given a specification of the static semantics of a programming language, origin tracking enables, once an error (such as type-incompatibility) has been detected, to infer the position of the error in the source program automatically.

1991 Mathematics Subject Classification: 68N15, 68N20, 68Q42, 68Q55, 68Q65
1991 CR Categories: D.2.1, D.2.5, D.3.1, F.3.2, F.4.2
Key Words & Phrases: Term Rewriting Systems, Origin, Residuals, Algebraic Specifications, Programming Language Semantics, Programming Environments, Automatic Programming, Debugging

1 Introduction

Given a reduction in a Term Rewriting System (TRS) [Klo91], we study the problem of relating parts of the intermediate terms occurring in the rewriting process to parts of the initial term. In a (complex) reduction sequence $t_0 \rightarrow t_1 \rightarrow ... \rightarrow t_n$ each elementary reduction step creates a new term t_{i+1} from a previous term t_i. Consecutive terms t_{i+1} and t_i will often look alike, having for instance common subterms. The contexts of the redex and its contractum will of course be the same; moreover, the redex and contractum themselves may be similar, depending on the rewrite rule applied. For instance, if a variable in the rewrite rule appears both in the left and right-hand side of the rule, then its instantiation will be a subterm occurring both in the redex and in the contractum.

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*.

Similarities between consecutive terms can be extended to arbitrary terms in the rewriting process. We are interested in similarities between parts of intermediate terms and parts of the *initial* term. A formalization of these similarities is given in a relation which we call the *origin* relation. Intuitively, it defines which parts of the initial term are responsible for the creation of particular parts of an intermediate term. In other words, it formalizes from which parts of the initial term a particular subterm originated. The process of incrementally computing origins we will call *origin tracking*.

In this document, we give a precise definition of the origin relation (Section 2). It will be used to study properties like soundness, and to classify origins according to the number of relations established for individual terms (Section 3). Besides, it will be used as a basis for an efficient implementation of origin tracking in a term rewriting machine (Section 4).

The reasons for studying origins are of a practical rather than a theoretical nature. We use term rewriting systems to execute algebraic specifications of programming languages. A typical function (like an evaluator, type checker, or translator) in such a specification operates on the abstract syntax tree of a program (which is part of the initial term). Pieces of the program such as identifiers, expressions, or statements, may recur in intermediate terms. These recurrences are formalized by the origin relation, and are of use for:

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

Examples like these will be presented in Section 5. For TRSs in areas different from programming language descriptions, there may be applications of origins as well.

We will see that the notion of origins is an extension of the well-known notion of *residuals* or *descendants* [O'D77, HL79]. Besides being an extension of residuals, origins are also defined for a wider class of TRSs, since we do not restrict ourselves to orthogonal TRSs, but allow ambiguous, not left-linear, conditional TRSs. Residuals are typically used to study concepts like confluence, termination, or reduction strategy, rather than for the practical purposes we have in mind. Bertot [Ber91a, Ber91b] also studies residuals in TRSs and in the λ-calculus with the aim of applying them to debuggers or automatic error handling. How his work and that of others relates to ours is discussed in Section 6.

The idea to study these topics came to mind while working on the ASF+SDF meta-environment [Hen91, Kli91]. The ASF+SDF meta-environment is a programming environment generator, implemented as part of the CENTAUR system [BCD$^+$89]. Given an algebraic specification for a programming language, it generates a programming environment for the language in question. Thus far, the research has resulted in:

1. An algebraic specification formalism, called ASF+SDF. It resulted from the integration of ASF, an acronym for Algebraic Specification Formalism [BHK89], and SDF, a shorthand for Syntax Definition Formalism [HHKR89]. ASF+SDF supports modularization, user-definable syntax, associative lists, and conditional equations.

2. The ASF+SDF meta-environment [Hen91, Kli91]. First, this is a tool generator which takes a specification in ASF+SDF and derives a lexical analyzer, a parser, and a rewrite machine to reduce terms in the TRS derived from the equations of the

specification. Second, it provides fully interactive support when writing, checking, and testing algebraic specifications.

All tools are generated in an incremental fashion: when the input specification is changed the existing tools are updated incrementally rather than regenerated from scratch.

A central theme in the research done in the context of the ASF+SDFmeta-environment, is that the formal specification of a programming language should be used directly for the generation of tools for that language. The origin relation matches this pattern. Without any adaptation of the specification, the implementation can automatically perform origin tracking. This in turn can be used for the generation of tools like debuggers and typecheckers.

1.1 Origins in Language Specifications

We assume the reader is familiar with TRSs, and refer to Section 2 for the basic concepts, and to [Klo91, DJ90] for full details. As our area of application, we are interested in—but not restricted to—TRSs describing programming languages [BHK89]. In such TRSs terms occur which represent abstract syntax trees of programs such as the term `program(decls(decl(n,natural)), stats( assign(n,34)))`. All kinds of functions operating on programs can be defined, where the program is represented by its abstract syntax tree. Typical functions are type checkers, evaluators, optimizers, and so on.

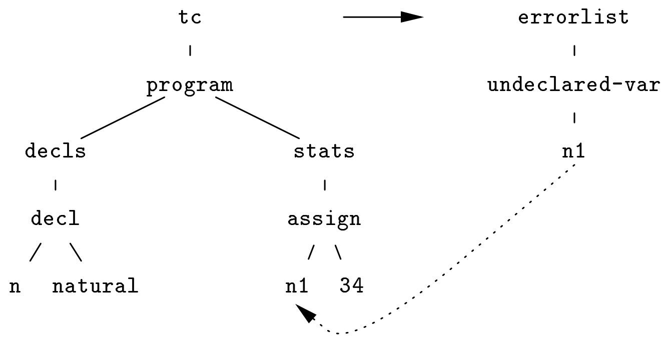

Figure 1: Type checking a simple program

A 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. 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 the function `undeclared-var`, and as argument the name of the variable in question. In Figure 1 we also see a dotted line representing an origin. It relates the occurrence of `n1` in the result to the `n1` in the initial term. Finding a relation in this example seems trivial, but our scheme will also find the correct relation if the program has a block structure in which at some place `n1` is declared and can be used correctly, whereas in another place it can not. The type checker will then again reduce to a term representing an error message, and the origin of its argument `n1` will

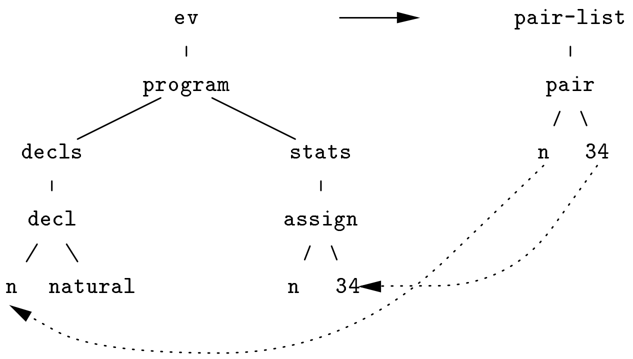

Figure 2: Evaluating a simple program

point to the occurrence where `n1` was used incorrectly. We will use this to indicate precisely the position where errors occurred in source programs, given an algebraic specification of the static semantics of a programming language (see Section 5).

Similarly, program evaluators can be defined. The result of a typical evaluator is a state in which the variables have values. In Figure 2, it is shown how evaluating a simple program results in a list of ⟨identifier, value⟩ pairs. The origins—in this example—relate the identifiers in this environment to their declarations, and the values to the place where they were assigned last.

Origins calculated during evaluation can be used by debugging facilities derived from specifications of the dynamic semantics of a programming language. Notice that the specification of the `ev` function will involve auxiliary functions like `ev-stat` to evaluate a single statement, or `ev-exp` to evaluate an expression (an example of a rule splitting the processing of a list of statements to evaluations of single statements is shown in Figure 3). Thus, evaluation of each statement corresponds to a reduction of a term `ev-stat(Stat,...)`. The origin of `Stat` will precisely indicate the statement that is currently being evaluated. This can be used when defining breakpoints on statements, for animation of execution (for more details, see Section 5).

1.2 Single-step Origin Relations

How can we relate intermediate terms, including the final one, back to the initial term? We will do this by carefully analyzing the syntactical shape of the rewrite rules, thereby indicating which relations should be constructed when applying that particular rewrite rule. We will distinguish four types of single-step origin relations. Once these are available for each rule applied, they can be used to find origins for any subterm by transitively going back to the initial term, according to these relations.

The first single-step relation is based on variable occurrences in the rewrite rule. 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. Consequently, it seems obvious to establish relations according to

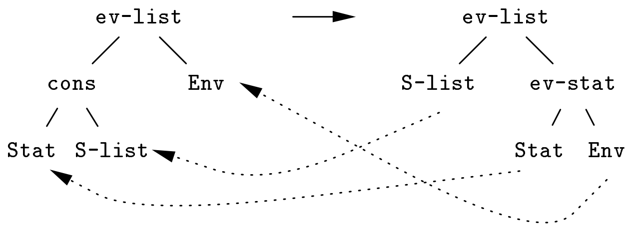

Figure 3: Relations according to variables in a rewrite rule.

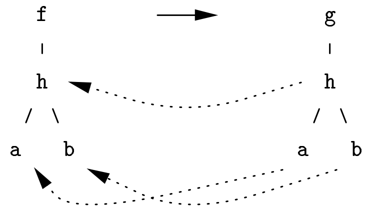

Figure 4: Relations according to variable `X` when applying rule `f(X) -> g(X)`.

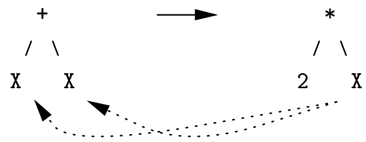

Figure 5: Relations for non left-linear rule `X + X -> 2 * X`.

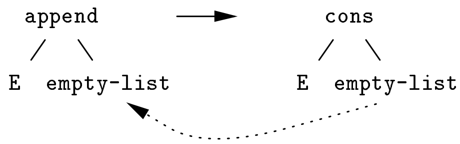

Figure 6: Relations between common subterms.

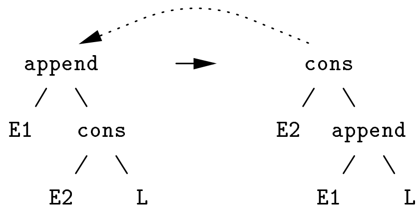

Figure 7: Relations between top symbol of redex and contractum.

variables common to the left-hand side and right-hand side. For the evaluator rule, this is shown in Figure 3. In general, if a rule $r : t_1 \rightarrow t_2$ with `X` occurring in t_1 and t_2 is applied, the symbols in the instantiated X in the right-hand side and those in the left-hand side are related. How all function symbols in the instantiation are related is illustrated in Figure 4.

Rules may be non-linear, i.e., some variable X may have multiple occurrences in its left-hand side like in `X + X -> 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, which is illustrated in Figure 5. As a consequence, non-linearity in the left-hand side causes one subterm to have several related subterms, whereas non-linearity in the right-hand side causes different terms to have the same origin.

The second kind of single-step relation is based on common subterms in the left-hand side and right-hand side. Take for example the rule defining how to append an element to the end of the list:

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

Linking the variables, 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 linked. This is solved by the second single-step relation, relating subterms common to the left-hand side and right-hand side. For equation `[a1]` this is shown in Figure 6.

The third single-step relation is based on the fact that the left-hand side and the right-hand side of any equation clearly denote the same object. We will relate the top symbol of the redex and the top symbol of the contractum, as shown in Figure 7 for equation `[a2]`. Finally, since the context in which a reduction is done remains unaltered, the function symbols in the context at the right-hand side and their corresponding symbols in the left-hand side are related.

1.3 Origin Tracking

To summarize the single-step origin relations, consider Figure 8. Assume a rewrite rule $r : t_1 \rightarrow t_2$ is applied in context C with substitution σ, thus giving rise to the elementary reduction $\mathrm{C}[t_1^\sigma] \rightarrow_r \mathrm{C}[t_2^\sigma]$. Four relations are distinguished:

- 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.
- Common subterms: if a term s is both a subterm of t_1 and of t_2, then these occurrences of s are related.
- Redex-contractum: the top symbol of the redex t_1^σ and the top symbol of its contractum t_2^σ are related
- 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.

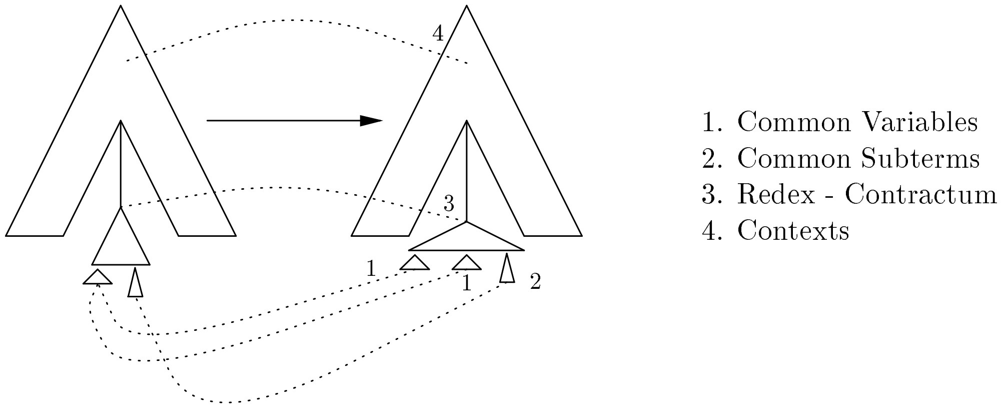

Figure 8: Single-step origin relations in a picture.

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). When a reduction is done, the origins of the redex are propagated to the contractum in accordance with the single-step origin relations. It is clear that this is simply a different view on the same relation.

1.4 Conditional Rules

Although origin tracking for conditional TRSs (CTRSs) is akin to that for the unconditional case, it is slightly more complicated. The main complications arise from the fact that (1) new (sub)reductions may take place in order to evaluate the conditions, and (2) new variables can be introduced in the conditions. As an example, consider the following conditional equation:

```
tc(Exp,  Env) = pair(Errors,  Type ),
tc(Exp', Env) = pair(Errors', Type'),      Type = Type'
==========================================================
tc(plus(Exp,Exp'), Env) -> pair(union(Errors,Errors'), Type)
```

It checks the correctness of a `plus` expression, given an environment containing the declared variables. To do this, the arguments of the `plus` have to be checked, which is expressed in the conditions. The check of each expression results in a pair consisting of a (possibly empty) list of errors and the inferred type of the expression.

Reduction of a term t according to this rule will typically proceed as follows. First, t is matched against the left-hand side, thus giving `Exp`, `Exp'`, and `Env` a value. Next, with these variables instantiated, the left-hand side of the first condition `tc(Exp,Env)` can be normalized. By matching this normal form against the right-hand side `pair(Errors, Type)`, the variables `Errors` and `Type` will obtain a value. In a similar way, `Errors'` and `Type'` will be bound to a value. The third condition only succeeds if `Type` and `Type'` (which are already in normal form) are identical. When all conditions succeed, all variables will have received a value, and the resulting contractum can be constructed. For a more detailed presentation of this way of conditional rewriting, and for a discussion of its consequences, we refer to [Wal91, Hen91], and Section 2.

But what about origins in the conditional case? The single-step origin relations do not change for the conditional case, but should be applied from the left-hand side via the conditions to the right-hand side. If evaluation of a condition involves a reduction of a term t, then the origins of the redex are passed to that term t, using 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 new variables, then these are matched against normal forms having obtained an origin already. These variables may be reused in other conditions, or in the right-hand side of the conclusion, thus giving the contractum its origins.

