Origin Tracking for Higher-Order Term Rewriting Systems

A. van Deursen, T.B. Dinesh

Computer Science/Department of Software Technology

CS-R9425 1994

Origin Tracking for Higher-Order Term Rewriting Systems

Arie van Deursen and T.B. Dinesh
{arie,dinesh}@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

Origin Tracking is a technique which, in the framework of first-order term rewriting systems, establishes relations between each subterm t of a normal form and a set of subterms, the *origins of t*, in the initial term. Origin tracking is based on the notion of residuals. It has been used successfully for the generation of error handlers and debuggers from algebraic specifications of programming languages. Recent experiments with the use of higher-order algebraic specifications for the definition of programming languages revealed a need to extend origin tracking for higher-order term rewriting systems.

In this paper, we discuss how origin information can be maintained for $\beta\eta$ reductions and expansions, during higher-order rewriting. We give a definition of higher-order origin tracking. The suitability of this definition is illustrated with a small, existing specification.

AMS Subject Classification (1991): 68N20, 68Q55, 68Q65.

CR Subject Classification (1991): D.2.5, D.2.6, D.3.4, F.3.2.

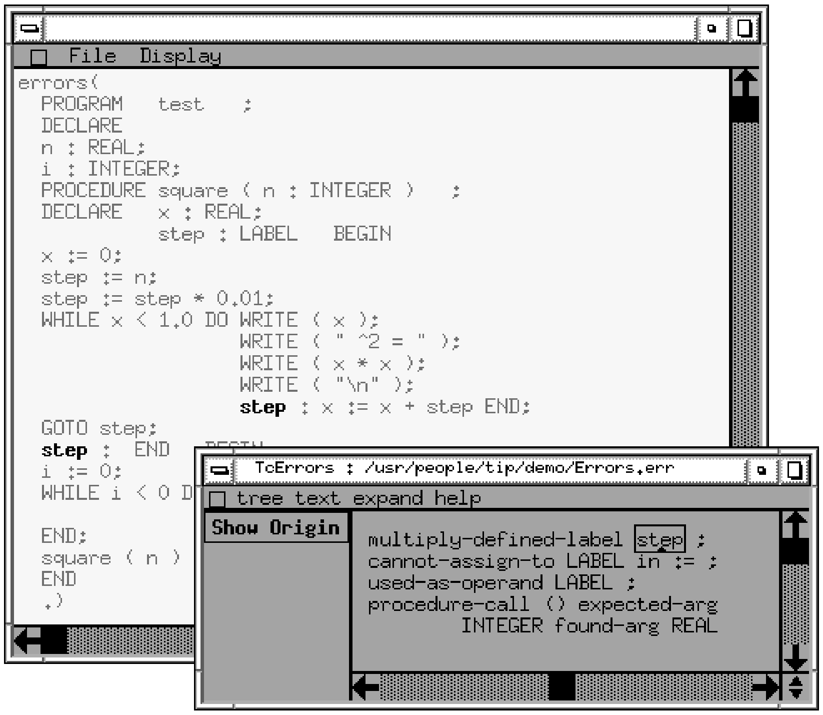
Keywords & Phrases: Algebraic specifications, higher-order rewriting, programming language semantics, programming environments, program generation, origin tracking, program schemes, primitive recursion.

Notes: Partial support has been received from the European Communities, ESPRIT projects 2177: Generation of Interactive Programming Environments II (GIPE II) and 5399: Compiler Generation for Parallel Machines (COMPARE) ; and from the Netherlands Organization for Scientific Research – NWO, project *Incremental Program Generators.* This paper will appear in the proceedings of the Workshop on Higher Order Algebra, Logic and Term Rewriting, Lecture Notes in Computer Science, Springer, 1994.

1. **Origin Tracking**

When algebraic specifications are executed as term rewriting systems (TRSs), computations are performed by reducing an initial term to its result value — its normal form. Often, it is enough just to compute this result value, but in many cases it is useful to have some additional information. For instance, one may wish to know how the initial term influenced the normal form; are there perhaps parts of the initial term that were copied without a change to the result term? Or, if a subterm of the normal form does not *literally* recur in the initial term, is it possible to identify a set of subterms in the initial term which in some sense were *responsible* for its creation?

Trying to capture how intermediate and final terms originate from the initial term is formalized in a notion called *origin tracking* [4, 5, 10]. Origin tracking is based on so-called *residuals*. Residuals have been used successfully in more theoretical work [15, 21, 23], for reasoning about optimal reduction strategies in TRSs.



```
errors(
  PROGRAM   test   ;
  DECLARE
  n : REAL;
  i : INTEGER;
  PROCEDURE square ( n : INTEGER )   ;
  DECLARE   x : REAL;
            step : LABEL   BEGIN
  x := 0;
  step := n;
  step := step * 0.01;
  WHILE x < 1.0 DO WRITE ( x );
                   WRITE ( " ^2 = " );
                   WRITE ( x * x );
                   WRITE ( "\n" );
                   step : x := x + step END;
  GOTO step;
  step :  END
  i := 0;
  WHILE i < 0 D

  END;
  square ( n )
  END
  .)
```

```
multiply-defined-label step ;
cannot-assign-to LABEL in := ;
used-as-operand LABEL ;
procedure-call () expected-arg
        INTEGER found-arg REAL
```

Figure 1: Example of a generated environment using origin tracking.

1.1 Applications

Our motivation for working on origin tracking is its applicability to the automatic generation of tools from algebraic specifications of programming languages. As an example, let us take an algebraic specification of a type checker for some programming language. Assume that this specification can be executed using rewriting, and that the type check function is called *tc*. In order to type check a program P, a term p is constructed representing

P. The term $tc(p)$ is reduced to its normal form, which is a list $[E_1, ..., E_n]$ of error messages. Just carrying out the reduction will only give such a list, whereas when reduced in combination with origin tracking we get additional information. Namely, for each error E_i, a relation to the part in the initial term $tc(p)$, e.g., a statement, an expression or an identifier, responsible for causing E_i is established.

In the ASF+SDF programming environment generator[1] [3, 17] origin tracking has been implemented. This implementation has been used to derive error reporters from algebraic specifications of static semantics of programming languages. As an example, Figure 1 shows a generated editor for a Pascal-like programming language, and an error reporting window which is displayed as a result of the user requesting a type-check of the program. Here the user has selected the error message "multiply-defined-label `step`". By clicking the "Show Origin" button, the user has requested additional information. This action has caused the relevant occurrences of "step", in the original program, to be high-lit.