As an example, assume that the check of the first expression detects the error `undeclared(Id)`. This `Id` is a subterm of `Exp`, and is passed to the contractum as a subterm of `Errors`. The origin relations take care of this by first passing the `Exp` with all its origins to `tc(Exp,Env)`. Reducing this will result in a normal form in which one of the errors has the `Id` together with its origin as argument. By matching it against `Errors`, this variable "inherits" all origins, including the one for `Id`. The `Errors`, together with their origins, are passed to the right-hand side.

1.5 Points of Departure

The remainder of the paper will cover origins in full detail. We conclude the introduction by summarizing our points of departure:

- No assumptions are made regarding the choice of a particular reduction strategy.
- No assumptions are made concerning confluence or termination; origins can be established for arbitrary reductions in any TRS.
- The origin relation must 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 conceptually simple property that if t has an origin t', then t' can be rewritten to t in zero or more steps.
- An efficient implementation must exist.

2 Formal Definition

An origin is defined as a *set of paths* (occurrences), identifying a set of subterms of the initial term. A basic assumption in the subsequent definitions is that the complete history of the term rewriting process is available, consisting of all rewrite rules applied, all substitutions used, as well as the position of the next redex in each term but the last. The availability of the history is by no means essential to our definitions, but has the following advantages:

- The origin function for conditional TRSs can be defined in a declarative, non-recursive manner. A major problem we encountered with more operational definition methods is that they give rise to ill-behaved forms of recursion in the definition itself.
- Uniformity of the origin functions for unconditional TRSs and for conditional TRSs. The latter can be defined as an extension of the former.

As a consequence, we are able to determine the origin of any subterm of every term in the history.

The remainder of this section is organized as follows. First, we introduce some basic concepts and abbreviations concerning term rewriting. Second, the origin function for unconditional TRSs will be defined, and illustrated by way of an example. After giving a short introduction to conditional term rewriting we consider the—somewhat more complicated—origin function for conditional term rewriting systems.

2.1 Basic concepts

2.1.1 Terms, Paths and Rewrite Rules

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

Given a term t, the set of all valid paths in t is denoted by $O(t)$. Furthermore, let $\mathit{Vars}(t)$ denote the set of variables which occur in t. We will use $t' \subset t$ to express that t' appears as a subterm of t; the reflexive closure of $\subset$ is denoted by $\subseteq$. The negations of the previous two operators are written as $\not\subset$ and $\not\subseteq$, respectively.

We will impose the usual restrictions on unconditional rewrite rules: the left-hand side may not consist of a single variable, and the right-hand side may not contain variables which do not occur in the left-hand side. Lastly, the left-hand side and the right-hand side of a rewrite-rule r will be denoted by $\mathit{Lhs}(r)$ and $\mathit{Rhs}(r)$, respectively.

2.1.2 Reduction Sequences and Sequence Elements

The notion of a "rewriting history" will be formalized by way of *reduction sequences.* In the case of unconditional TRSs, the rewriting process can be regarded as a linear sequence of reduction steps $\mathcal{S}$. The individual elements $\mathcal{S}_i$ of a *reduction sequence*, $\mathcal{S}$, contain all information regarding the i^{th} rewrite step. Here, i ranges from 1 to $|\mathcal{S}|$ where $|\mathcal{S}|$ represents the length of sequence $\mathcal{S}$. Formally, $\mathcal{S}_i$ is a quadruple $(t_{\mathcal{S}:i},\ r_{\mathcal{S}:i},\ p_{\mathcal{S}:i},\ \sigma_{\mathcal{S}:i})$ where $t_{\mathcal{S}:i}$ is the i^{th} term of the sequence, $r_{\mathcal{S}:i}$ the i^{th} rewrite rule applied, $p_{\mathcal{S}:i}$ the path to the redex in the i^{th} term, and $\sigma_{\mathcal{S}:i}$ the substitution used in the i^{th} rewrite step[1].

[1]The last element of a reduction sequence is irregular, because the term associated with this element is in normal form. Consequently, no paths, rules and substitutions are associated with last elements. This sit-

We will use $\mathcal{S}$ (possibly with superscripts) to indicate reduction sequences, and $\mathcal{S}_i$ and s (again, possibly with superscripts) to indicate sequence elements. Finally, $Lhs(s)$ and $Rhs(s)$ are shorthands for $Lhs(r(s))$ and $Rhs(r(s))$, respectively (if s is not the last element of a sequence).

2.1.3 Common variables and common subterms

Two basic functions on terms which we will use frequently are *ComVars* and *ComTerms*. *ComVars* (Definition 2.1) takes a substitution σ and two open terms, t and t' as arguments, and computes the set containing all pairs $(p \cdot q,\ p' \cdot q)$ such that there is a variable X occurring at path p in t, and at path p' in t' , and q is a path in X^σ. It is required that σ contains bindings for all variables occurring in t and t'.

Definition 2.1 (*ComVars*)

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

ComTerms (Definition 2.2) computes, given two open terms t and t', the set containing all pairs $(p,\ p')$ such that there is a non-variable term t'', which is a subterm at path p in t and at path p' in t'.

Definition 2.2 (*ComTerms*)

$$ComTerms(\ t,\ t') \equiv \{\ (p,\ p') \mid t/p = t'/p' \notin\ Vars(t)\ \}$$

2.2 The origin function for unconditional TRSs

2.2.1 Definition of *ORG*

The function *LR* (see Definition 2.3) defines the relations which are established as a direct result of the syntactical form of the left-hand and right-hand side of the rewrite-rule $r(s)$. The aforementioned relations are pairs of the form $(q,\ q')$, where q and q' are paths to a common subterm in the redex and contractum, respectively. These common subterms are either the subterm of a variable binding, or they occur directly in the sides of $r(s)$; this is expressed via the functions *ComVars* and *ComTerms*.

Definition 2.3 (*LR*) For s not the last element of a sequence:

$$LR(s) \equiv ComVars(\sigma(s),\ Lhs(s),\ Rhs(s)) \cup ComTerms(Lhs(s),\ Rhs(s))$$

Below, a formal definition of the origin function *ORG* for unconditional TRSs is given. Relations are expressed by way of *relate* clauses; a clause $relate(\mathcal{S}_i,\ p,\ \mathcal{S}_{i+1},\ p')$ denotes the fact that a subterm at path p in $t(\mathcal{S}_i)$ and a subterm at path p' in $t(\mathcal{S}_{i+1})$ are related. In **(u1)**, all relations within the redex in $t(\mathcal{S}_i)$ and the contractum in $t(\mathcal{S}_{i+1})$ are defined, excluding the top-symbols of redex and contractum. In **(u2)**, the fact that all symbols in the context of the redex remain unaltered is expressed, by relating all these symbols in $t(\mathcal{S}_i)$ and $t(\mathcal{S}_{i+1})$. In addition, the top-symbols of the redex and contractum are related by **(u2)**.

uation will be modeled by way of a special value *undefined*; $\mathcal{S}_{|\mathcal{S}|} = (t_{\mathcal{S}:|\mathcal{S}|},\ undefined,\ undefined,\ undefined)$.

$$\begin{array}{ll}\mathtt{A1}: & \mathtt{append(E,empty)} \rightarrow \mathtt{cons(E,empty)} \\ \mathtt{A2}: & \mathtt{append(A,cons(B,C))} \rightarrow \mathtt{cons(B,append(A,C))}\end{array}$$

Figure 9: A TRS for appending elements to lists.

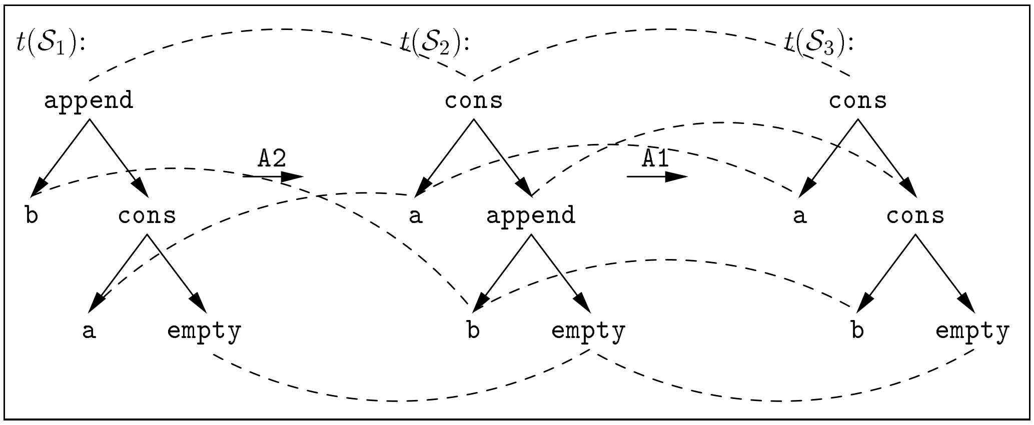

Figure 10: The normalization of `append(b, cons(a, empty))`. Origin relations are depicted by dashed lines.

Given a term $t(\mathcal{S}_i)$, and a path p within that term, the set of related subterms in the original term, $t(\mathcal{S}_0)$, is denoted by $ORG(t(\mathcal{S}_i),\ p)$. In fact, ORG is the transitive and reflexive closure of *relate*.

DEFINITION 2.4 (ORG) For $1 \leq i < |\mathcal{S}|$:

$$\begin{array}{lll}(\mathbf{u1}) & \forall(q,\ q') \in LR(\mathcal{S}_i): & relate(\mathcal{S}_i,\ p(\mathcal{S}_i)\cdot q,\ \mathcal{S}_{i+1},\ p(\mathcal{S}_i)\cdot q') \\ (\mathbf{u2}) & \forall p : (p \preceq p(\mathcal{S}_i)) \vee (p \mid p(\mathcal{S}_i)): & relate(\mathcal{S}_i,\ p,\ \mathcal{S}_{i+1},\ p)\end{array}$$

$$\begin{array}{ll}(\mathbf{u3}) & ORG(t(\mathcal{S}_1),\ p) \equiv \{\ p\ \} \\ (\mathbf{u4}) & ORG(t(s),\ p) \equiv \{\ q' \mid q' \in ORG(t(s'),\ q),\ relate(s',\ q,\ s,\ p)\ \}\ \ (s \neq \mathcal{S}_1)\end{array}$$

2.2.2 Example

As an example, we consider the TRS of Figure 9 which defines a function `append`. This function takes two arguments, an element and a list, and returns the list with the element appended to it. Lists are constructed by way of a constant `empty` which denotes an empty list, and a function `cons`. For example, the list `a,b` is denoted by `cons(a, cons(b, empty))`.

We consider the normalization of the term `append(b, cons(a, empty))`, which is depicted in Figure 10. In this figure, the origin relations are indicated by dashed lines. Below, we will show how these relations can be derived from Definition 2.4.

The corresponding reduction sequence has three elements. For the first element, $\mathcal{S}_1$, we have:

$$p(\mathcal{S}_1) \quad = \quad () \tag{1}$$

$$r(\mathcal{S}_1) \quad = \quad \mathtt{A2} \tag{2}$$

$$\sigma(\mathcal{S}_1) \quad = \quad \{\, \mathtt{A} = \mathtt{b},\ \ \mathtt{B} = \mathtt{a},\ \ \mathtt{C} = \mathtt{empty} \,\} \tag{3}$$

From (3) we derive:

$$O(A^{\sigma(\mathcal{S}_1)}) = O(B^{\sigma(\mathcal{S}_1)}) = O(C^{\sigma(\mathcal{S}_1)}) \quad = \quad \{\,()\,\} \tag{4}$$

From (2) – (4) and the definitions of *ComVars* and *ComTerms* (Definitions 2.1 and 2.2), it follows that:

$$\mathit{ComVars}(\sigma(\mathcal{S}_1),\ \mathit{Lhs}(\mathcal{S}_1),\ \mathit{Rhs}(\mathcal{S}_1)) \quad = \quad \{\, ((1),\ (2\,1)),\ ((2\,1),\ (1)),\ ((2\,2),\ (2\,2)) \,\} \tag{5}$$

$$\mathit{ComTerms}(\mathit{Lhs}(\mathcal{S}_1),\ \mathit{Rhs}(\mathcal{S}_1)) \quad = \quad \emptyset \tag{6}$$

Using (5) and (6) and the definition of *LR*, we obtain:

$$\mathit{LR}(\mathcal{S}_1) \quad = \quad \{\, ((1),\ (2\,1)),\ ((2\,1),\ (1)),\ ((2\,2),\ (2\,2)) \,\} \tag{7}$$

Consequently, according to **(u1)** of Definition 2.4, the *relate*-clauses (8) – (10) follow from (7); *relate*-clause (11) follows from **(u2)** and (1), since we have $p(\mathcal{S}_1) = () \preceq ()$.

$$\mathit{relate}(\mathcal{S}_1,\ (1),\ \mathcal{S}_2,\ (2\,1)) \tag{8}$$

$$\mathit{relate}(\mathcal{S}_1,\ (2\,1),\ \mathcal{S}_2,\ (1)) \tag{9}$$

$$\mathit{relate}(\mathcal{S}_1,\ (2\,2),\ \mathcal{S}_2,\ (2\,2)) \tag{10}$$

$$\mathit{relate}(\mathcal{S}_1,\ (),\ \mathcal{S}_2,\ ()) \tag{11}$$

In a similar fashion, we have:

$$\mathit{ComVars}(\sigma(\mathcal{S}_2),\ \mathit{Lhs}(\mathcal{S}_2),\ \mathit{Rhs}(\mathcal{S}_2)) \quad = \quad \{\, ((1),\ (1)) \,\} \tag{12}$$

$$\mathit{ComTerms}(\mathit{Lhs}(\mathcal{S}_2),\ \mathit{Rhs}(\mathcal{S}_2)) \quad = \quad \{\, ((2),\ (2)) \,\} \tag{13}$$

$$\mathit{LR}(\mathcal{S}_2) \quad = \quad \{\, ((1),\ (1)),\ ((2),\ (2)) \,\} \tag{14}$$

Thus, according to **(u1)** of Definition 2.4 and (12) – (14), the *relate*'s (15) – (16) are generated. The *relate*'s (17) – (19) are generated according to **(u2)** of Definition 2.4.

$$\mathit{relate}(\mathcal{S}_2,\ (2\,1),\ \mathcal{S}_3,\ (2\,1)) \tag{15}$$

$$\mathit{relate}(\mathcal{S}_2,\ (2\,2),\ \mathcal{S}_3,\ (2\,2)) \tag{16}$$

$$\mathit{relate}(\mathcal{S}_2,\ (),\ \mathcal{S}_3,\ ()) \tag{17}$$

$$\mathit{relate}(\mathcal{S}_2,\ (1),\ \mathcal{S}_3,\ (1)) \tag{18}$$

$$\mathit{relate}(\mathcal{S}_2,\ (2),\ \mathcal{S}_3,\ (2)) \tag{19}$$

As an example, we will now compute the origin $\mathit{ORG}(t(\mathcal{S}_3),\ (1))$ of the symbol `b` at path (1) in the normal form $t(\mathcal{S}_3)$. Using **(u4)** of Definition 2.4 and (18), the paths to related subterms in the previous term, $t(\mathcal{S}_2)$ can be determined:

$$\{\, p \mid \mathit{relate}(\mathcal{S}_2,\ p,\ \mathcal{S}_3,\ (1)) \,\} \quad = \quad \{\, (1) \,\} \tag{20}$$

Likewise, using **(u4)** of Definition 2.4 and (9), the paths to related subterms in the initial term $t(\mathcal{S}_1)$ are:

$$\{\, p \mid \mathit{relate}(\mathcal{S}_1,\ p,\ \mathcal{S}_2,\ (1)) \,\} \quad = \quad \{\, (2\,1) \,\} \tag{21}$$

Therefore, we have:

$$ORG(t(\mathcal{S}_3),\ (1)) \quad = \quad \{\, (2\ 1)\, \} \tag{22}$$

Informally stated, the origin of the constant `b` in the normal form consists of the constant `b` in the initial term.

We will conclude this example with some brief remarks. First, not all symbols are related to parts of the initial term. For example, the `cons` symbol at path (2) in $t(\mathcal{S}_3)$ is related to the `append` symbol at path (2) in $t(\mathcal{S}_2)$. However, as this symbol is not related to any symbol in $t(\mathcal{S}_1)$, in this case the origin consists of the empty set. A second remark is that we have chosen a trivial example in which no symbol has an origin containing more than one path. Such a situation may arise when variables or common subterms occur more than once in the left-hand side of a rule, or when a rule has a right-hand side consisting of a single variable or a common subterm.

2.3 Conditional Term Rewriting

2.3.1 Basic concepts

A conditional rewrite-rule takes the form:

$$\frac{l_1 = r_1,\ \cdots,\ l_n = r_n}{l \rightarrow r}$$

Note that the restrictions we imposed on *unconditional* rewrite rules (see subsection 2.1.1) are *not* imposed here. We evaluate conditions as a join system [Klo91]: both sides are instantiated and normalized. If the resulting normal forms match, the condition succeeds. As an extension, at most one side of a condition may introduce variables; we will refer to such variables as *new variables*. New variables may occur in subsequent conditions as well as in the right-hand side. Condition sides which introduce variables are *not* normalized; new variables are bound during the matching with the other, normalized, condition side.

Let $|r|$ be the number of conditions of r, and let $Cond(r,j)$ denote the j^{th} condition of r where $1 \leq j \leq |r|$. Moreover, the left-hand and right-hand side of the j^{th} condition of r are indicated by $Side(r,\ j,\ left)$ and $Side(r,\ j,\ right)$, respectively. Furthermore, let $\overline{left} = right$ and $\overline{right} = left$.

The function *VarIntro* (Definition 2.5) defines, given a rewrite rule r, where variables are introduced in the conditions of r. Informally, *VarIntro* computes triples $(X,\ j,\ side)$ indicating that the variable X was introduced in side *side* of the j^{th} condition of r.

Definition 2.5 (*VarIntro*)

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

We will frequently refer to the first and last element of a reduction sequence $\mathcal{S}$, which we will denote by $First(\mathcal{S})$ and $Last(\mathcal{S})$, respectively. Moreover, we will use the shorthands $Cond(s,\ j)$, $Side(s,\ j,\ side)$ and $VarIntro(s)$ for $Cond(r(s),\ j)$, $Side(r(s),\ j,\ side)$ and $VarIntro(r(s))$, respectively, if s is not the last element of a sequence.

2.3.2 A model for conditional term rewriting

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

1. Select a possible redex: find a match between a subterm t and the left-hand side of a rule r.
2. Evaluate the conditions of r: this entails the normalization of instantiated condition sides of r.
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 according to a CTRS as follows. The *initial reduction sequence* $\mathcal{S}^{init}$ starts with the initial term and contains sequence elements $\mathcal{S}_i^{init}$ that describe successive transformations of the initial term. Whenever a condition side v of rule $r(\mathcal{S}_i)$ is to be normalized, work on sequence element $\mathcal{S}_i$ is suspended. Control transfers to a *new* reduction sequence, $\mathcal{S}'$, starting with the term $t(\mathcal{S}'_1) = v^{\sigma(\mathcal{S}_i)}$. After the normalization of this term, control reverts to sequence $\mathcal{S}$.

In order to be able to express the complete history of the rewriting of a term, we assume the availability of a function $Seq(s, j, side)$ which indicates the reduction sequence starting with the instantiated side $side$ of the j^{th} condition of $r(s)$. In the following definition, we will assume that the history does not contain sequences for sides of conditions that failed. We may do so, because failing conditions do not contribute to the terms and to the computation of origins further on in the rewriting process.

In Figure 11, the rewriting of a term according to a CTRS is shown. In this figure, the application of $r(\mathcal{S}_1^{init})$ involves 4 new reduction sequences of which only the first, $\mathcal{S}'$, starting with an instantiated side of $Cond(\mathcal{S}_1^{init}, 1)$ is shown. The application of the first rule in $\mathcal{S}'$, $r(\mathcal{S}'_1)$ entails another 2 reduction sequences. Hence, the history of the rewriting of a term can be regarded as a tree with nodes corresponding to reduction sequences and arrows to indicate the transfer of control to another sequence due to the evaluation of a condition side. More precisely, respective arrows from left to right starting at sequence element s correspond to respective condition sides of $r(s)$ (in order of normalization). Therefore, the rewriting process can be regarded as a depth-first traversal of the tree, starting at $\mathcal{S}_1^{init}$.

2.4 The origin function for conditional TRSs

2.4.1 Basic origin functions

LR (Definition 2.3) defines relations between subsequent elements $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ of the same reduction sequence. Below, three more basic origin functions are presented which can be considered as cross-links: they define relations between elements of different sequences. In Figure 12, the basic origin functions for CTRSs are visualized.

LC (Definition 2.6) connects a sequence element s to the first element of a sequence s' which arose as a result of the normalization of a condition side of $r(s)$. In this case, the relations take the form of triples (q, q', s'), indicating that the subterm at path q in the redex, and the subterm at path q' in $t(s')$ are related.

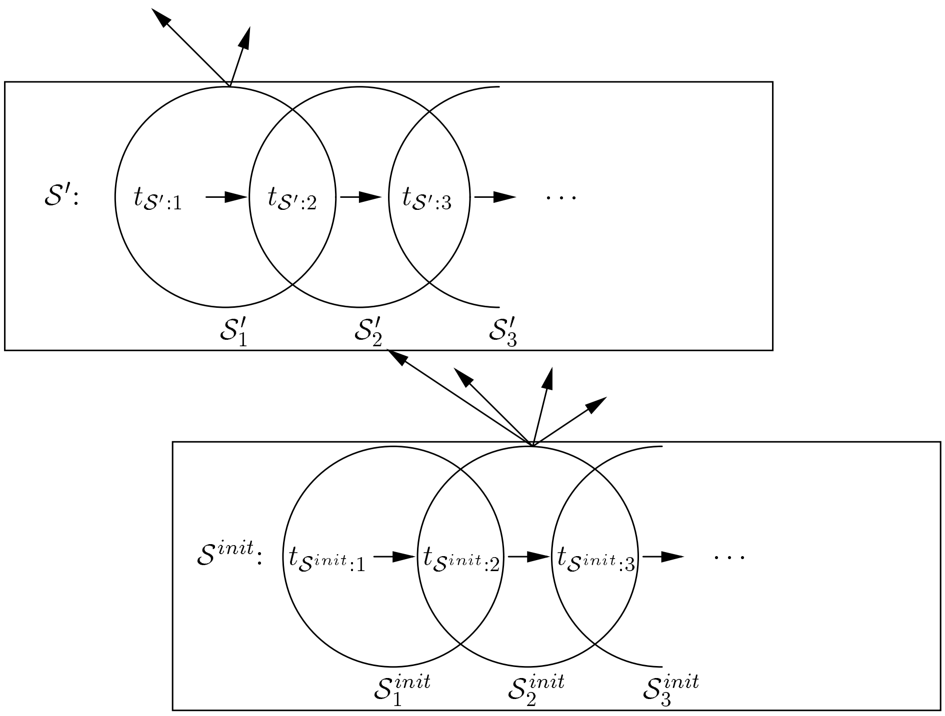

Figure 11: Reduction according to a CTRS.

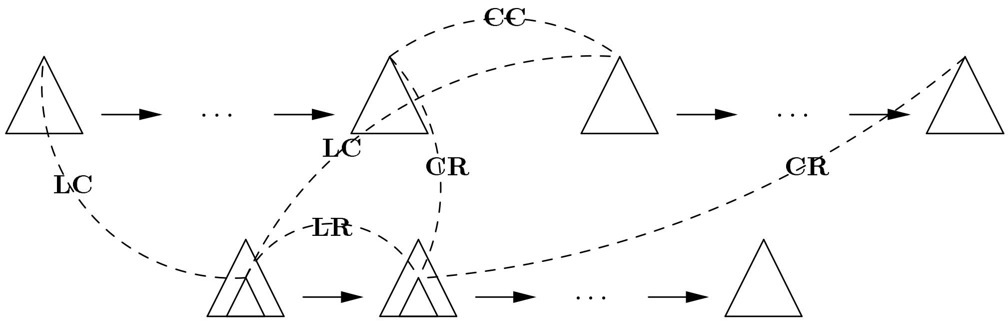

Figure 12: Basic origin relations. LR defines direct relations between the redex and the contractum. Three other basic origin relations provide links with new sequences for the normalization of condition sides. LC defines relations between the redex and the first element of a new sequence; CC links the last element of a new sequence to the first element of another new sequence; CR consists of relations between the last element of a new sequence and the contractum.

DEFINITION 2.6 (LC) For s not the last element of a sequence:

$$\begin{aligned} LC(s) \equiv \{\, (q,\ q',\ s') \mid s' = \mathit{First}(\mathit{Seq}(s,\ j,\ \mathit{side})),\ (1 \le j \le |r(s)|,\ \mathit{side} \in \{\ \mathit{left},\ \mathit{right}\ \}), \\ (q,\ q') \in (\quad \mathit{ComVars}(\sigma(s), \mathit{Lhs}(s),\ \mathit{Side}(s,\ j,\ \mathit{side})) \cup \\ \mathit{ComTerms}(\mathit{Lhs}(s), \mathit{Side}(s,\ j,\ \mathit{side}))\ )\ \} \end{aligned}$$

CR (Definition 2.7) connects the last sequence element s' of a sequence which appeared as a result of the normalization of a condition side of $r(\mathcal{S}_i)$ to the sequence element $\mathcal{S}_{i+1}$. The established relation consists of 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[2]. CR-relations reflect the fact that a new variable which was introduced in a condition side occurs in the right-hand side of a rule.

DEFINITION 2.7 (CR) For s not the last element of a sequence:

$$\begin{aligned} CR(s) \equiv \{(q,\ q',\ s') \mid (X,\ h,\ \mathit{side}) \in \mathit{VarIntro}(s),\ s' = \mathit{Last}(\mathit{Seq}(s,\ h,\ \overline{\mathit{side}})), \\ (q,\ q') \in \mathit{ComVars}(\sigma(s),\ \mathit{Side}(s,\ h,\ \mathit{side}),\ \mathit{Rhs}(s))\ \} \end{aligned}$$

Finally, CC (see Definition 2.8) connects the last element s' of a sequence $\mathcal{S}'$ to the first element s'' of another sequence $\mathcal{S}''$. Here, $\mathcal{S}'$ is the sequence for the normalized side of a condition which introduced a variable, and $\mathcal{S}''$ is a sequence for a subsequent condition side in which the variable under consideration occurs. CC-relations take the form of quadruples $(q,\ q',\ s',\ s'')$ which express that the subterm at path q in $t(s')$ is related to the subterm at path q' in $t(s'')$.

DEFINITION 2.8 (CC) For s not the last element of a sequence:

$$\begin{aligned} CC(s) \equiv \{(q,\ q',\ s',\ s'') \mid (X,\ h,\ \mathit{side}) \in \mathit{VarIntro}(s),\ s' = \mathit{Last}(\mathit{Seq}(s,\ h,\ \overline{\mathit{side}})), \\ s'' = \mathit{First}(\mathit{Seq}(s,\ k,\ \mathit{side}')),\ (h < k \le |r(s)|,\ \mathit{side}' \in \{\ \mathit{left},\ \mathit{right}\ \}), \\ (q,\ q') \in \mathit{ComVars}(\sigma(s),\ \mathit{Side}(s,\ h,\ \mathit{side}),\ \mathit{Side}(s,\ k,\ \mathit{side}'))\ \} \end{aligned}$$

2.4.2 Definition of $CORG$

The origin function $CORG$ for CTRSs (Definition 2.9) is basically an extension of ORG. Although there seem to be no differences between **(u1)** and **(u2)** of Definition 2.4, and **(c1)** and **(c2)**, the sequence element $\mathcal{S}_i$ may now be part of any reduction sequence $\mathcal{S}$ which appears during the rewriting process. The *relate* clauses which are defined in **(c3)** provide the link between $\mathrm{t}(\mathcal{S}_{i-1})$, and the first elements s' of sequences starting with instantiated condition sides of $\mathrm{r}(\mathcal{S}_{i-1})$.

Cases **(c4)** and **(c5)** are more complicated. Informally stated, case **(c4)** relates the introduction of variables in a condition of a rule $r(\mathcal{S}_{i-1})$, to uses of that variable in the right-hand side of $r(\mathcal{S}_{i-1})$. Similarly, **(c5)** defines *relate*-clauses which correspond to the relation of the introduction of a variable in a condition, and the use of that variable in a subsequent condition.

$CORG$ is the origin function for CTRSs. When applied to a term $t(s)$ and a path p in that term, it computes a set of paths to related subterms in $t(\mathcal{S}_1^{init})$. Once again, the origin

[2]Here, s' is the last element of the reduction-sequence associated with the normalized side $\overline{\mathit{side}}$ of condition h. Informally, subterms of the normalized side of a condition are "assigned" to the new variables they are matched against. Following uses of a new variable in the right-hand side result in relations with all subterms of the normalized condition side that defined its "value".

$$
\begin{array}{ll}
\mathtt{A1:} & \mathtt{append(E, empty)} \rightarrow \mathtt{cons(E, empty)} \\
\mathtt{A2:} & \mathtt{append(A, cons(B, C))} \rightarrow \mathtt{cons(B, append(A, C))} \\
\mathtt{R1:} & \mathtt{rev(empty)} \rightarrow \mathtt{empty} \\
\mathtt{R2:} & \dfrac{\mathtt{F = rev(E)}}{\mathtt{rev(cons(D, E))} \rightarrow \mathtt{append(D, F)}}
\end{array}
$$

Figure 13: A CTRS for reversing lists.

function is the transitive closure of *relate*. A major difference with the unconditional case, however, is that the "predecessor" of a sequence element $\mathcal{S}_i$ is no longer uniquely defined. Whenever $i > 1$, $\mathcal{S}_{i-1}$ is a predecessor of $\mathcal{S}_i$. Furthermore, for all conditions $Cond(\mathcal{S}_{i-1}, j)$ which introduce new variables in side *side*, the sequence elements $Last(Seq(\mathcal{S}_{i-1}, j, \overline{side}))$ are predecessors of $\mathcal{S}_i$ as well.

DEFINITION 2.9 ($CORG$) For $1 \leq i < |\mathcal{S}|$:

$$
\begin{array}{lll}
(\mathbf{c1}) & \forall (q,\, q') \in LR(\mathcal{S}_i): & relate(\mathcal{S}_i,\, p(\mathcal{S}_i) \cdot q,\, \mathcal{S}_{i+1},\, p(\mathcal{S}_i) \cdot q') \\
(\mathbf{c2}) & \forall p : (p \preceq p(\mathcal{S}_i)) \vee (p \mid p(\mathcal{S}_i)): & relate(\mathcal{S}_i,\, p,\, \mathcal{S}_{i+1},\, p) \\
(\mathbf{c3}) & \forall (q,\, q',\, s') \in LC(\mathcal{S}_i): & relate(\mathcal{S}_i,\, p(\mathcal{S}_i) \cdot q,\, s',\, q') \\
(\mathbf{c4}) & \forall (q,\, q',\, s') \in CR(\mathcal{S}_i): & relate(s',\, q,\, \mathcal{S}_{i+1},\, p(\mathcal{S}_i) \cdot q') \\
(\mathbf{c5}) & \forall (q,\, q',\, s',\, s'') \in CC(\mathcal{S}_i): & relate(s',\, q,\, s'',\, q')
\end{array}
$$

$$
\begin{array}{ll}
(\mathbf{c6}) & CORG(t(\mathcal{S}_1^{init}),\, p) \equiv \{\, p \,\} \\
(\mathbf{c7}) & CORG(t(s),\, p) \equiv \{\, q' \mid q' \in CORG(t(s'),\, q),\; relate(s',\, q, s,\, p) \,\} \;\; (s \neq \mathcal{S}_1^{init})
\end{array}
$$

2.4.3 Example

As an example, we will consider the CTRS of Figure 13 of a function `rev` for reversing lists. In rule `R2`, a new variable `F` is introduced in the left-hand side of its condition. Actually, the use of a new variable is not necessary in this case: we may alternatively write `append(D, rev(E))` for the right-hand side of `R2`. The new variable is used solely for the sake of illustration.