More details concerning the application of origin tracking to error reporting are given in [11]. Origin tracking can also be used to link source and target code in an algebraically specified compiler, thus facilitating the generation of source-level debuggers. It has also been used to link intermediate steps in an interpreter to the source program (given a specification of an evaluator), thereby aiding the generation of program animators [25].

1.2 Preliminaries: First-Order Rewriting

Before defining origins more rigorously, we borrow some preliminary definitions concerning first-order term rewriting from [19, 15]. Given an alphabet containing *variables* and *function symbols* each equipped with an *arity* (a natural number), a set of *terms* is constructed by considering

- all variables as terms.
- $f(t_1, ..., t_n)$ is a term when $t_1, ..., t_n$ $(n \geq 0)$ are terms and f is an n-ary function symbol.

A term t can be *reduced* to a term t' according to a *rewrite rule* $r : p \to q$ by identifying a context $C[]$ and a subterm s in t such that $t \equiv C[s]$, and by finding a substitution σ such that $s \equiv p^\sigma$. Then $t \equiv C[p^\sigma]$ rewrites to $C[q^\sigma] \equiv t'$ by one *elementary reduction*, written $t \to t'$. We call p^σ the *redex*, and q^σ the *contractum*. The concatenation of reduction steps $t_0 \to t_1 \to ... \to t_n$ is also written $t_0 \to^* t_n$ $(n \geq 0)$.

Subterms are characterized by *occurrences* (paths), which are either equal to [ ] for the entire term or to a sequence of integers (the *branches*) $[n_1, ..., n_m]$ $(m \geq 0)$ representing the access path to the subterm. The occurrence [1, 2]

[1]ASF+SDF is the name of the formalism used to specify programming languages; it originated from combining the Algebraic Specification Formalism ASF and the Syntax Definition Formalism SDF.

denotes the second son of the first son of the root, i.e., for term $f(g(a,b),c)$ it denotes subterm b. The subterm in t at occurrence u is written t/u. Paths are concatenated by the (associative) operator "$\cdot$". If u, v, w are occurrences and $u = v \cdot w$, then v is *above* u, written $v \preceq u$, or $v \prec u$ if $w \neq []$. If neither $u \preceq v$ nor $v \preceq u$ then u and v are *disjoint*, written $u \mid v$. The set of all occurrences in a particular term t is identified by $\mathcal{O}(t)$, which we furthermore partition into a set $\mathcal{O}_{var}(t)$ denoting variable occurrences and a set $\mathcal{O}_{fun}(t)$ denoting function symbol occurrences.

When we wish to identify the redex, rule, and substitution explicitly, we will write $t \xrightarrow{u,\sigma}_r t'$ for the one-step rewrite relation, indicating that rule r is applied at occurrence u in term t under substitution σ.

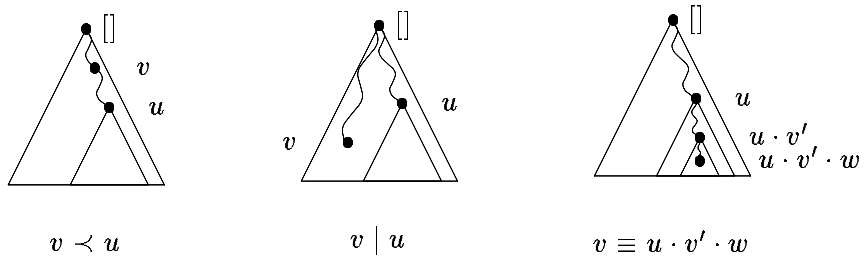


Figure 2: Relative positions of v with respect to contractum position u

1.3 Definition of the Origin Function

We give the definition of origins as described in [10], following the presentation of [4]. Let $t \xrightarrow{u,\sigma}_r t'$, where r is a rule $p \to q$, be an elementary reduction step. With each step we associate a function $org : \mathcal{O}(t') \to \mathcal{P}(\mathcal{O}(t))$ mapping occurrences in t' to sets of occurrences in t. Let $v \in \mathcal{O}(t')$. We define org by distinguishing the following cases (see Figure 2):

- (Context)

  If $v \prec u$ or $v \mid u$ then $org(v) = \{v\}$;

- (Common Variables)

  If $v \equiv u \cdot v' \cdot w$ with $v' \in \mathcal{O}_{var}(q)$ denoting some variable X in the right-hand side, and $w \in \mathcal{O}(X^\sigma)$ an occurrence in the instantiation of that variable, then

  $$org(v) = \{u \cdot v'' \cdot w \mid p/v'' \equiv X\}$$

  Hence, $v'' \in \mathcal{O}_{var}(p)$ denotes an occurrence of X in the left-hand side p.

For the time being, we will assume that $org(v) = \emptyset$ for the remaining case, i.e., where v denotes a function symbol in the right-hand side (see also Section 1.5).

Function *org* covers one-step reductions. It is generalized to a function org^* for a multistep reduction $t_1 \to^* t_n$ $(n \geq 0)$ by considering the origin functions for the individual steps in the complete reduction $t_0 \to t_1 \to \cdots \to t_n$. Let $o_i : \mathcal{O}(t_i) \to \mathcal{P}(\mathcal{O}(t_{i-1}))$ be the origin function associated with rewrite step $t_{i-1} \to t_i$ $(0 < i \leq n)$. Recursively define $org^j : \mathcal{O}(t_j) \to \mathcal{P}(\mathcal{O}(t_0))$ for $0 \leq j \leq n$, and $v \in \mathcal{O}(t_j)$:

- $j = 0$: $org^j(v) = \{v\}$.
- $0 < j \leq n$: $org^j(v) = \{w \mid w \in org^{j-1}(w'),\ w' \in o_j(v)\}$

Then org^* is equal to org^n for multistep reduction $t_0 \to^* t_n$ $(n \geq 0)$.

For orthogonal (left-linear and non-overlapping) TRSs the origin function is the reversal of the well-known notion of *descendant* or *residual* [15]; origins "point backward", whereas residuals indicate what remains of a term during rewriting. In the orthogonal case, the org^* function always yields a set consisting of at most one element.