In Figure 14, the rewriting of the term `rev(cons(a, cons(b, empty)))` is shown. Three reduction sequences occur in this rewriting process. Besides the initial sequence $\mathcal{S}^{init}$, two new sequences $\mathcal{S}^1$ and $\mathcal{S}^2$ appear as a result of two applications of the conditional rule `R2`. In Figure 14, origin relations are shown by way of dashed lines. Dashed lines denote LR-relations, dashed arrows going up indicate LC-relations, and dashed arrows going down depict CR-relations. In the example, the following connections between reduction sequences exist:

$$Seq(\mathcal{S}_1^{init},\, 1,\, right) \;=\; \mathcal{S}^1 \tag{23}$$

$$Seq(\mathcal{S}_1^1,\, 1,\, right) \;=\; \mathcal{S}^2 \tag{24}$$

Below, we will determine the set of *relate*-clauses which is generated for this example. For $\mathcal{S}_1^{init}$, we have:

$$p(\mathcal{S}_1^{init}) \;=\; () \tag{25}$$

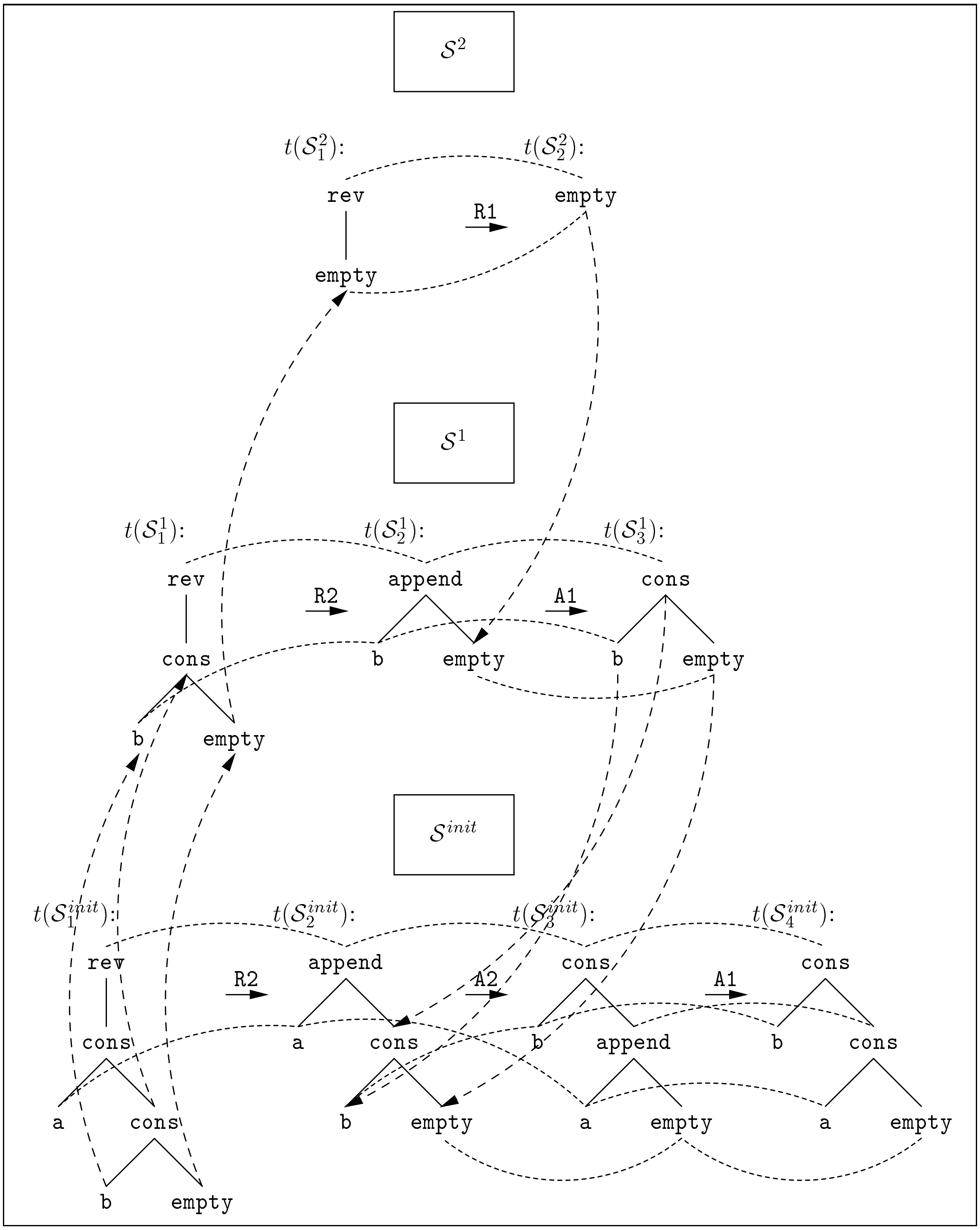

Figure 14: The normalization of `rev(cons(a, cons(b, empty)))`. Origin relations are indicated by way of dashed lines. Dashed lines without arrows indicate LR-relations; dashed arrows going up denote LC-relations, and dashed arrows going down depict CR-relations.

$$\sigma(\mathcal{S}_1^{init}) \;=\; \{\, \mathtt{D} = \mathtt{a},\ \ \mathtt{E} = \mathtt{cons}(\mathtt{b},\ \ \mathtt{empty}),\ \ \mathtt{F} = \mathtt{cons}(\mathtt{b},\ \ \mathtt{empty}) \,\} \tag{26}$$

From (26), we derive:

$$O(\mathtt{D}^{\sigma(\mathcal{S}_1^{init})}) \;=\; \{\, ()\, \} \tag{27}$$

$$O(\mathtt{E}^{\sigma(\mathcal{S}_1^{init})}) = O(\mathtt{F}^{\sigma(\mathcal{S}_1^{init})}) \;=\; \{\, (),\ (1),\ (2)\, \} \tag{28}$$

A new variable F is introduced in the left-hand side of R2; thus according to Definition 2.5 we have :

$$\mathit{VarIntro}(\mathcal{S}_1^{init}) \;=\; \{\, (\mathtt{F},\ 1,\ \mathit{left})\, \} \tag{29}$$

By analysis of the syntactical shape of R2, and using (26) – (27) and the definitions of *ComVars*, *ComTerms* (Definitions 2.1 and 2.2) and *LR* (Definition 2.3), we obtain:

$$\mathit{ComVars}(\sigma(\mathcal{S}_1^{init}),\ \mathit{Lhs}(\mathcal{S}_1^{init}),\ \mathit{Rhs}(\mathcal{S}_1^{init})) \;=\; \{\, ((1\,1),\ (1))\, \} \tag{30}$$

$$\mathit{ComTerms}(\mathit{Lhs}(\mathcal{S}_1^{init}),\ \mathit{Rhs}(\mathcal{S}_1^{init})) \;=\; \emptyset \tag{31}$$

$$\mathit{LR}(\mathcal{S}_1^{init}) \;=\; \{\, ((1\,1),\ (1))\, \} \tag{32}$$

Likewise, we compute $LC(\mathcal{S}_1^{init})$, $CC(\mathcal{S}_1^{init})$ and $CR(\mathcal{S}_1^{init})$:

$$LC(\mathcal{S}_1^{init}) \;=\; \{\, ((1\,2),\ (1),\ \mathcal{S}_1^1),\ ((1\,2\,1),\ (1\,1),\ \mathcal{S}_1^1),\ ((1\,2\,2),\ (1\,2),\ \mathcal{S}_1^1)\, \} \tag{33}$$

$$CC(\mathcal{S}_1^{init}) \;=\; \emptyset \tag{34}$$

$$CR(\mathcal{S}_1^{init}) \;=\; \{\, ((),\ (2),\ \mathcal{S}_3^1),\ ((1),\ (2\,1),\ \mathcal{S}_3^1),\ ((2),\ (2\,2),\ \mathcal{S}_3^1)\, \} \tag{35}$$

According to Definition 2.9, the following *relate*-clauses are generated for sequence element $\mathcal{S}_1^{init}$. Below, an indication **(c1)** – **(c5)** indicates which part of Definition 2.9 brought about the *relate* under consideration. Observe the correspondence between the lines in Figure 14, and these *relate*'s.

$$(\mathbf{c1}):\quad \mathit{relate}(\mathcal{S}_1^{init},\ (1\,1),\ \mathcal{S}_2^{init},\ (1)) \tag{36}$$

$$(\mathbf{c2}):\quad \mathit{relate}(\mathcal{S}_1^{init},\ (),\ \mathcal{S}_2^{init},\ ()) \tag{37}$$

$$(\mathbf{c3}):\quad \mathit{relate}(\mathcal{S}_1^{init},\ (1\,2),\ \mathcal{S}_1^1,\ (1)) \tag{38}$$

$$(\mathbf{c3}):\quad \mathit{relate}(\mathcal{S}_1^{init},\ (1\,2\,1),\ \mathcal{S}_1^1,\ (1\,1)) \tag{39}$$

$$(\mathbf{c3}):\quad \mathit{relate}(\mathcal{S}_1^{init},\ (1\,2\,2),\ \mathcal{S}_1^1,\ (1\,2)) \tag{40}$$

$$(\mathbf{c4}):\quad \mathit{relate}(\mathcal{S}_3^1,\ (),\ \mathcal{S}_2^{init},\ (2)) \tag{41}$$

$$(\mathbf{c4}):\quad \mathit{relate}(\mathcal{S}_3^1,\ (1),\ \mathcal{S}_2^{init},\ (2\,1)) \tag{42}$$

$$(\mathbf{c4}):\quad \mathit{relate}(\mathcal{S}_3^1,\ (2),\ \mathcal{S}_2^{init},\ (2\,2)) \tag{43}$$

Similarly, the following *relate*'s are generated for $\mathcal{S}_2^{init}$:

$$(\mathbf{c1}):\quad \mathit{relate}(\mathcal{S}_2^{init},\ (1),\ \mathcal{S}_3^{init},\ (2\,1)) \tag{44}$$

$$(\mathbf{c1}):\quad \mathit{relate}(\mathcal{S}_2^{init},\ (2\,1),\ \mathcal{S}_3^{init},\ (1)) \tag{45}$$

$$(\mathbf{c1}):\quad \mathit{relate}(\mathcal{S}_2^{init},\ (2\,2),\ \mathcal{S}_3^{init},\ (2\,2)) \tag{46}$$

$$(\mathbf{c2}):\quad \mathit{relate}(\mathcal{S}_2^{init},\ (),\ \mathcal{S}_3^{init},\ ()) \tag{47}$$

The *relate*'s for $\mathcal{S}_3^{init}$ are:

$$
\begin{aligned}
(\mathbf{c1}): &\quad relate(\mathcal{S}_3^{init},\ (2\,1),\ \mathcal{S}_4^{init},\ (2\,1)) && (48)\\
(\mathbf{c1}): &\quad relate(\mathcal{S}_3^{init},\ (2\,2),\ \mathcal{S}_4^{init},\ (2\,2)) && (49)\\
(\mathbf{c2}): &\quad relate(\mathcal{S}_3^{init},\ (),\ \mathcal{S}_4^{init},\ ()) && (50)\\
(\mathbf{c2}): &\quad relate(\mathcal{S}_3^{init},\ (1),\ \mathcal{S}_4^{init},\ (1)) && (51)\\
(\mathbf{c2}): &\quad relate(\mathcal{S}_3^{init},\ (2),\ \mathcal{S}_4^{init},\ (2)) && (52)
\end{aligned}
$$

Turning to sequence $\mathcal{S}^1$, we compute the following *relate*'s for element $\mathcal{S}_1^1$:

$$
\begin{aligned}
(\mathbf{c1}): &\quad relate(\mathcal{S}_1^1,\ (1\,1),\ \mathcal{S}_2^1,\ (1)) && (53)\\
(\mathbf{c2}): &\quad relate(\mathcal{S}_1^1,\ (),\ \mathcal{S}_2^1,\ ()) && (54)\\
(\mathbf{c3}): &\quad relate(\mathcal{S}_1^1,\ (1\,2),\ \mathcal{S}_1^2,\ (1)) && (55)\\
(\mathbf{c4}): &\quad relate(\mathcal{S}_2^2,\ (),\ \mathcal{S}_2^1,\ (2)) && (56)
\end{aligned}
$$

For element $\mathcal{S}_2^1$, the following *relate*'s are computed:

$$
\begin{aligned}
(\mathbf{c1}): &\quad relate(\mathcal{S}_2^1,\ (1),\ \mathcal{S}_3^1,\ (1)) && (57)\\
(\mathbf{c1}): &\quad relate(\mathcal{S}_2^1,\ (2),\ \mathcal{S}_3^1,\ (2)) && (58)\\
(\mathbf{c2}): &\quad relate(\mathcal{S}_2^1,\ (),\ \mathcal{S}_3^1,\ ()) && (59)
\end{aligned}
$$

Finally, the following *relate*'s are computed for element $\mathcal{S}_1^2$ of sequence $\mathcal{S}^2$:

$$
\begin{aligned}
(\mathbf{c1}): &\quad relate(\mathcal{S}_1^2,\ (1),\ \mathcal{S}_2^2,\ ()) && (60)\\
(\mathbf{c2}): &\quad relate(\mathcal{S}_1^2,\ (),\ \mathcal{S}_2^2,\ ()) && (61)
\end{aligned}
$$

Having derived all *relate*-clauses, we will now compute the origin of the symbol `b` at path (1) in the normal form, $CORG(t(\mathcal{S}_3^{init}),\ (1))$, by recursively determining the related paths in the "predecessors" of $t(\mathcal{S}_4^{init})$. Using (51), (45), (42), (57), (53) and (39) we have:

$$
\begin{aligned}
\{\,(s,\ p) \mid relate(s,\ p,\ \mathcal{S}_4^{init},\ (1))\,\} &= \{\,(\mathcal{S}_3^{init},\ (1))\,\} && (62)\\
\{\,(s,\ p) \mid relate(s,\ p,\ \mathcal{S}_3^{init},\ (1))\,\} &= \{\,(\mathcal{S}_2^{init},\ (2\,1))\,\} && (63)\\
\{\,(s,\ p) \mid relate(s,\ p,\ \mathcal{S}_2^{init},\ (2\,1))\,\} &= \{\,(\mathcal{S}_3^1,\ (1))\,\} && (64)\\
\{\,(s,\ p) \mid relate(s,\ p,\ \mathcal{S}_3^1,\ (1))\,\} &= \{\,(\mathcal{S}_2^1,\ (1))\,\} && (65)\\
\{\,(s,\ p) \mid relate(s,\ p,\ \mathcal{S}_2^1,\ (1))\,\} &= \{\,(\mathcal{S}_1^1,\ (1\,1))\,\} && (66)\\
\{\,(s,\ p) \mid relate(s,\ p,\ \mathcal{S}_1^1,\ (1\,1))\,\} &= \{\,(\mathcal{S}_1^{init},\ (1\,2\,1))\,\} && (67)
\end{aligned}
$$

Consequently, using (62) – (67) and Definition 2.9, we obtain:

$$
CORG(t(\mathcal{S}_4^{init}),\ (1)) \;=\; \{\,(1\,2\,1)\,\} \qquad (68)
$$

Thus, the `b` in the normal form, $t(\mathcal{S}_4^{init})$, is related to the `b` in the initial term, $t(\mathcal{S}_1^{init})$. Notice that the 6 *relate*'s that were applied in the above computation correspond to a 6-step walk in the graph of Figure 14 from the occurrence of `b` in the normal form to the occurrence of `b` in the initial term.

3 Properties

Given the formal definition, we study several interesting properties of origins. We claim a form of *soundness* for *CORG*, and briefly consider alternative origin schemes. Moreover, we investigate when an origin consists of at least one path (*non-empty* origins), at most one path (*precise* origins) or of exactly one path (*unitary* origins). We pay some attention to the many-sorted case which, surprisingly, for some cases allows a reduction of the complexity of computing origins. Finally, we study origins in relation to existing notions such as *descendants* or *primitive recursive schemes*.

3.1 Soundness

Given a rewriting history, let an *origin scheme* be any function *Or* which for every term t occurring in the history, and path p in t, computes a set of paths in the original term. We define a simple soundness criterion on origin schemes, by requiring that if initial subterm t is an origin of some intermediate term t', then t can be rewritten to t' in zero or more steps. Formally:

Definition 3.1 : Let t be an arbitrary term such that $t \equiv t(s)$ for some sequence element s in a rewriting history, and let t_1 be the initial term $t(\mathcal{S}_1^{init})$ in that history. An origin scheme *Or* is *sound* if for every t, and path $p \in O(t)$:

$$q \in Or(t,p) \Rightarrow t_1/q \rightarrow^* t/p$$

To prove the soundness of *CORG*, we first prove that *relate* only relates terms such that they are either syntactically equal, or that one can be rewritten to the other in exactly one step. Consider a reduction sequence $\mathcal{S}$, and an arbitrary subterm t' in $\mathcal{S}$ (i.e., there is a sequence element s' and a path p' such that $t(s')/p' \equiv t'$).

Property 3.1 If term t' is related to some term t in an earlier sequence element s, i.e., $t(s)/p = t$ for some path p, and $relate(s,p,s',p')$ holds, then either t and t' are identical, or t can be rewritten to t' 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. Referring to the formal definition, *ComVars* and *ComTerms* only combine identical terms, and consequently *LR*, *LC*, *CC*, and *CR* only establish *relate*'s between syntactically equal subterms. Likewise, (**c2**) establishes *relate*'s between equal terms in the context, and between the redex which can be rewritten in exactly one step to its contractum.

Property 3.2 *CORG* is sound.

This follows directly from Property 3.1, and from the fact that the origin relation *CORG* is defined as the transitive and reflexive closure of *relate*.

One could wonder whether it would be useful for origin schemes to satisfy the inversion of the soundness property as well; i.e., whether $t_1/q \rightarrow^* t/p \Rightarrow q \in Or(t,p)$. This would mean that whenever an initial subterm t *can* be rewritten to some intermediate subterm t', then t is one of the origins of t'. Clearly, *CORG* does not satisfy this inverse-soundness property, since *CORG* relates what actually has been rewritten, rather than what *can* be rewritten (actually, already ORG does not satisfy this property). For instance, overlapping rules may introduce accidental possibilities for reduction, which are not related. Likewise, if the initial term contains accidental equal instantiations of different variables, then in general these are not related.

3.2 Non-empty, precise and unitary origins

Let r be a conditional rewrite rule. Then:

DEFINITION 3.2 r is a *X-collapse* rule if $Rhs(r)$ consists of the single variable X.

DEFINITION 3.3 r is a *s-collapse* rule if the (possibly open) subterm s occurs in $Lhs(r)$, and $Rhs(r)$ consists of the subterm s.

DEFINITION 3.4 A term t is *linear in X*, if the variable X occurs at most once in t. Furthermore, a rewrite-rule is *left-linear in X* if its left-hand side is linear in X.

DEFINITION 3.5 A term t is *linear in s*, if the (open) subterm s occurs at most once in t. A rewrite-rule r is *left-linear in s* if its left-hand side is linear in s.

Some examples: $r_1 : f(X) \to X$ is an X-collapse rule; $r_2 : f(g(X)) \to g(X)$ is a $g(X)$-collapse rule; $r_3 : f(g(X), g(X)) \to h(g(X))$ is neither left-linear in X, nor left-linear in $g(X)$.

The predicate $LinearIntro(X,\ r)$ tells us if a variable X is introduced linearly, in either the left-hand side of r or in a condition side of r. Formally:

DEFINITION 3.6 $LinearIntro(X, r) \Leftrightarrow X$ appears exactly once in $Lhs(r)$ or $(X,\ h,\ side) \in VarIntro(r)$ and $Side(r,\ h,\ side)$ is linear in X.

Let o be an origin, and let $|o|$ denote the number of paths in o. Then:

DEFINITION 3.7 o is *empty* $\Leftrightarrow |o| = 0$.

DEFINITION 3.8 o is *non-empty* $\Leftrightarrow |o| \geq 1$.

DEFINITION 3.9 o is *precise* $\Leftrightarrow |o| \leq 1$.

DEFINITION 3.10 o is *unitary* $\Leftrightarrow |o| = 1$.

In the sequel r will denote an arbitrary rule, $1 \leq j \leq |r|$ and $side \in \{\ left,\ right\ \}$.

3.2.1 Single-sorted CTRSs

PROPERTY 3.3 (Non-empty origins) All terms with top symbol f will 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 property can be proved by induction over all *relate*'s, after introducing an ordering on all sequence elements. Informally, all terms with top symbol f will have non-empty origins if nowhere an f is introduced that is not *relate*d to a "previous" f. Note that relations according to variables have no effect on origins being (non-)empty.

A term with top symbol f with an empty origin *may* arise when a symbol f which is not part of a common subterm occurs in the right-hand side (or in a condition side) of a rule.

PROPERTY 3.4 (Precise origins) All terms with top symbol f will have precise origins if for all open terms u with top function symbol f:

(1) The CTRS does not contain X-collapse rules
(2) The CTRS does not contain u-collapse rules
(3) $u \subset \mathit{Rhs}(r) \Rightarrow r$ is left-linear in u
(4) $u \subseteq \mathit{Side}(r,\, j,\, \mathit{side}) \Rightarrow r$ is left-linear in u
(5) $X \subseteq \mathit{Rhs}(r) \Rightarrow \mathit{LinearIntro}(X,\, r)$
(6) $X \subseteq \mathit{Side}(r,\, j,\, \mathit{side}) \Rightarrow \mathit{LinearIntro}(X,\, r)$

This property can also be proved by induction over all *relate*'s, in a similar fashion as the previous property. In essence, the proof verifies that no term with top function symbol f is introduced in a way that it is *relate*d to more than one "previous" term.

PROPERTY 3.5 (Unitary origins) All terms with top symbol f will have unitary origins if for all open terms u with top function symbol f:

(1) The CTRS does not contain X-collapse rules
(2) The CTRS does not contain u-collapse rules
(3) $u \subset \mathit{Rhs}(r) \Rightarrow u \subseteq \mathit{Lhs}(r)$ and r is left-linear in u
(4) $u \subseteq \mathit{Side}(r,\, j,\, \mathit{side}) \Rightarrow u \subseteq \mathit{Lhs}(r)$ and r is left-linear in u
(5) $X \subseteq \mathit{Rhs}(r) \Rightarrow \mathit{LinearIntro}(X,\, r)$
(6) $X \subseteq \mathit{Side}(r,\, j,\, \mathit{side}) \Rightarrow \mathit{LinearIntro}(X,\, r)$

The proof for this property is straightforward: "non-empty" and "precise" implies "unitary". Therefore, we obtain this property by combining Properties 3.3 and 3.4.

3.2.2 Many-sorted CTRSs

Until now we considered only single-sorted CTRSs, although all definitions apply to many-sorted CTRSs as well. Below, we discuss some specific properties of many-sorted CTRSs.

The first property follows from the fact that term rewriting is sort-preserving, i.e., a term and its reduct belong to the same sort. Thus, using Property 3.1, we know that *relate* only relates terms of the same sort. Since CORG is the transitive and reflexive closure of *relate*, we have:

PROPERTY 3.6 CORG is sort-preserving.

PROPERTY 3.7 *relate* can be partitioned into a set which contains one subrelation for each sort.

This follows directly from Property 3.6. In Section 4 we will describe an optimization of the implementation of CORG which makes use of this property.

DEFINITION 3.11 Given two sorts $\mathtt{S}$ and $\mathtt{T}$, we will write $\mathtt{S} \sqsubseteq \mathtt{T}$ if terms of sort $\mathtt{T}$ can contain subterms of sort $\mathtt{S}$.

PROPERTY 3.8 (Non-empty origins) All terms of sort $\mathtt{S}$ will have non-empty origins if for all open terms v of sort $\mathtt{S}$ which are not a single variable:

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

This property easily follows from the single-sorted case: if conditions (1) and (2) hold for all terms of sort $\mathtt{S}$, then they hold for all terms with a top symbol of that sort. Then, Property 3.3 can be applied.

PROPERTY 3.9 (Precise origins) All terms of sort $\mathtt{S}$ will have precise origins if for all open terms v of sort $\mathtt{S}$ which are not a single variable:

(1) The CTRS does not contain X-collapse rules with X of sort $\mathtt{S}$
(2) The CTRS does not contain v-collapse rules
(3) $v \subset \mathit{Rhs}(r) \Rightarrow r$ is left-linear in v
(4) $v \subseteq \mathit{Side}(r,\ j,\ \mathit{side}) \Rightarrow r$ is left-linear in v
(5) $X \subseteq \mathit{Rhs}(r) \Rightarrow \mathit{LinearIntro}(X,\ r)$, for all X of sorts $\mathtt{T}$ such that $\mathtt{S} \sqsubseteq \mathtt{T}$
(6) $X \subseteq \mathit{Side}(r,\ j,\ \mathit{side}) \Rightarrow \mathit{LinearIntro}(X,\ r)$, for all X of sorts $\mathtt{T}$ such that $\mathtt{S} \sqsubseteq \mathtt{T}$

Note that this property is not a straightforward extension of the single-sorted case because, in (5) and (6), we restricted our constraints to variables of sorts $\mathtt{T}$ such that $\mathtt{S} \sqsubseteq \mathtt{T}$. We cannot restrict ourselves to variables of sort $\mathtt{S}$ here because a non left-linear occurrence of a variable of sort $\mathtt{T}$ with $\mathtt{S} \sqsubseteq \mathtt{T}$ may cause the origin of a term of sort $\mathtt{S}$ to be non-precise. To see this, consider a rule r which contains two occurrences of a variable X of sort $\mathtt{T}$ in the left-hand side, and one occurrence of X in the right-hand side. Furthermore, suppose that X is bound to a term which contains a subterm of sort $\mathtt{S}$. Now if the two subterms of sort $\mathtt{S}$ in the redex both have origins containing exactly one path, the related subterm in the contractum will have an origin containing two paths, and will not be precise. Consequently, we have to take variables of all sorts $\mathtt{T}$ such that $\mathtt{S} \sqsubseteq \mathtt{T}$ into account.

PROPERTY 3.10 (Unitary origins) All terms of sort $\mathtt{S}$ will have unitary origins if for all open terms v of sort $\mathtt{S}$ which are not a single variable:

(1) The CTRS does not contain X-collapse rules with X of sort $\mathtt{S}$
(2) The CTRS does not contain v-collapse rules
(3) $v \subset \mathit{Rhs}(r) \Rightarrow v \subseteq \mathit{Lhs}(r)$ and r is left-linear in v
(4) $v \subseteq \mathit{Side}(r,\ j,\ \mathit{side}) \Rightarrow v \subseteq \mathit{Lhs}(r)$ and r is left-linear in v
(5) $X \subseteq \mathit{Rhs}(r) \Rightarrow \mathit{LinearIntro}(X,\ r)$, for all X of sorts $\mathtt{T}$ such that $\mathtt{S} \sqsubseteq \mathtt{T}$
(6) $X \subseteq \mathit{Side}(r,\ j,\ \mathit{side}) \Rightarrow \mathit{LinearIntro}(X,\ r)$, for all X of sorts $\mathtt{T}$ such that $\mathtt{S} \sqsubseteq \mathtt{T}$

This is a straightforward result of combining the Properties 3.8 and 3.9.

PROPERTY 3.11 (Partitioning of sorts) Through static analysis we can partition the sorts of a many-sorted CTRS in three parts:

- Sorts $\mathtt{S}$ such that we can guarantee that all terms of sort $\mathtt{S}$ will have *unitary* origins.

- Sorts `T` such that we can guarantee that all terms of sort `T` will have *non-empty* origins, but we cannot guarantee that all terms of sort `T` will have unitary origins.

- Sorts `U` such that we *cannot guarantee* that all terms of sort `U` will have either unitary or non-empty origins.

3.3 Descendants

Descendants [HL79] are used to study reappearances of redexes in orthogonal (only left-linear rules, and no overlapping rules [Klo91]) TRSs without conditional rules. 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 the common variables in the left-hand side and right-hand side of a rule are related. Note that because of non-ambiguity subredexes within a redex must be contained in the instantiations of the variables of the rewrite rule. Consequently, tracing function symbols (as we do in the "common-subterms" rule) is not interesting when studying reappearances of redexes in orthogonal TRSs.

The origin relation restricted to the "common variables rule" and the "context" rule, and applied only to orthogonal, non-conditional TRSs is equivalent to the descendants relation. Sometimes the notion of *quasi-descendant* is used [BK86]. Here the redex and contractum are also related. Thus, the origin relation without the "common-subterms" rule (again restricted to orthogonal, non-conditional TRSs) is equivalent to the quasi-descendants relation.

3.4 Primitive Recursive Schemes

A Primitive Recursive Scheme is a program scheme which formalizes the notion of functions defined inductively over some structure (for our applications typically a function like a type checker defined inductively over the abstract syntax of a programming language). The definition of PRSs can be found in [CFZ82]; In [Meu90] it is described how PRSs can be used for incremental rewriting and how PRSs can be extended to conditional specifications.

Informally stated, in a PRS a set of constructors G is defined, over which a set of functions Φ is defined inductively. In our case, this G represents the abstract syntax of the language, and a typical set of functions are the "tc" or "ev" functions in a type checker or evaluator. Equations defining these functions from Φ will have a constructor $p(x_1, ..., x_n)$ in their left-hand side, and will use $x_1, ..., x_n$ in their right-hand sides as arguments of other Φ-functions. One of the restrictions imposed on PRSs is that defining equations having a $p(x_1, ..., x_n)$ in their left-hand side, may use no other G-terms in the right-hand side than these x_i (i.e., defining equations are decreasing in G-terms).

This property is particularly interesting with respect to origins. Since G-terms are only passed as variables, and no new G-terms are introduced, every G-term occurring in the rewrite process will have a unitary origin (see also property 3.5). Consequently, for our applications, PRSs for evaluators or type checkers are guaranteed to have unitary origins for all syntax terms occurring throughout the rewriting process.

3.5 Discussion

The principles on which our origin relations are based, can be summarized as follows:

1. Relations involve "equal" subterms.
2. Relations can be derived statically from the rewrite-rules.
3. In cases where no distinction can be made between "intended" and "unintended" relations, all alternatives are included in the origin.

The motivation for the first principle is that the equality relation can be regarded as the maximal relation which is applicable in all situations. In the future, we plan to investigate extensions which, for example, reflect dependence relations.

The second principle was adopted because—in our opinion—relations which can only be established as a result of the rewriting of a particular term are *coincidental*, and therefore unwanted.

The third principle has repercussions in two situations. First, common subterms often appear in many-sorted TRSs in order to force a specific match, however, in many cases unintended occurrences of common subterms will appear. Second, when non left-linear rules or collapse rules are applied, more than one subterm is a candidate origin. In both cases, the writer of the TRS will often have some intuition which of these relations are intended, and which are not. Nevertheless, as we cannot make this distinction by examining the TRS, the origin relation incorporates all alternatives.