1.4 Example

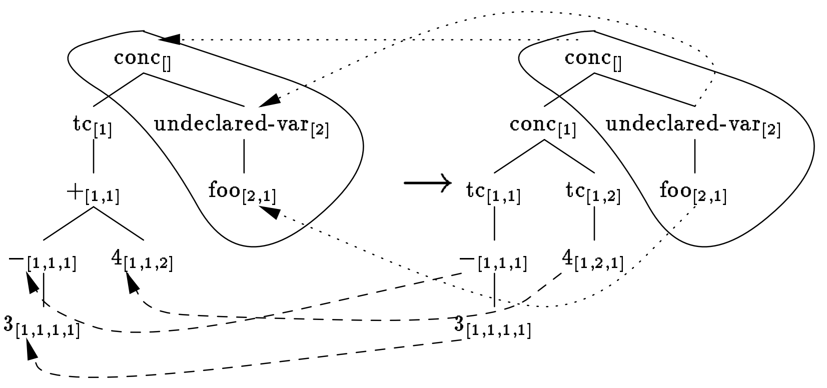
As an example, Figure 3 shows a reduction step of a typical type checker. The redex "tc($E_1 + E_2$)" is contracted at occurrence [1] in the given context. Following the definition of the function *org* just given, origins for nodes within the context are mapped onto themselves. The context positions (on top of or next to the redex) are [], [2], and [2,1], denoting "conc", "undeclared-var", and "foo". For these, we have, $org([]) = \{[]\}$, $org([2]) = \{[2]\}$, and $org([2,1]) = \{[2,1]\}$. Within the contractum, the positions corresponding to function symbol occurrences in the right-hand side obtain the empty set as origin. These positions are [1], [1,1] and [1,2], denoting "conc", "tc", and "tc" respectively, for which we have $org([1]) = org([1,1]) = org([1,2]) = \emptyset$. Finally, the origins within the contractum corresponding to variable occurrences receive an origin to the recurrences of these variables. From variable E_1 we have $org([1,1,1]) = \{[1,1,1]\}$ and $org([1,1,1,1]) = \{[1,1,1,1]\}$, and from E_2 we have $org([1,2,1]) = \{[1,1,2]\}$.

In this example, the origins are sets of at most one element. Sets with more elements can be caused by non-linearity. E.g., rule "and$(X, X) \to X$" will cause X to have origins to both occurrences of X in the left-hand side.

1.5 Discussion

Are the origins in the previous example the ones we were looking for? The origin of "4" to "4" was good, but it is doubtful that the empty set is the best origin for the two occurrences of "tc". Here we summarize some issues we should be aware of when dealing with (extensions of) origins.

Typically, having origins based only on the Common Variables case is insufficient. These will only establish origins for literal recurrences of terms and not for any function symbols introduced. Therefore, in addition to rela-



Rewrite rule:	$\mathrm{tc}(E_1 + E_2) \to \mathrm{conc}(\mathrm{tc}(E_1), \mathrm{tc}(E_2))$
Substitution:	$\{E_1 \mapsto -3,\ E_2 \mapsto 4\}$
Context:	$\mathrm{conc}(\Box, \text{undeclared-var}(\mathrm{foo}))$
Dashed Lines:	Origins for Common Variables
Dotted Lines:	Context Origins.

Figure 3: Origins established for one rewrite step.

tions based on common variables, relations following from function symbol occurrences in left- and right-hand sides of rewrite rules are needed.

Blindly relating any symbol in the right-hand side to all symbols in the left-hand side will not do either, since this would result in origin sets that are too big to give accurate information. On the other hand, it should not be too restrictive. An error message indicating a discrepancy between declaration and use of an identifier should have an origin containing at least two paths: one to the use and one to the declaration. In general, however, we will try to keep the origin sets small.

We will refer to the origins based only on Contexts / Common Variables as *primary origins.* These are clearly necessary and are useful in all applications. Moreover, we will deal with *secondary origins*, where the emphasis is on relations established due to function symbols occurring in left- and right-hand sides of rewrite rules. Proposals for secondary origins may be biased towards particular applications, with emphasis on, e.g., error handling or debugger generation.

1.6 Goal of this Paper

Recent experiments by Heering demonstrated that the use of higher-order algebraic specifications can be advantageous for the definition of programming languages [14]. These experiments, however, also revealed that rapid prototyping of these specifications using higher-order term rewriting would only be of limited use unless some form of origin tracking were available [14, Section 2.2]. Moreover, they suggest that a simple origin scheme based only on the primary origins rule would be inadequate.

This paper addresses these problems. First, we briefly summarize the definitions of higher-order rewriting in Section 2, along with a small example. Next, we present primary origins for the higher-order case in Section 3, and extensions to secondary origins in Section 4. In Sections 5 and 6 we mention related work and draw some conclusions.

2. **Higher-Order Term Rewriting**

For the definition of Higher-Order Term Rewriting Systems (HRSs), we follow [26, 22, 24]. The main difference from the first-order case is that terms in HRSs are constructed according to the simply-typed λ-calculus [7].

2.1 The Simply-Typed λ-Calculus

The set of *type symbols* T consists of *elementary* type symbols from T_0 and of *functional* type symbols $(\alpha \rightarrow \beta)$, where $\alpha, \beta \in T$. We may abbreviate a type $(\alpha_1 \rightarrow (\alpha_2 \rightarrow (\cdots \rightarrow (\alpha_n \rightarrow \beta)\cdots)))$ to $(\alpha_1, \ldots, \alpha_n \rightarrow \beta)$. *Terms* are built using *constants* and *variables*, each of which has an associated type symbol. The type of t is written $\tau(t)$. If x is a variable with $\tau(x) = \alpha$, and t a term with $\tau(t) = \beta$, then the *abstraction* $(\lambda x.t)$ is a term of type $(\alpha \rightarrow \beta)$. If t, t' are terms with $\tau(t) = (\alpha \rightarrow \beta)$ and $\tau(t') = \alpha$, then the *application*[2] $(t\ t')$ is a term of type β. When omitting brackets, application is left-associative.

Occurrences in λ-terms are defined as for the first-order terms, by representing abstraction as a node with 1 son and application as a node with 2 sons. As an example, Figure 4 shows all occurrences in the term (add $((\lambda N.N)$ zero) zero).