4 Implementation Aspects

An efficient algorithm which implements $CORG$ is now described. It is based on the following two principles. First, all origins are tracked *during* the rewriting process. Second, origins are stored "inside" terms, as annotations of symbols. As a result of this approach, no global history of the rewriting process has to be maintained.

The remainder of this section is organized as follows. We will start by deriving a simple implementation from the definition, which will be illustrated by an example. Several optimizations will be discussed next, resulting in an equivalent but much more efficient implementation. In ASF+SDF specifications, associative lists are allowed [Wal91], which can be regarded as functions with a variable number of arguments. We will present some minor extensions of origin tracking to handle lists. Furthermore, we will examine the consequences of representing terms as DAGs (directed acyclic graphs). For a detailed discussion of various aspects of subterm sharing, the reader is referred to [Mar92]. Using DAGs instead of trees results in a major efficiency improvement. However, we will show that cases exist in which the usage of DAGs results in inconsistent origins, with respect to the definition.

4.1 A first implementation

In Figure 15, some distinct moments in the process of conditional term rewriting are listed. We parameterize these moments with the sequence elements, condition numbers and condition sides involved, in order to ease referring to related parts of the definition.

In the following, we will frequently say that an origin o is propagated to path p in term t. By this we mean that all paths in o are *added to* the origin at path p in t.

We will use tables to store the origins of bindings of new variables in, containing triples (X, p, o) denoting that the set of paths o is included in the origin at path p in the binding of new variable X. The reason for introducing tables is that at the moments **side** and **apply**, the subterms matched against variables bound in previously normalized condition sides are no longer available, since we do not maintain a global history.

As argued before, the application of a rule is suspended during the evaluation of its conditions, and the evaluation of a condition entails the normalization of instantiated condition sides. Therefore, a *stack* of tables is needed, containing one table for each (suspended) application of a rule. We will use $Table(s)$ to denote the table associated with the application of rule $r(s)$. Moreover, ϵ will be used to denote an empty table.

The actions which take place at each of the points **init**, **match**, **side**, **cond** and **apply** are described schematically in Figure 16. Without going into too much detail, we will give some brief remarks on the equivalence of the formal definition and our implementation. Clearly, at the moment **init**, the initial annotations correspond to the origins of the initial term, according to **(c6)** of Definition 2.9. Furthermore, for each of the cases **(c1)**, **(c2)** and **(c3)** of Definition 2.9, the generation of a *relate*-clause directly corresponds to the propagation of annotations in the implementation. Whereas *relate*-clauses are generated for new variables in **(c4)** and **(c5)**, this consists of two separate actions in our implementation. At the moment **cond** origins are propagated to a table, and at the moments **side** and **apply** these origins are propagated from this table to an instantiated condition side and the contractum, respectively.

Thus far, no special attention was paid to failure of conditions. In the definition, we assumed that the rewriting history only contains sequence elements for successful conditions. We made this assumption because failure of a condition aborts the application of a rule; hence, no relations will be established with terms which appear further on in the rewriting process. As we cannot make the distinction between successful and failing conditions in advance during rewriting, it is unavoidable that origins will be propagated for failing conditions as well. This has no effect on the correctness with respect to the definition.

4.2 Example

As an example, we consider the CTRS for reversing lists of Figure 13 and the term `rev(cons(a, cons(b, empty)))`. The reader is referred to Figure 14 for the rewriting history of this term.

The situation at the moment $\mathbf{match}(\mathcal{S}_1^{init})$ is shown in Figure 17: the initial term $t(\mathcal{S}_1^{init})$ is annotated and the stack contains an empty table for the application of rule `A2`. In Figure 18, the situation is shown after the evaluation of the (first) condition of rule `A2`, at the moment $\mathbf{cond}(\mathcal{S}_1^{init}, 1)$. Figure 19 shows the annotations of the normal form, `cons(b, cons(a, empty))`, when the rewriting process has finished.

Moment	Parameters	Description
init	–	Before the rewriting process has started.
match	s	A match of a subterm and the left-hand side of rule $r(s)$ is found.
side	s, j, $side$, s'	Side $side$ of condition j of rule $r(s)$ is instantiated, creating the first element s' of a new sequence.
cond	s, j	Condition j of rule $r(s)$ is evaluated.
apply	s	The right-hand side of rule $r(s)$ is instantiated, and replaces the redex.

Figure 15: Distinct moments in the rewriting process.

Moment	Actions	Def.
init	The initial term is annotated: the symbol at path p in $t(\mathcal{S}_1^{init})$ is annotated with the set $\{ p \}$.	**(c6)**
	The stack of tables is empty.	–
match(s)	An empty table for s is pushed on the stack.	–
side$(s,\ j,\ side,\ s')$	$\forall(q,\ q',\ s') \in LC(s)$, the origin at path q in $t(s)$ is propagated to path q' in $t(s')$.	**(c3)**
	$\forall(X,\ p,\ o) \in Table(s)$ and $\forall q$: $Side(s,\ j,\ side)/q = X$, o is propagated to path $q \cdot p$ in $t(s')$.	**(c5)**
cond$(s,\ h)$	$\forall(X,\ h,\ side) \in VarIntro(s)$, the table on top of the stack, $Table(s)$, is extended: for each $p \in O(X^{\sigma(s)})$ and $\forall q$: $Side(X, h, side)/q = X$, an entry (X, p, o) is added, where o is the annotation at path $q \cdot p$ in $t(Last(Seq(s,\ h,\ \overline{side})))$.	**(c4), (c5)**
apply$(\mathcal{S}_i)$	$\forall(q,\ q') \in LR(\mathcal{S}_i)$, the origin at path q in $t(\mathcal{S}_i)$ is propagated to path q' in $t(\mathcal{S}_{i+1})$.	**(c1)**
	Since the context of the redex remains unaltered, the origins of all symbols in the context are implicitly propagated from $t(\mathcal{S}_i)$ to $t(\mathcal{S}_{i+1})$. Hence, **(c2)** can be implemented by solely propagating the origin at path $p(\mathcal{S}_i)$ in $t(\mathcal{S}_i)$ to path $p(\mathcal{S}_i)$ in $t(\mathcal{S}_{i+1})$.	**(c2)**
	$\forall(X,\ p,\ o) \in Table(\mathcal{S}_i)$ and $\forall q$: $Rhs(\mathcal{S}_i)/q = X$, o is propagated to path $q \cdot p$ in $t(\mathcal{S}_i)$.	**(c4)**
	The table for $\mathcal{S}_i$ is popped off the stack.	–

Figure 16: Actions for the implementation of $CORG$. In the first column, the moment and parameter values are listed. The second column specifies the actions taken for the given moment and parameter values. The third column lists the related parts of Definition 8.

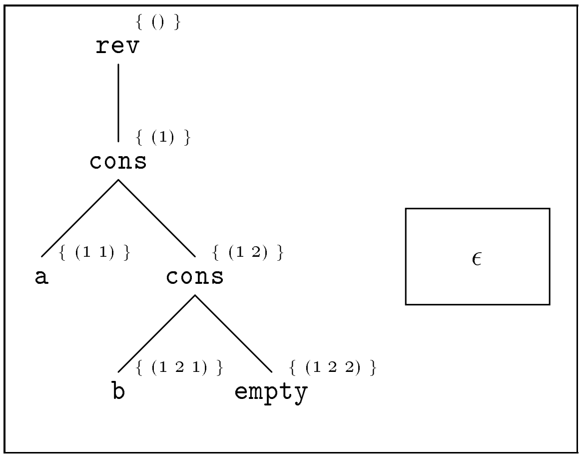

Figure 17: The situation at the moment $\mathbf{match}(\mathcal{S}_1^{init})$. The initial term is annotated, and the stack contains an empty table for the application of `R2`.

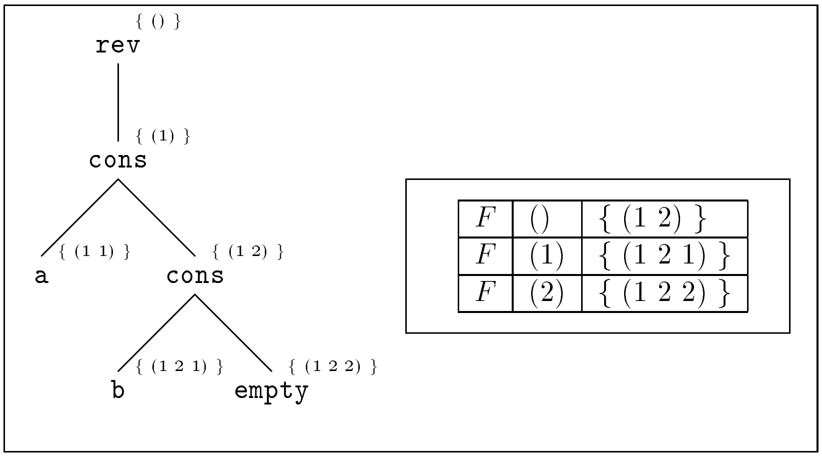

Figure 18: The situation at the moment $\mathbf{cond}(\mathcal{S}_1^{init},\ 1)$. The table on top of the stack now contains entries for the origins in the binding of new variable `F`.

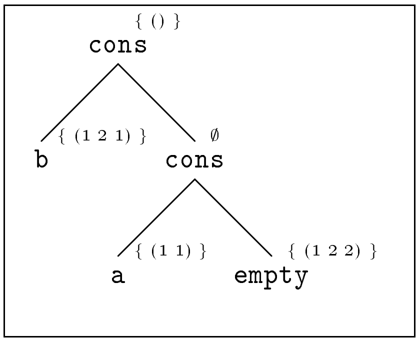

Figure 19: The situation when the rewriting process has finished.

4.3 Optimizations

4.3.1 Implicit origin tables

A first optimization of the algorithm consists of making origin tables implicit. This implicit representation is based on the fact that variable bindings are constructed explicitly for each sequence element, consisting of a bindings table with a variable-term pair for each variable occurring in the rewrite rule. The term in the bindings is a copy of the subterm that matched the variable. Initially, when a match is found with the left-hand side of a rule, only entries for the variables in the left-hand side are present. New entries are added after the evaluation of each condition which introduces variables. When control proceeds and a right-hand side or condition side is instantiated, the variables in this term are instantiated with the corresponding terms in the bindings.

Origin tables can now be represented implicitly by adding annotations to the terms appearing in the bindings table. For every term that is matched against a new variable at the end of the evaluation of a condition, the annotations of that term are propagated to the entry for that variable. Furthermore, variables bound during matching will be treated in a similar fashion. The advantages of this approach are twofold:

- No separate tables are required for storing origins.
- Actual propagation of origins is performed only once, when origins are propagated to the bindings. Propagation from bindings to terms takes place "automatically", during instantiation.

4.3.2 Pre-computing rule-dependent information

An obvious optimization is based on the fact that positions of variables and common subterms in a rewrite-rule are fixed. This positional information needs to be computed only once, and can be re-used for subsequent applications of that rewrite-rule. Our approach is to store the following positional information as annotations of rewrite-rules:

- *lhs-vars*: a table which contains all pairs (V, *path-list*), where V is a variable occurring in the left-hand side, and *path-list* is the list of paths where V occurs in the left-hand side.
- *rhs-subs*: a table which contains all pairs (*lhs-path-list*, *rhs-path-list*) for all common subterms (which are not a single variable) occurring at paths in *lhs-path-list* in the left-hand side and at paths in *rhs-path-list* in the right-hand side.
- SN*-vars* where $S \in \{$ *left*, *right* $\}$ and $1 \leq I \leq |r|$: a table containing all pairs (V, *path-list*), where V is a variable introduced in side S of the N^{th} condition of r. Again, *path-list* is a list containing all paths where V occurs.
- SN*-subs* where $S \in \{$ *left*, *right* $\}$ and $1 \leq I \leq |r|$: a table containing all pairs (*lhs-path-list*, SN*-path-list*) for all common subterms (which are not a single variable) occurring at paths in *lhs-path-list* in the left-hand side of r, and at paths in SN*-path-list* in side S of the N^{th} condition of r.

4.4 Extension to lists

Until now, we assumed CTRSs to be single-sorted, although our definition and implementation are applicable to many-sorted CTRSs as well. Furthermore, it was assumed that all functions had a fixed arity. The CTRSs we want to apply our algorithm to, however, are derived from equational specifications in which *associative lists* are commonly used. In this paper, lists will be regarded as functions with a variable number of arguments. In the following, variables bound to sublists will be referred to as *list variables*.

When lists and list variables are allowed, many possible matches may exist for a given left-hand side of a rule and a term, since *-list variables and +-list variables match sublists of arbitrary length greater than zero and one, respectively. Moreover, if a list match is found, and a condition of the involved rule fails, all other matches still have to be considered. In other words, a limited amount of backtracking is necessary; limited in the sense that once a rule is applied, its matches will never have to be reconsidered. For a detailed discussion of lists, list sorts and list matching, the reader is referred to [Hen91, Wal91].

List-matching introduces some minor complications with respect to origin tracking. Whereas every pair of occurrences of a non-list variable in the left-hand and the right-hand side (or a condition side) of a rule gives rise to relations between two subterms, a generalization of this scheme is necessary for list variables. Consider the following example where `l` is a list function, `X+` a +-list variable and `Y*` a *-list variable:

$$\mathtt{r}:\mathtt{f(l(X+,\ X+,\ a,\ Y*))} \rightarrow \mathtt{g(l(X+,\ a,\ Y*))}$$

If we rewrite the term `f(l(b, b, b, b, a))` according to `r`, the result consists of the term `g(l(b, b, a))`. Clearly, our intention is to relate the constants `a` in the redex and contractum. Intuitively, we furthermore want to relate the first and third `b` in the redex with the first `b` in the contractum, and the second and fourth `b` in the redex with the second `b` in the contractum, counting from left to right.

Generally, a list variable that matches with a sublist of length n gives rise to relations between n pairs of terms. More precisely, a list variable matching against sublists with the first element occurring at path $(l_1, \ldots, l_i)$ in the redex and at path $(m_1, \ldots, m_j)$ in the contractum gives rise to relations between the subterms at path $(l_1, \ldots, (l_i + x))$ in the redex and at path $(m_1, \ldots, (m_j + x))$ in the contractum, for all $0 \leq x < n$. Two special cases exist: if $n = 0$ then no terms are related at all (as variable `Y*` in our example), and if $n = 1$ we have the same situation as for non-list variables.

A second problem caused by allowing list variables, is the fact that positions in patterns no longer directly correspond to positions in the terms they are matched against. For instance, in the above example the constant `a` occurs at path (1 3) in the left-hand side of `r`, but at path (1 5) in the redex. Hence, argument positions "below" list functions which follow list variables are dependent on the bindings of the rewrite-step. We will refer to such positions as *relative* positions, as opposed to the *absolute* positions in the corresponding redex/contractum. Given a term at relative position j below a list function, the corresponding actual position can be determined by computing one plus the sum of the length of the previous (j-1) arguments counting every closed term and non-list variable as 1, and every list variable as the length of the sublist it matches. In our example, where the constant `a` occurs at relative position 3 below the list `l` in the left-hand side of `r`,

```
r1 :  f(X,  Y) → g(Y,  h(Y,  X))
r2 :  h(X,  X) → X
```

Figure 20: A TRS which potentially gives rise to inconsistent origins when sharing is allowed.

the corresponding absolute position is obtained by computing one plus the length of the arguments `X+` at positions 1 and 2, yielding the absolute position 5.

Our strategy is to encode relative positions by way of negative numbers. Whenever propagation is done, the corresponding absolute positions are computed. Consequently, the *lhs-vars* annotation of `r` will contain the entries (1 1) and (1 -2) for the variable `X+`.

4.5 Sharing

For reasons of efficiency, implementations of term rewriting systems allow sharing of subtrees, thus giving rise to DAGs instead of trees. We will assume that the initial term is represented as a tree, and that sharing is introduced via non-linear right-hand sides or condition sides. For every variable, the bindings table contains a *pointer* to one of the terms it was matched against. When a right-hand or condition side is instantiated, each occurrence of a variable will be replaced by the associated pointer in the bindings.