All occurrences of x in $(\lambda x.t)$ are said to be *bound*. Non-bound occurrences are *free*. A term is *closed* if it does not contain free variables, *open* otherwise. Bound variables can be renamed according to the rule of α-conversion. A *replacement* of a term t at occurrence u by subterm s is denoted by $t[u \leftarrow s]$. A *substitution* σ is a mapping from variables to terms. Application of a substitution σ to a term t, written t^σ, has the effect that all free occurrences of variables in the domain of σ are replaced by their associated term. Following the *variable convention* [2], bound variables are renamed if necessary.

Let x be a variable, t_1, t_2 terms, and let substitution $\sigma = \{x \mapsto t_2\}$. Then the term $((\lambda x.t_1)\ t_2)$ is a *β-redex* and can be transformed to t_1^σ by β-reduction. A term without β-redex occurrences is said to be in *β-normal form*. All typed λ-terms have a β-normal form, which is unique up to α-conversion. A β-normal form always has the form

[2]We use $@(t, t')$ alternatively, when there is a need to make the application operator explicit, as in Figure 4. We also use $t(t')$ in the context of algebraic specification, as in Figure 5.

$$(\lambda x_1.(\lambda x_2.\cdots(\lambda x_n.\{(\cdots((H\ t_1)\ t_2)\cdots\ t_m)\})\cdots))$$

where $x_1,\ldots,x_n$ are variables, $t_1,\ldots,t_m$ terms in β-normal form, H a constant or a variable, $m,n \geq 0$. We will sometimes write this as $\lambda x_1\cdots x_n.H(t_1,\ldots,t_m)$. In such a term, H is called the *head*, $H(t_1,\ldots,t_m)$ is called the *matrix*, and $\lambda x_1\cdots x_n$ is called the *binder*.

The rule of η-reduction states that terms of the form $\lambda x.(t\ x)$ can be transformed to just t, provided that x does not occur freely in t. Its counterpart is $\overline{\eta}$-expansion: if a head H of a β-normal form $\lambda x_1\cdots x_n.H(t_1,\ldots,t_m)$ is of type $(\alpha_1,\ldots,\alpha_{m+k}\rightarrow\beta)$ $(k>0)$, then clearly as H expects more arguments, we can add these as extra abstractions. The term above can be $\overline{\eta}$-expanded to $\lambda x_1\cdots x_n y.H(t_1,\ldots,t_m,y)$, where y is a fresh variable of type α_{m+1}. Every term has a $\overline{\eta}$-normal form.

Let χ be any of $\{\alpha,\beta,\eta,\overline{\eta}\}$. If t can be transformed to t' by performing a χ-reduction at occurrence u, we write this as $t \triangleright_{\chi,u} t'$, or alternatively as $t' \triangleleft_{\chi,u} t$, where we may omit occurrence u. Repeated χ-reduction is written $t \triangleright^*_\chi t'$. Since $\triangleright^*_\alpha$ is a symmetric relation, we will sometimes write it as $=_\alpha$. The $\beta\overline{\eta}$-normal form of t is indicated by $t\!\downarrow_{\beta\overline{\eta}}$. The relation $t =_{\beta\overline{\eta}} t'$ holds if and only if $t\!\downarrow_{\beta\overline{\eta}} =_\alpha t'\!\downarrow_{\beta\overline{\eta}}$.

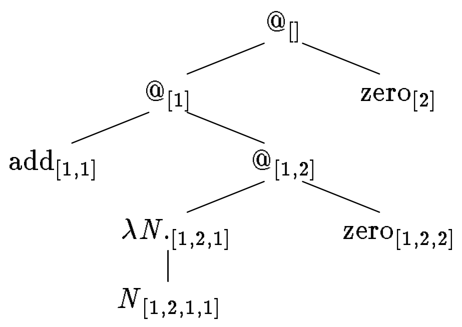


Figure 4: Occurrences in the term "(add $((\lambda N.N)$ zero)) zero".

2.2 Higher-Order Rewrite Steps

If p,q are open simply-typed λ-terms of the same type and in $\beta\overline{\eta}$-normal form, and if every free variable in q also occurs in p, then $p\rightarrow q$ is a (higher-order) rewrite rule. A reduction $t \xrightarrow{u,\sigma}_r t'$, where t,t' are closed λ-terms in $\beta\overline{\eta}$-normal form, σ is a substitution, and u is an occurrence in $\mathcal{O}(t)$ denoting the redex position, is possible if:

- The types of the redex and the left-hand side of the rule are the same: $\tau(t/u) = \tau(p)$
- The instantiated left-hand side is $\beta\overline{\eta}$-equal to the redex: $\{p^\sigma\}\!\downarrow_{\beta\overline{\eta}} \;=_\alpha\; \{t/u\}\!\downarrow_{\overline{\eta}}$

- Replacement of the redex by the instantiated right-hand side followed by $\beta\overline{\eta}$-normalization yields the result t':
  $\{t[u \leftarrow q^\sigma]\}\downarrow_{\beta\overline{\eta}} =_\alpha t'$

Notice the variety of $\{\alpha, \beta, \overline{\eta}\}$-conversions involved in the application of one rule. This turns out to have consequences for the definition of origins. Also note that matching the redex against a left-hand side may yield more than one substitution. For origin tracking purposes, however, we are not concerned with finding matches; we assume that in some way it has been decided to apply a rewrite rule under a given substitution (see also Section 4.3).

2.3 Example
Consider the second-order algebraic specification of a simple type checker shown in Figure 5, which was taken from [14]. The objective of this specification is to replace all simple expressions (identifiers, string or natural constants) by a term "tp(τ)", where τ is the type of that simple expression (see equations [1], [2], and [3]). Next, type correct expressions are reduced to their type (equation [4]). Finally, type correct statements are eliminated (equation [5]). The resulting normal form only contains the incorrect statements.

Take the initial term P_1:

```
program( decls( decl(n,natural),  decls( decl(s,string), emptydecls) ),
         stats( assign(s, plus(id(n),id(n))), emptystats )   )
```

It can be reduced according to equation [1] with, e.g., the substitution[3] σ_1:

$$\begin{array}{lcll} \{ & \mathcal{D} & \mapsto & \lambda \mathit{Decl}.\ \text{decls}(\mathit{Decl}, \text{decls(decl(s,string), emptydecls)}), \\ & \mathcal{S} & \mapsto & \lambda \mathit{Id}.\ \text{stats(assign(s,plus(id(}\mathit{Id}\text{),id(}\mathit{Id}\text{))), emptystats)}, \\ & X & \mapsto & \text{n}, \\ & \tau & \mapsto & \text{natural} \quad \} \end{array}$$

Applying this rule replaces occurrences of "n" by "tp(natural)", which results in a term P_2:

```
program( decls( decl(n,natural),  decls( decl(s,string), emptydecls) ),
         stats( assign(s, plus(id( tp(natural) ),
                               id( tp(natural) ))), emptystats )  )
```

Next, equation [1] can be applied again, this time replacing "s" by "tp(string)", yielding a P_3. Finally, equation [4] can be used to replace the "plus" expression by a representation of its type (natural) resulting in P_4, which is the normal form of P_1.

[3]It is necessary to avoid vacuous abstraction of *Decl* in the assignments of $\mathcal{D}$ [14].

Initially, we are allowed to apply equation [1] on P_1, since under substitution σ_1, the left-hand side of equation [1] produces a new term P_1'', which after two β-reductions (one for $\mathcal{D}$ and one for $\mathcal{S}$) is exactly equal to term P_1.

To construct the result P_2 of this one-step reduction, we first apply σ_1 to the right-hand side of equation [1], producing some term P_2''. Then two more β-reductions transform P_2'' to its β-normal form, which results in the desired P_2. We can summarize this first single-step rewrite as follows:

$$P_1 \lhd_\beta P_1' \lhd_\beta P_1'' \equiv l_1^{\sigma_1} \leadsto r_1^{\sigma_1} \equiv P_2'' \rhd_\beta P_2' \rhd_\beta P_2$$

where $\leadsto$ denotes the replacement of the instantiated left-hand side by the instantiated right-hand side, and l_1 and r_1 are the left and right-hand side of equation [1]. Our definition of origins also follows this "flow" where origins between P_2 and P_1 are defined using elementary origin definitions between the pairs P_2 – P_2', P_2' – P_2'', etc.

3. **Higher-Order Origins**

We define origins for higher-order rewriting by (i) indicating how origins are to be established for $\rhd_\alpha$, $\rhd_\beta$, $\rhd_\eta$, and $\rhd_{\overline{\eta}}$ conversion; then (ii) describing how the inverses $\lhd_\beta$ and $\lhd_{\overline{\eta}}$ can be dealt with; and (iii) explaining how origin relations can be set up between the left- and right-hand side of a rewrite rule. In this section we give a very basic definition, which we refer to as *primary origins*. In the next section we discuss various proposals and heuristics to extend these origins.

We use the following notational conventions. For a term t and variable x, we write $\mathcal{O}_{fvars}(t)$ for all free variable occurrences in t, $\mathcal{O}_{fvars(x)}(t)$ for the occurrences of x in t that are free, and $\mathcal{O}_{bfun}(t)$ for the application, abstraction, or constants as well as the bound variable occurrences in t. Moreover, we abbreviate occurrences of a series of n b-branches as $[b^n]$. For example, for a β-normal form $\lambda x_1 \cdots x_n.H(t_1,\ldots,t_m)$, the path to λx_j is $[1^{j-1}]$ $(1 \le j \le n)$ and the path to t_i is $[1^n]\cdot[1^{m-i}]\cdot[2]$. The left side of Figure 6 shows a term in β normal form, and some path abbreviations.

3.1 Conversions

Let t, t' be terms, $u \in \mathcal{O}(t)$, and let χ be any of $\{\alpha,\beta,\eta,\overline{\eta}\}$. Given $t \rhd_{\chi,u} t'$, we define $org(v)$ for $v \in \mathcal{O}(t')$. First, if $v \mid u$ or $v \prec u$ then $org(v) = \{v\}$. Otherwise,

- $\chi = \alpha$:

  α-Conversion does not change the term structure, so we simply have $org(v) = \{v\}$.

- $\chi = \beta$:

sorts: PROG DECLS DECL STAT STATS ID TYPE EXP ...

functions:

program	: DECLS, STATS	$\rightarrow$ PROG
decls	: DECL, DECLS	$\rightarrow$ DECLS
emptydecls	:	$\rightarrow$ DECLS
decl	: ID, TYPE	$\rightarrow$ DECL
natural	:	$\rightarrow$ TYPE
string	:	$\rightarrow$ TYPE
stats	: STAT, STATS	$\rightarrow$ STATS
emptystats	:	$\rightarrow$ STATS
assign	: ID, EXP	$\rightarrow$ STAT
plus	: EXP, EXP	$\rightarrow$ EXP
id	: ID	$\rightarrow$ EXP
nat	: NAT	$\rightarrow$ EXP
str	: STRING	$\rightarrow$ EXP
...		
tp	: TYPE	$\rightarrow$ ID

variables:

$\mathcal{D}$	: DECL $\rightarrow$ DECLS	X	: ID
τ	: TYPE	$\mathcal{S}$	: ID $\rightarrow$ STATS
S	: STATS	N	: NAT
R	: STRING		

equations:

[1] $\mathrm{program}(\mathcal{D}(\mathrm{decl}(X,\tau)), \mathcal{S}(X))$
$= \mathrm{program}(\mathcal{D}(\mathrm{decl}(X,\tau)), \mathcal{S}(\mathrm{tp}(\tau)))$

[2] $\mathrm{nat}(N) = \mathrm{id}(\mathrm{tp}(\mathrm{natural}))$

[3] $\mathrm{str}(R) \ = \mathrm{id}(\mathrm{tp}(\mathrm{string}))$

[4] $\mathrm{plus}(\mathrm{id}(\mathrm{tp}(\mathrm{natural})), \mathrm{id}(\mathrm{tp}(\mathrm{natural}))) = \mathrm{id}(\mathrm{tp}(\mathrm{natural}))$

[5] $\mathrm{stats}(\mathrm{assign}(\mathrm{tp}(\tau), \mathrm{id}(\mathrm{tp}(\tau))), S) = S$

Figure 5: Part of the static semantics specification

Since t/u is a β-redex, we have $t/u \equiv ((\lambda x.t_1)\ t_2)$. Note that the path to t_1 is $[1,1]$, and to t_2 is $[2]$. Now let $w_1 \in \mathcal{O}(t_1), w_2 \in \mathcal{O}(t_2)$. We distinguish two cases:

1. $v \equiv u \cdot w_1$: Then $org(v) = \{u \cdot [1,1] \cdot w_1\}$.
2. $v \equiv u \cdot w_1 \cdot w_2$, and $w_2 \succ [\,]$: then $org(v) = \{u \cdot [2] \cdot w_2\}$.
   The condition $w_2 \succ [\,]$ avoids overlap with the former case.