In several ways, origin tracking benefits from sharing. For every variable occurring exactly once in the left-hand side, the origins of the subterm matched against it will be propagated implicitly. The same obviously holds for every new variable which occurs exactly once in the "introducing" condition side. Therefore, we may delete from the *lhs-vars* and *SN-vars* annotations of every rule all entries (V, *pathlist*) where *pathlist* contains exactly one path. Moreover, if it is statically known which subterm is selected for the binding of a variable which occurs more than once, the remaining table-entries may be simplified by removing the appropriate path from *pathlist*.

Unfortunately, sharing also causes problems with respect to origin tracking. Suppose that a variable X occurs more than once in the left-hand side of a conditional rule r. When a match is found, the origins of all subterms of the redex matched against X are propagated to the bindings. However, this entry is merely a pointer to one of the subterms in the redex. Consequently, if a condition of r fails, an inconsistent situation arises, as one of the subterms that matched X now contains the annotations of all other subterms that matched X. A solution to this problem consists of *saving* the origins of the symbols in the redex when the match is found, and *restoring* these origins upon failure of a condition.

A more serious problem occurs when a rewrite-rule is not left-linear in, for example, variable X, and one of the subterms matched against X in the redex is already shared from "outside" the redex. If the entry for X in the bindings consists of a pointer to the already shared subterm, the pointer from "outside" will be inconsistent after the rewrite-step, according to our definition. This situation is illustrated in Figure 21, using the TRS of Figure 20 and the term `f(a, a)`.

A solution to this problem could be devised by relating origins specifically to parent-child relations in DAGs, instead of to symbols. An origin would be assigned to each such relation. Nodes without parents would have to be treated differently. Although we are

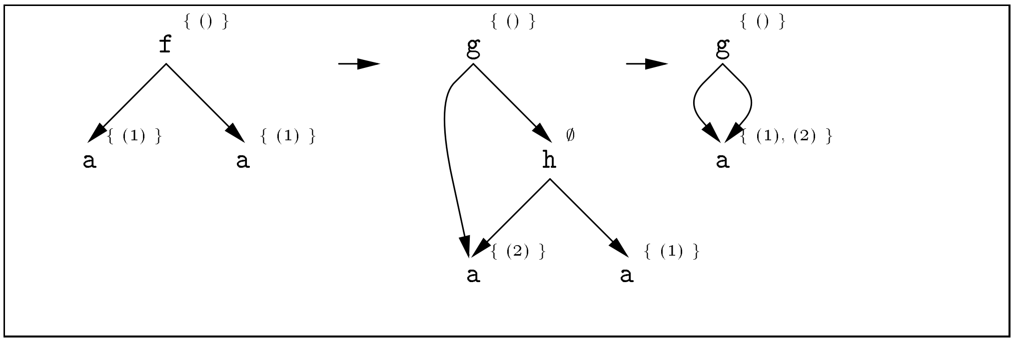

Figure 21: Inconsistent annotations due to sharing. According to the definition, the constant `a` in the normal form should have the origin $\{(2)\}$ when reached from path (1), and the origin $\{(1), (2)\}$ when reached from path (2). As a result of sharing, the incorrect origin $\{(1), (2)\}$ is found at path (1).

aware of the possibility of unexpected behaviour, no solution for the problem has been implemented for the following reasons:

- The problem sketched above does not occur for subterms which appear only once in the left-hand sides of the rewrite-rules. For our present applications, this is sufficient.
- The inconsistencies described above consist of origins that contain *too many* paths—clearly an undesirable situation. Still, we can be sure that an origin contains *at least* all paths it should contain, with respect to the definition.
- Implementing a solution based on indirections would be much less efficient, in terms of both time overhead and space overhead.

Future applications of origin tracking for other purposes may however lead to a situation where the implementation of a solution to this problem is necessary.

4.6 Restricting origin tracking to selected sorts

Often, we are only interested in the origins of subterms of a particular sort (or set of sorts). Property 3.7 of Section 3 states that *relate* can be partitioned into a set which contains one subrelation for each sort. This may lead to the belief that only propagations according to variables and common subterms of the selected sort are necessary. Unfortunately, this is not true—to see this, consider the following example. Suppose we are interested in the origins of terms of sort `S`, that terms of sort `T` may contain subterms of sort `S`, and that propagations are only done for sort `S`. Now consider the case that a rewrite rule is applied containing a variable X of sort `T` in both sides. Then, if the binding of X contains a subterm of sort `S`, its origin will not be tracked.

Consequently, if we are interested in the origins of subterms of sort `S`, we have to track origins for all sorts `T` such that subterms of sort `S` may occur in terms of sort `T` (i.e., $\mathtt{S} \sqsubseteq \mathtt{T}$ according to Definition 3.11).

When subterms are shared, restriction to selected sorts results in more efficiency improvement. The reason for this is that origins have to be tracked only for the non left-linear occurrences of variables and common subterms of sorts `T` which are higher in the $\sqsubseteq$ ordering.

5 Applications

Now that we have a better understanding of origins, we will present some applications in the area of automatic tool generation from formal specifications of programming languages. First, we cover the use of origins in algebraic specifications of static semantics of programming languages. Such specifications will typically produce error messages, and the user will be interested *where* in the source program he (or she) made the error. Origin tracking can be used to find these error locations automatically. Next, we will show how origin tracking can be used to construct generic debuggers. Two aspects of debuggers are studied in detail: animation of program execution, and the possibility to define various notions of breakpoints. The section is concluded by a brief description of a prototype implementation of origin tracking, which has been used to experiment with automatic error location, animation, and breakpoints in debuggers.

5.1 Positions of Errors in Source Programs

As an example of a typical equation in a type checker, consider a rule for checking the use of an identifier in an expression:

```
[i1]  lookup(Id,Env) = pair(false, undefined)
      ======================================
      tc-exp(Id,Env) -> error-list( not-declared(Id) )
```

The rule states that if an identifier `Id` cannot be found in the environment `Env` provided, the list containing the new error `not-declared` is the result of typechecking an expression using `Id`. The origin of `Id` in the subterm `not-declared(Id)` will precisely indicate for which occurrence of `Id` this error was detected. One can easily imagine some kind of tool receiving such an error message, which by analyzing the origins highlights the position in the source program where the user has made the mistake.

This works good because a typical type checker is defined inductively over the abstract syntax. Checking a complex language construct is expressed in terms of checking its substructures. As argued in Section 3.4, inductively defined rules will use variables to pass these structures from function to function (examples of rules with an inductive character can be found in Sections 1.2 and 1.4). The "common variable" origin rule will relate these variables. Consequently, the positions of pieces of source text will be indicated precisely by the origins. As a result, for instance, if an undeclared identifier is used somewhere, the origins of that identifier precisely indicate where this occurred, even if the program contains multiple occurrences of that identifier (some of which may be legal due to block structure).

This idea can be used quite well for many type checkers; for example, it correctly locates all errors for the specification of the type checker of an ML-subset [Hen91, BHK89], and for the error-message producing specification of the static semantics of Pico [BHK89]. Likewise, for the algebraic specification of the static semantics of Pascal [Deu91], origins provide locations for 20 out of the total of 50 error messages.

But why can this scheme not be used for the remaining 30 error messages of Pascal? The "problem" with these messages is that their arguments are computed intermediate structures, rather than actual pieces of the initial program. Consider for example a function type checking a range type as in Pascal:

```
[e1]  tc( range(Ch1,Ch2) ) -> tc( range(ord(Ch1), ord(Ch2)) )

[e2]  greater-than(Num1,Num2) = true
      ==========================================================
      tc( range(Num1,Num2) ) -> empty-range-not-allowed(Num1,Num2)
```

It checks whether a subrange type is not empty, i.e., whether the first value is smaller than the second one. Several range types are allowed, and character ranges (like `range("a","z")`) are checked by checking the associated ordinal values (i.e. `tc(range(1,26))`). Typically, computed structures such as these ordinal values will have empty origins. Consequently, if they appear as arguments in error messages, they cannot be used to locate the position of the error in the source program.

As a way to solve this problem, one can think of an approach related to the solution chosen for TYPOL [Kah87] (see Section 6). An implementation of TYPOL maintains a special variable (the *subject*) which keeps track of the syntactic construct currently processed. Rather than getting the locations out of the arguments of error messages, this scheme allows to retrieve the location of errors detected directly from the origin of that variable's value.

5.2 Generic debuggers

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. Applications of this feature can mainly be found in the areas of source-level debugging and tutoring. We are interested in the animation of program execution in a very fine-grained manner: not only do we want to visualize the moments that a particular declaration or statement is being processed, but in addition we want to visualize the evaluation of any expression. We will describe how this goal can be achieved easily, using our origin tracking technique.

A second usage of origin tracking in the area of debugging is the possibility to define various notions of breakpoints. Most source-level debuggers, such as gdb [SP91] and dbx [DT86], often have a completely fixed notion of a breakpoint, based on line-numbers, procedure calls and machine addresses. (Although the latest version of gdb allows breakpoints on the change of value of expressions.) We will indicate how the origin relation can be used to define breakpoints in a much more uniform and generic way.

The algebraic specification of a program-evaluator which is (partially) shown in Figure 22 below, will be used to illustrate the applications of origin tracking to debugging.

5.2.1 A sample specification

The language of Figure 22 contains only three kinds of statements: assignments, if-statements and while-statements. Subterms that represent statements are of sort `Statement`, and `Stats` is a list sort, denoting a list of statements. Furthermore, `Declaration` and `Declarations` are sorts for a single declaration and for a list of declarations, respectively. The sort `Program` corresponds to complete programs, which contain a list of declarations and a list of statements.

In order to specify an evaluator for this language, a set of functions is defined which together map a program to an "environment" containing variable-value pairs (sort `Env`; the

```
sorts
  Id Value Declaration Decls Expression Statement Stats Program Boolean Env EnvPair
functions
  program( Decls, Stats )          -> Program
  assign( Id, Expression )         -> Statement
  if( Expression, Stats, Stats )   -> Statement
  while( Expression, Stats )       -> Statement
  stats( Statement* )              -> Stats
  decls( Declaration* )            -> Decls

  eval-program( Stats, Decls )     -> Env
  eval-decl( Declaration )         -> EnvPair
  eval-decls( Decls )              -> Env
  eval-stat( Statement, Env )      -> Env
  eval-stats( Stats, Env )         -> Env
  eval-exp( Expression, Env )      -> Value
  lookup( Id, Env )                -> Value
  update( Id, Value, Env )         -> Env
  equal( Value, Value )            -> Boolean
equations
  [1] eval-program(program(Decls, Stats)) = eval-stats(Stats, eval-decls(Decls))
  [2] eval-stats(stats(), Env) = Env
  [3] eval-stats(stats(Stat, Stat*), Env) = eval-stats(stats(Stat*), eval-stat(Stat, Env))
  [4] eval-stat(assign(Id, Exp), Env) = update(Id, eval-exp(Exp, Env), Env)
  [5] equal(eval-exp(Exp, Env), 0) = false
      ============================================================
      eval-stat(if(Exp, Stats, Stats'), Env) = eval-stats(Stats, Env)
  [6] equal(eval-exp(Exp, Env), 0) = true
      ==============================================================
      eval-stat(if(Exp, Stats, Stats'), Env) = eval-stats(Stats', Env)
  [7] equal(eval-exp(Exp, Env), 0) = false
      =====================================
      eval-stat(while(Exp, Stats), Env) = Env
  [8] equal(eval-exp(Exp, Env), 0) = true
      ==================================================================================
      eval-stat(while(Exp, Stats), Env) = eval-stat(while(Exp, Stats), eval-stats(Stats, Env))
```

Figure 22: (Part of) an algebraic specification of a program evaluator.

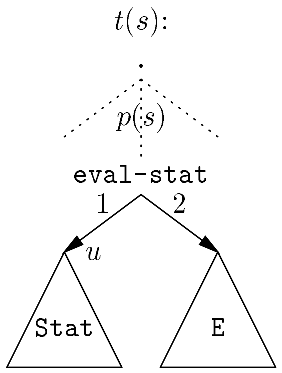

Figure 23: A redex at path $p(s)$ in the term matches the pattern `eval-stat(Stat, E)`. The statement being executed can be found at the paths in the initial term indicated by the origin u of the subterm $t(s)/p(s) \cdot (1)$.

exact form of environments is not specified here). The function `eval-program` takes a term of sort `Program` as argument, and computes the environment which results from executing the program. This function is expressed by way of two other functions, `eval-stats` and `eval-decls` (equation `[1]`). The latter of these, `eval-decls`, takes a term of sort `Decls` (a declaration section), and returns an environment containing the initial value of each variable. `Eval-decls` is specified in terms of `eval-decl` which processes one declaration. In equations `[2]` and `[3]`, we see how the evaluation of a list of statements (`eval-stats`) is expressed in terms of `eval-stat`, a function which computes the effect of executing one statement.

Some brief remarks on remaining functions: `update` updates the value of an identifier in an environment, returning the modified environment; `eval-exp` computes, given an expression and an environment, the value of that expression. In our example language, integer expressions are used in conditions of if-statements and while-statements (as for example in the language C). Observe, how the semantics of the if and while are expressed by way of a conditional equation (we omitted the specification of the function `equal`).

5.2.2 Animation of execution

We will now address the subject of animation of execution. In the sequel, it is assumed that a program $\mathcal{P}$ is represented by a corresponding program term `P`. $\mathcal{P}$ is executed by interpreting the equations of our specification as a CTRS, by reading them from left to right. Starting with the initial term, `eval-program(P)`, rules are applied until the normal form (an environment) is computed.

In the following, we will denote the part of the term currently being rewritten (i.e., the current redex) by $t = t(s)/p(s)$, where s is the "current" sequence element. Now, by examining our specification, we make the following observations:

OBSERVATION 5.1 A statement is executed when t matches the pattern (i.e., open term) `eval-stat(Stat, E)`, where `Stat` and `E` are variables of sorts `Statement` and `Env`, respectively.

OBSERVATION 5.2 A declaration is being processed when t matches the pattern `eval-decl(Decl)`, where `Decl` is a variable of the sort `Declaration`.

OBSERVATION 5.3 An expression is evaluated when t matches the pattern `eval-exp(Exp, E)`, where `Exp` and `E` are variables of sorts `Expression` and `Env`, respectively.

The next step is to apply origin tracking to the CTRS derived from our algebraic specification. In each of the cases listed above, the parts of the program being processed (i.e., statements, declarations and expressions) correspond to the origin of the subterm representing the statement, declaration or expression in the redex. Each of these subterms can be found at path (1) relative to the root of the redex.

Below, we will use u to denote the origin of the statement, declaration or expression being processed: $u = CORG(t(s), p(s) \cdot (1))$. This situation is illustrated in Figure 23.

OBSERVATION 5.4 If t matches one of the patterns `eval-stat(Stat, E)`, `eval-exp(Exp, E)` or `eval-decl(Decl)`, the parts of the program currently being processed can be found at the paths $p \in u$ in the initial term. Moreover, according to Property 3.10 we know that—

in this example—the origin of every expression, statement and declaration is unitary, i.e., exactly one subterm of the original term is related to each of these subterms.

Consequently, we can animate program execution as follows. Every redex is matched against the three patterns described above. Whenever a match succeeds, the origin of the subterm at path (1) from the root of the redex is retrieved, and the corresponding subterm of the initial term is high-lighted.