Thus, origins in the body t_1 "remain the same"; origins for the top node of an instantiated variable have an origin to their corresponding variable position in the body t_1, which is indicated by the dashed lines in Figure 7; and origins to non-top nodes of an instantiated variable have an origin to their position in the actual parameter t_2, which is

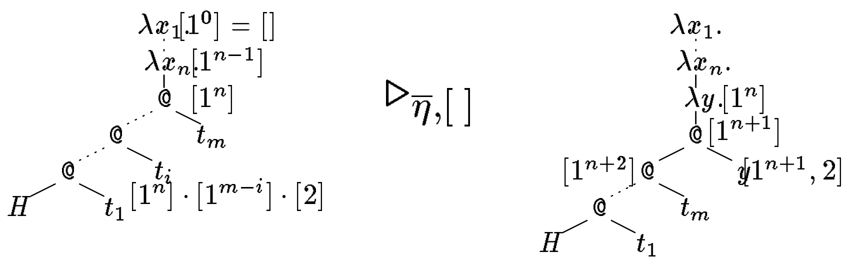


Figure 6: $\overline{\eta}$-Expansion.

indicated by the dotted lines.

- $\chi = \eta$:

  In η-reduction one λ is eliminated. Since t/u is an η-redex, we can assume $t/u = \lambda x.(t_1\ x)$. Realizing that the path to t_1 is $[1,1]$, we simply have: $org(u \cdot v') = \{u \cdot [1,1] \cdot v'\}$.

- $\chi = \overline{\eta}$:

  In η-expansion, an extra λ is added. The origins of the old parts point to the same old parts, while the origin of the new λ is the empty set:

  Since t/u is an $\overline{\eta}$-redex, we have $t/u = \lambda x_1 \cdots x_n.H(t_1, \ldots, t_m)$. We distinguish three cases for $v = u \cdot v'$:

  1. For $v' \preceq [1^{n-1}]$, $org(u \cdot v') = \{u \cdot v'\}$.
  2. For $v' \in \{[1^n], [1^{n+1}], [1^{n+1}, 2]\}$, $org(u \cdot v') = \emptyset$.
     Figure 6 shows, using tree representations, the occurrences $[1^n]$, $[1^{n+1}]$ and $[1^{n+1}, 2]$ introduced by $\overline{\eta}$-expansion.
  3. For $v' \succeq [1^{n+2}]$, $org(u \cdot [1^{n+2}] \cdot v'') = \{u \cdot [1^{n+1}] \cdot v''\}$ where $v' \equiv [1^{n+2}] \cdot v''$.

Assume that we have an origin function O mapping occurrences of t' to sets of occurrences in t. Then O is said to be *unitary* if its result values are always sets containing exactly one element, and *unique* if they contain at most one element. If an occurrence can have the empty set as origin, we say O is *forgetful*. If several occurrences in t' have an origin to the same node in t, we may refer to O as *many-to-one*, while its counterpart, where an origin set can contain more than one path, is called *one-to-many*. Finally, if for every $v \in \mathcal{O}(t')$ we have $O(v) = \{v\}$, then we say O is *identical*.

Thus, the origin function is identical for α, is unitary for η, is forgetful for $\overline{\eta}$ and finally, is unitary and many-to-one for β. None of these is one-to-many, which is fortunate, since in Section 1.5 we concluded that it was advisable to keep the origin sets small.

3.2 Equality modulo $\beta\overline{\eta}$-conversions

In Section 2.3 we discussed, reversed β and $\overline{\eta}$-reductions that need to take place. The origin functions for $\rhd_{\{\alpha,\beta,\eta,\overline{\eta}\}}$ defined in the previous section

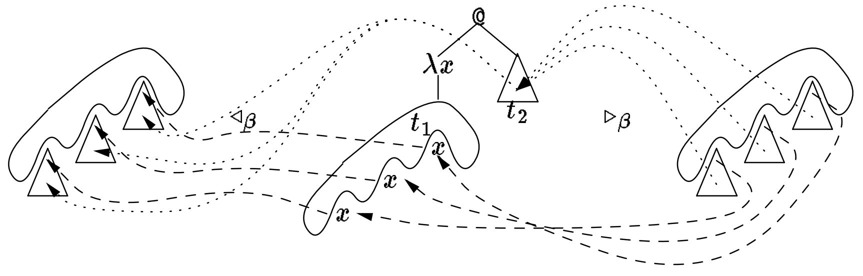


Figure 7: β-reduction in both directions.

can easily be inverted, thus yielding origin functions for $\triangleleft_{\{\alpha,\beta,\eta,\overline{\eta}\}}$. Note that, from an origin tracking point of view, the inverse of η-reduction is $\overline{\eta}$-expansion.

Since the origin function for α-conversion is identical, performing several α-conversions in one direction or another does not affect the origins. This is not the case for $\overline{\eta}$ or β reduction. Since β-reduction is many-to-one, its inverse must be one-to-many. As can be seen from Figure 7, this may lead to a growth of the origin sets. Consider a reduction $t \triangleleft_\beta t' \triangleright_\beta t''$, where $t' = ((\lambda x.t_1)\ t_2)$, and $t, t'' = t_1^{\{x \mapsto t_2\}}$, then the origins from t'' to t' will cause all instantiated occurrences of x to be related to the same t_2 in t'; the origins of t' to t in turn will link this t_2 to all instantiated occurrences of x in t. Thus, transitively, one occurrence of t_2 in t'' has origins to *all* occurrences of t_2 in t. This is illustrated by the dotted lines of Figure 7. Note that the definition of the origin function for the β reduction (case 1), relates the *top node* of t_2, via the xs occurring in t_1 to its position in t (dashed lines of Figure 7).

Since the origins for $\overline{\eta}$ conversions are unique this problem does not arise for $\overline{\eta}$ conversions. However, the $\triangleright_{\overline{\eta}}$ are forgetful, so checking for $\overline{\eta}$-equality may result in loss of some origin information (in particular in the binders).

3.3 Left- and Right-Hand Sides

We define the relations between the instantiated left and right-hand side of a rewrite rule, where we assume that these are instantiated but not yet $\beta\overline{\eta}$-normalized. We closely follow the first-order case defined in Section 1.3.

Let $p \to q$ be a rewrite rule, and σ a substitution. The function $org : \mathcal{O}(q^\sigma) \to \mathcal{P}(\mathcal{O}(p^\sigma))$, for a path $v \in \mathcal{O}(q^\sigma)$, is defined as follows:

- (Common Free Variables)

  If $v \equiv v' \cdot w$ with $v' \in \mathcal{O}_{fvars}(q)$ denoting some variable X in the right-hand side, and $w \in \mathcal{O}(X^\sigma)$ an occurrence in the instantiation of that variable. Then:

$$org(v) = \{v'' \cdot w \mid q/v' \equiv p/v'',\ v'' \in \mathcal{O}_{fvars(X)}(p)\}$$

Thus, v'' denotes an occurrence of X in left-hand side p.

- (Function Symbols)

  If $v \in \mathcal{O}_{bfun}(q)$, then $org(v) = \emptyset$.

This is obviously a forgetful definition, but this situation is improved in Section 4. As in the first-order case, it is also possibly one-to-many (in the case of non-left-linearity).

Note that the common free variables case results in the same origins as in the common variables case of Section 1.3, when the specification does not use the higher-order features. The Context case will be dealt with in the next section.

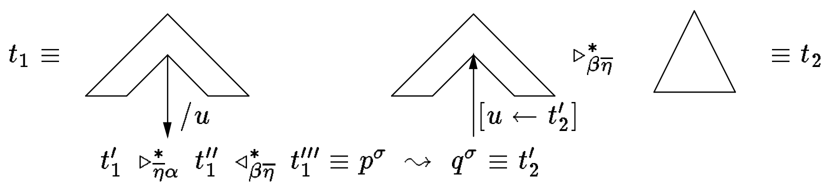

$t_1 \equiv$
$/u$
$[u \leftarrow t'_2]$
$\rhd^*_{\beta\overline{\eta}}$
$\equiv t_2$
$t'_1 \ \rhd^*_{\overline{\eta}\alpha} \ t''_1 \ \lhd^*_{\beta\overline{\eta}} \ t'''_1 \equiv p^\sigma \ \rightsquigarrow \ q^\sigma \equiv t'_2$

Figure 8: All conversions for one reduction step $t_1 \to t_2$, applying rule $p \to q$ at occurrence u in t_1 under substitution σ.

3.4 Rewrite Steps

Knowing how to both establish origins for α-, β-, and $\overline{\eta}$-conversions in either direction and to set up origins between the instantiated left- and right-hand side, we can obtain the origins for one complete reduction step $t_1 \to t_2$. Figure 8 summarizes the work to be done for one reduction, following as described in Section 2.2.

Note that in general the situation is slightly more complicated than in the example of Section 2.3

$$P_1 \lhd_\beta P'_1 \lhd_\beta P''_1 \equiv l_1^{\sigma_1} \rightsquigarrow r_1^{\sigma_1} \equiv P''_2 \rhd_\beta P'_2 \rhd_\beta P_2$$

where the rewrite rule is applied at the root of P_1 which has the effect that Figure 8 can be reduced to just "one level": The context is empty ($u = []$), and consequently the term t/u is already a $\overline{\eta}$-normal form, hence the result need not be put back into the context (in the figure, $[\,[] \leftarrow t'_2]$ is just equal to t'_2).

3.5 Example

Consider reduction $P_1 \to P_2$ as presented in Section 2.3. Most occurrences in P_2 have their intuitive origin; mainly because they also occur in bodies of the instantiations of $\mathcal{D}$ and $\mathcal{S}$ in substitution σ_1. However, some origins are lost; in particular for nodes occurring in the right-hand side of rule [1]. Thus, symbols "program", "decl" (for the declaration of n), and "tp" do not have an origin. Moreover, rule [1] is non-linear in X, and therefore the X-occurrence in the declaration at the right-hand side has an origin to the occurrence in the statement as well as in the declaration. Thus, the single n in P_2 has origins to all n occurrences in P_1 (this does not seem intuitive). All occurrences of "natural" in P_2 have their origin to the declaration it came from (seems reasonable).

Now consider the entire reduction $P_1 \rightarrow^* P_4$, where normal form P_4 is:

```
program( decls( decl(n,natural),  decls( decl(s,string), emptydecls) ),
         stats( assign( tp(string),
                  plus(id( tp(natural) ),id( tp(natural) ))),
                emptystats )  )
```

In this case, more origins are lost. In particular, the two "decl" nodes have an empty origin, and the reduction according to rule [4] did not establish any origins, so "tp(natural)" does not have any origins.

4. **Extensions**

The origins in the previous example were nice, but still not sufficient for using them in practice. In this section we present some extensions of the origin function. Some of these extensions are of a heuristic nature, based on frequently occurring forms of (higher-order) rewrite rules.

4.1 Extended Contexts

Taking a close look at equation [1] of Figure 5, we see that its intention is to identify some context "program(...)" in which a certain term (the identifier denoted by X) is to be replaced by another term (in this case tp(τ)). This context is exactly the same in the left- and right-hand side of the rewrite rule.

It seems reasonable to extend the notion of a context to cover such similarities within rewrite rules as well. Considering a rewrite rule $p \rightarrow q$, we can look for a (possibly empty) *common context* C and *holes* (terms) $h_1, \ldots, h_m$ and $h'_1 \ldots h'_m$ ($m \geq 0$) such that $p =_\alpha C[h_1, \ldots, h_m]$ and $q =_\alpha C[h'_1, \ldots, h'_m]$, where $h_j \neq_\alpha h'_j$ for all $1 \leq j \leq m$. We are actually looking for the biggest of such contexts which contain the smallest possible number of holes where none of the holes h_j, h'_j ($1 \leq j \leq m$) start with a non-empty context $\overline{C}$ such that $h_j =_\alpha \overline{C}[\overline{h}_1, \ldots \overline{h}_n]$ and $h'_j =_\alpha \overline{C}[\overline{h'}_1, \ldots \overline{h'}_n]$. As an example, equation [1] of Figure 5 has a common context C = "program($\mathcal{D}$(decl(X,τ)), $\mathcal{S}(\Box)$)", where the hole h_1 at the left is equal to "X", and h'_1 at the right to "tp(τ)".