5.2.3 Breakpoints

We will show that—in our setting—the notion of a breakpoint is closely related to the animation of execution described previously. We will subsequently use the term "breakpoint" for any condition the current state of program execution should satisfy in order to give control to the user. Furthermore, we will make a distinction between *positional* breakpoints, and *position-independent* breakpoints. Positional breakpoints consist of a location in the source-text or a node in the abstract syntax tree; the breakpoint becomes effective when program execution reaches this position. Position-independent breakpoints are conditions on the current state of program-execution which do not depend on a particular location. Although the advantages of such breakpoints were already identified in [Bal69], few debugging systems actually incorporate these features [Han78, OCH91]. The "watchpoints" of gdb [SP91] are another example of position-independent breakpoints: execution is suspended when the value of a given expression changes. Below, we will outline how both types of breakpoints can be obtained.

Our starting point will be the conventional breakpoint on a statement: the user indicates a statement in the source-text of his program and program execution halts when this statement is reached. At that moment, the programmer may examine the values of variables, examine the call stack, set new breakpoints, and so on. In this paper, we will not study how these user commands can be implemented in our setting. Suppose that we want to break on a statement at path p in the initial term. Then, using Observations 5.1 and 5.4, we have:

Observation 5.5 The statement at path p in the initial term is executed when t matches `eval-stat(Stat, E)` and $p \in u$.

A positional breakpoint on the evaluation of an expression at path p in the initial term can be defined in a similar way:

Observation 5.6 The expression at path p in the initial term is evaluated when t matches `eval-exp(Stat, E)` and $p \in u$.

Finally, we consider two simple cases of position-independent breakpoints: the assignment to a specific variable and the reference to a specific variable. As mentioned, the key issue is that the location where the actions, assignment and referencing, take place is not fixed. However, using one of these position-independent breakpoints, the origin relation will reflect the position in the program where the assignment or reference takes place.

From the fact that the assignment of a value to a variable corresponds to an application of the function `update`, it follows that:

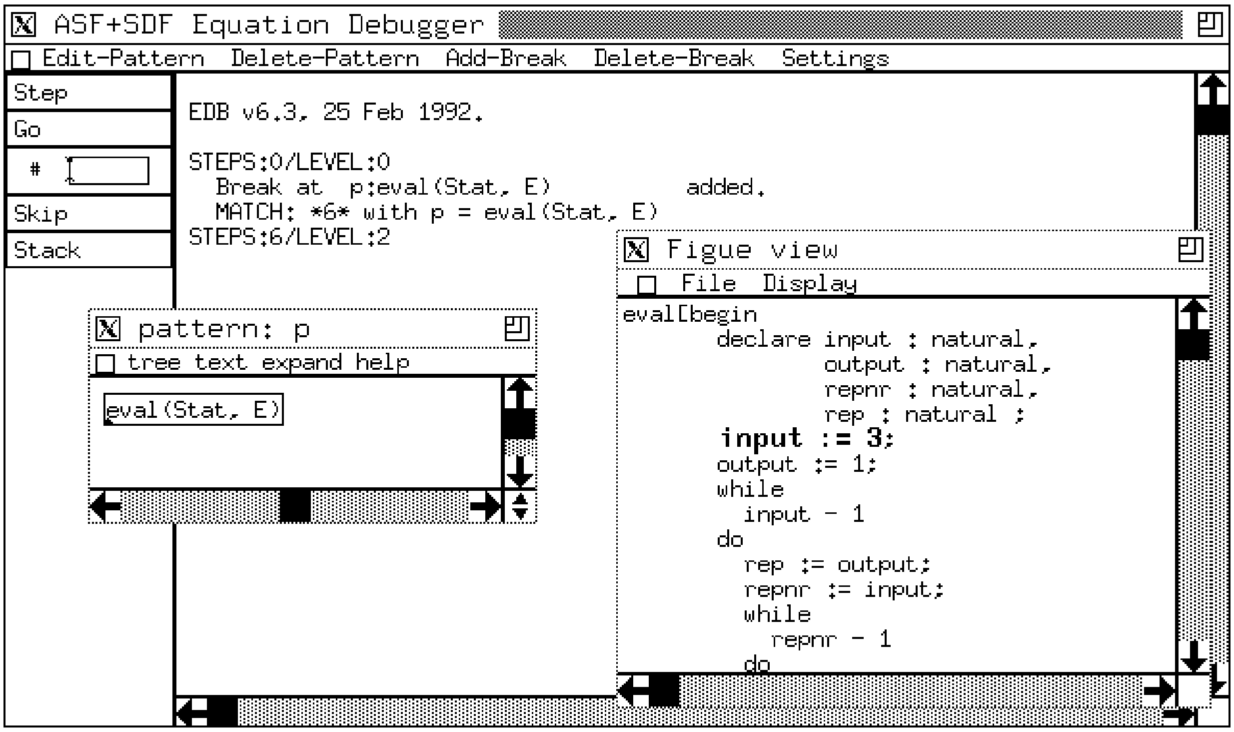

Figure 24: Animation of execution in the ASF+SDF system.

OBSERVATION 5.7 Variable `x` is updated when t matches `update(x, Val, E)`, where `x` is a constant of sort `Id`, `Val` is a variable of sort `Value` and `E` a variable of sort `Env`. (Observe, that replacing the *constant* `x` by a *variable* of sort `Id` would correspond to a breakpoint on the update of *any* variable.) Moreover, the origin u of the symbol `x` in the redex will indicate the place in the program from where the update originated.

Next, we consider the breakpoint on the reference to a variable `x`. A reference to a variable corresponds to an application of the function `lookup`, which takes an identifier and an environment as arguments and returns the current value of that identifier. Therefore:

OBSERVATION 5.8 Variable `x` is referenced when t matches `lookup(x, E)`, where `x` is a constant of sort `Id` and `E` a variable of sort `Env`. Moreover, the origin u of the symbol `x` in the redex will reflect the position in the program where the identifier is being referenced.

Although we have described animation of execution and the definition of breakpoints using a fixed example, our method of achieving these goals is a very generic one. Given an algebraic specification of the execution behavior of a programming language, the task of specifying debugging support for that language is greatly helped by origin tracking. In Section 7, current limitations of our approach as well as future plans regarding generic debuggers are discussed.

5.3 A prototype implementation

Origin tracking has been implemented in the ASF+SDF system [Hen91]. In the present preliminary version, we make use of the Equation Debugger [Tip91] in order to have access to the individual steps of the rewriting process. It is expected that origin tracking will become a more independent component in the future, which only interfaces with the rewriting engine.

Figure 24 contains a window dump of animation of execution, as was described in Section 5.2. Whenever the current redex matches the pattern `eval(Stat, E)`, the origin of the statement is indicated in the initial term. In the example, the first match with the pattern `eval(Stat, E)` occurred after 6 rewrite-steps. According to the origin of the statement subterm in the redex, the corresponding (first) statement is indicated in **boldface**.

Some measurements of the time overhead caused by origin tracking in the preliminary implementation have been performed. In all benchmarks the run-time overhead lies between 10% and 100%, excluding the costs of pre-computing positional information.

6 Related Work

In TRS theory, the notion of *descendant* [Klo91, Chapter 8] (sometimes called *residual* [O'D77, HL79]) is well-known. Descendants are very close to origins (see Section 3.3), but they are more limited, and are only defined for orthogonal (left-linear and non-overlapping) TRSs without conditions. The reason for introducing descendants is of a theoretical nature; they are used to study properties like confluence or normalization, and to find optimal orders for contracting redexes [HL79]. Current research concerning this topic can be found in [Mar92].

Bertot [Ber91b, Ber91a] studies residuals in Term Rewriting Systems 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.

Some ideas of Bertot have been implemented in the language TYPOL [Kah87, Des88], a formalism to specify programming languages, based on *natural semantics*, which in turn is based on the structural operational semantics of Plotkin [Plo81]. A characteristic of natural semantics is that the meaning of a language construct is expressed in terms of the substructures of that construct. In TYPOL, a variable called *Subject* is automatically maintained, having as value the path from the root to the construct processed by the rule currently in action. The TYPOL implementation built on top of the CENTAUR-system [BCD+89], can use this variable *Subject* in implementations of debuggers or error handlers.

Thus, in TYPOL at any moment it is known which language construct is currently processed. However, that is the only information that is known. When type-environments are constructed, no origin information is maintained for the identifiers, types, etc. When having animation in mind, this is especially bad for procedure calls; the body of the procedure would be stored in an environment, and thus in TYPOL the origin information of the body is lost. A specification writer can circumvent this by explicitly storing the value of the variable *Subject* in the entry for the procedure declaration in the type-environment. When evaluating a procedure P, the body B with origin O for procedure P is fetched from the environment. A special language construct *withsubject* allows to change the origin of the new subject to the origin O stored for the procedure body B, after which this body is evaluated. In our approach, this effect can be achieved automatically without any additions to the specification.

Kishon et al. [KHC91] studied a problem related to origin tracking in the framework of denotational semantics. They aim at deriving executable monitors (animators) out of a denotational semantics of a programming language. A *monitoring semantics* is defined as a conservative extension of a language's standard continuation semantics, parameterized with respect to specifications of monitoring activity. Basically, the denotational semantics is extended to call monitor functions just before and just after each continuation. The monitor specification, in turn, defines the actions to be performed. To handle origin-like information, every syntactic category is annotated with information such as its location from the root of the program's syntax tree.

In the context of the PSG system [BS86], a generator for language-specific debuggers was described in [BMS87]. Debuggers are generated from a specification of the denotational semantics of a language and some additional debugging functions. A set of built-in debugging concepts is provided to this end. Most features of traditional source-level debuggers are supported, and as a special feature incomplete programs containing placeholders may be executed. When the flow of control reaches a placeholder, the user may enter the missing parts.

7 Concluding Remarks

7.1 Achievements

Now we can summarize what we have achieved in this paper:

- A precise definition of origin tracking as the *inverse* of the rewrite relation, i.e., if a term t has a subterm t' in the initial term as origin then the latter can be rewritten to t. This definition does not depend on a particular rewrite strategy and only establishes relations which can be derived from the syntactic structure of the rewrite rules.

- Sufficient criteria 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 perspective how origin tracking can be applied in the context of generating interactive language-based environments from formal language definitions. In particular, generic techniques for debugging and error reporting have been discussed.

7.2 Limitations

Our current method for origin tracking has limitations most of which are related to the introduction of new function symbols in the right-hand side of equations and conditions. Let us first have a look at some typical problem cases.

- In the context of translating arithmetic expressions to stack machine code one may encounter an equation of the form

```
trans(plus(E1, E2)) = conc(trans(E1), conc(trans(E2),add))
```

Here, the function `conc` stands for the list constructor for code sequences, and `add` stands for the addition instruction of the stack machine. In this case, only the rule for the propagation of origins of variables applies, and no origins will be associated with `conc`[3] or `add`. Intuitively, the `plus` function symbol on the left-hand side could serve very well as the origin of all these symbols on the right-hand side.

- 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.

- As a last example, consider a function `type` that calculates the type of a declaration as might occur in a typechecker for the C language:

  ```
  type(array(Id,int)) = pointer(int)
  ```

  where `Id` is an arbitrary identifier. In this equation the type "array of base type integer" is converted into the type "pointer to base type integer". Again, no origin will be associated with the type expression on the right-hand side.

All these examples have the flavor of translating terms from one representation to another and they illustrate that *more* origin relations between both sides of equations have to be established than we currently do. But which? We have already started exploring extensions of the definition of origins that overcome these limitations. One promising direction is to distinguish *defined functions* and *constructor functions*. Defined functions are eliminated by applying rewrite rules, until a normal form appears that consists solely of constructors. The definition of origins could now be extended by associating with the constructor functions in the right-hand side of an equation an origin consisting of *all* the constructor functions appearing in the left-hand side.

Other possible extensions are

- when a function symbol on the right-hand side of an equation has no origin, the origin of one (or more) of its children can be associated with it;
- when a subterm on the right-hand side has no origin, it could inherit one from its parents;
- when additional relations should be established, one could indicate this by (manually) adding annotations to the specification.

[3]Strictly speaking, the first occurrence of `conc` on the right-hand side will get the same origin as `trans`. However, in many cases this will be the empty origin.

Each of these approaches would solve the problems in the examples given above, but there will be cases where these schemes would yield *too many* or *inappropriate* origins. The last alternative has, of course, the disadvantage that we leave the domain of automatic origin tracking. In all these cases, the major problem is that we no longer have a clear intuition of the relations which are being established.

7.3 Future Work

There are several directions in which the current results can be extended or applied.

Generic debugging techniques and animation. Although we have a first implementation of the debugging and animation examples presented, more work is needed to generate language-specific debuggers that are smoothly integrated in the ASF+SDF Meta-environment. Some of the issues involved are how to specify the desired animation behavior and how to specify debugger behavior like "show the value of variables `a`, `b` and `c` at each breakpoint".

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. Here, the tracking of origins is only half of the story. The other part is concerned with the specification (or automatic derivation) of the text of error messages and the manner in which the specification of error processing is integrated in the specification formalism.

Translation/preprocessing. An extended notion of origins as discussed above can be used to construct automatically bi-directional mappings between source programs and generated code. An obvious application is in code-level debugging, where links are required between assembly language statements and statements in the source program.

This is of particular importance when debugging highly optimized code. In that case, complex reorderings have been performed on the generated code and we expect that the origin information attached to individual assembly language statements will largely survive such reorderings.

Another application concerns the role of preprocessors in compilation systems. A typical preprocessor, like the one for C, extends the base language with macro definitions, directives for conditional compilation, etc. The output of the preprocessor is a C program in which all these extensions have been replaced by conventional C constructs. Unfortunately, when compiling this C program, error messages cannot be related back to the original program with directives, but only to the intermediate, generated, C program. Using the approach of origin tracking, the generated C constructs will have the original preprocessor directives as origins.

Program slicing. A recently introduced notion in the area of debugging and testing is that of a *dynamic program slice* [KL90, AH90, KSF92]. A dynamic program slice is that part of the program that actually determines the value of a given variable at a given occurrence in a program. Clearly, origins describe—to some extent—a similar notion. We will investigate how these notions are related and 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 typechecking and translation [Meu90]. We will investigate the commonalities and differences between these two forms of information.

Acknowledgements

Comments by Jan Heering have resulted in an improved presentation of the formal definition of origins in Section 2. J.W. Klop suggested several improvements

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).

[Bal69] R.M. Balzer. EXDAMS - extendable debugging and monitoring system. In *Proceedings of the AFIPS SJCC*, volume 34, pages 567–586, 1969.

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 14(2).

[Ber91a] 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).

[Ber91b] 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).

[BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[CFZ82] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I and II. *Theoretical Computer Science*, 17:163–191 and 235–257, 1982.

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

[Deu91] A. van Deursen. An algebraic specification for the static semantics of Pascal. Report CS-R9129, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991. Extended abstract in: *Conference Proceedings of Computing Science in the Netherlands CSN'91*, pages 150-164.

[DJ90] N. Dershowitz and J.-P Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol B.*, pages 243–320. Elsevier Science Publishers, 1990.

[DT86] K.J. Dunlap and B. Tuthill. Debugging with dbx. In *Unix Programmer's Manual version 4.3BSD*, chapter PS1:11. University of California, Berkeley, California, April 1986.

[Han78] D.R. Hanson. Event associations in SNOBOL4 for program debugging. *Software - Practice and Experience*, 8:115–129, 1978.

[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.

[KHC91] A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 338–352, 1991. Appeared as SIGPLAN Notices 26(6).

[KL90] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13:187–195, 1990.

[Kli91] P. Klint. A meta-environment for generating programming environments. In J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 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 published as CWI report CS-R9073, Amsterdam, 1990.

[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. To appear.

[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.

[OCH91] R.A. Olsson, R.H. Crawford, and W.W. Ho. A dataflow approach to event-based debugging. *Software - Practice and Experience*, 21(2):209–229, 1991.

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

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, 1981.

[SP91] R.M. Stallman and R.H. Pesch. *Using GDB, A guide to the GNU Source-Level Debugger.* Free Software Foundation Cygnus Support, 1991. GDB version 4.0.

[Tip91] F. Tip. The equation debugger. Master's thesis, University of Amsterdam, 1991.

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

Contents