For every node in this extended context, the origin should point only to its corresponding occurrence in that same context at the left-hand side. Note that, as a consequence, the common variables case should *not* apply to variables occurring in the common context. For example, in equation [1], the origin of X at the right will only point to its counterpart under the "decl" at the left and not to the X in the statements. Moreover, when trying to find origins for a node in a hole h'_j, it seems reasonable to focus on origins that can be found within the corresponding hole h_j. Only if it is impossible to find origins there, an origin can be looked for in the rest of the left-hand side.

There is, however, a minor catch in this. If two consecutive holes h_j and h_{j+1} are only separated by an application in the context C, i.e. they actually occur as $@(h_j, h_{j+1})$ at the left and as $@(h'_j, h'_{j+1})$ at the right, then it is more natural to regard these two as one hole ($H = @(h_j, h_{j+1})$ instead of h_j and h_{j+1}). As an example, equation [2] in applicative form reads as @(nat,N) = @(id, @(tp, natural)). It would be counter-intuitive to regard the top-application as a common context $@(\Box, \Box)$ with two holes: h_1 = nat, h'_1 = id, and $h_2 = N$, h'_2 = @(tp, natural).

Note that this new extended context case would be useful in the first-order case as well.

4.2 Origins for Constants
Let $p \equiv C[h_1, \ldots, h_m] \rightarrow C[h'_1, \ldots, h'_m] \equiv q$ be a rewrite rule with the common context C and m ($m \geq 0$) holes. We define origins for constants occurring in the h'_j ($1 \leq j \leq m$) according to the following three cases:

1. Head-to-Head

   The origin for the occurrence of the head symbol of a hole h'_j at the right is the occurrence of the head symbol of that same hole h_j at the left. For example, the "tp" symbol in equation [1] is linked to the occurrence of X in the statements at the left. This head-to-head rule corresponds to the "redex-contractum" rule of the first-order origins as described in [10]. Note that if the head symbol at the right is a free variable, the common variables case is applicable as well. This can, in general, have the effect that the origin set for the head symbols consist of more than one path.

2. Common Subterms.

   If a term s is a subterm of both h'_j and h_j, then these occurrences of s are related. For example, the subterm "tp(natural)" at the right of equation [4] (Figure 5) is related to both occurrences of "tp(natural)" at the left. Note that these common subterms are identified in the *un-instantiated* left- and right-hand side. This rule can in some cases lead to seemingly wild connections, but has already proven its usefulness for the first-order case [10, 11]. The common subterms behave slightly different in the higher-order case, due to the applicative form of the λ-terms. In the first-order case, function symbols were only related if all arguments were identical at the left and right. In the higher-order case, function symbols are constants. Each constant F in h'_j is related to all occurrences of F in h_j. This effect is similar to the *tokenization* discussed in [11].

If for a subterm s of h'_j no occurrences of s can be found in h_j, then the entire left-hand side p can be used to find a common subterm occurrence of s.

3. Any to All.

   If after application of the head-to-head and common subterms case there are still constants in h'_j with an empty origin, the set of all constant occurrences at the left-hole h_j is defined as its origin set. For example, in equation [2], the subterms "tp(natural)" and "natural" relate to both "nat(N)" and "N".

4.3 Abstraction and Concretization Degree

Let us end our discussion with an interesting observation. Recall from Section 3.2 that $\lhd_\beta$ conversions are one-to-many. Assume that $t' \lhd_\beta t$ with $t \equiv ((\lambda x.t_1)\ t_2)$. It would be useful to call the number of free occurrences of x in t_1 the *abstraction degree* of $\lambda x.t_1$, and the number of occurrences of term t_2 in t_1 the *concretization degree*. When trying to find a matching substitution σ in order to apply a rewrite rule, freedom exists concerning the abstraction and concretization degree. For example, if σ assigns F a value T with abstraction degree $N > 0$ and concretization degree $M \geq 0$, then an alternative match σ' can also be possible which assigns F a term T' with abstraction degree $N - 1$ and concretization degree $M + 1$. The problems with $\lhd_\beta$ are minimized if matches with abstraction degree 1 are preferred over those with a higher abstraction degree.

In practice, however, such a preference may be somewhat problematic. Firstly, a substitution with a lower degree of abstraction may not even exist. Secondly, the repeated application of a substitution with abstraction degree 1 need not yield the same result as a single application with a higher abstraction degree. Finally, repeated applications may be more expensive in terms of run time behavior, than a single application with a high abstraction degree.

4.4 Example

With these extensions, suitable origins for the example in Section 2.3 are obtained. We assume that equation [1] is applied with substitutions of abstraction degree 1 only. The extended contexts assure that "program" and "decl" are linked. Moreover, the effect of linking variables in contexts only to the same occurrence in the context, guarantees that the n and s in the declaration have the proper unitary origin. Furthermore, relating heads of holes guarantees that the "tp" nodes get the right origin to the variable they were substituted for. Likewise, the application of equation [4] results in "plus" as the origin of "tp". Finally, common subterms results in "tp(natural)" to be linked to both occurrences of "tp(natural)" in the "plus" expression (equation [4]).

The example given here is only part of the specification discussed in [14].

The origins with extensions create the proper relations for the full specification as well.

5. Related Work

The current document is part of a series of papers studying origins and their applications to the automatic generation of parts of compilers or programming environments – in particular error handlers, symbolic debuggers, and animators. The extensions to primary origins studied in [10] establishes relations between *common subterms* in left and right-hand side of rewrite rules, as well as a link between the top-node of the *redex* and the *contractum*. Moreover, origins are defined for *conditional* rewrite rules. Several issues related to the efficient implementation of origin tracking in the ASF+SDF Meta-Environment [17] are discussed in [10]. The applicability of origins in practice, using a specification of the semantics of a subset of Pascal, is studied by Dinesh and Tip where the static semantics and generated error handler is covered in [11] and the dynamic semantics and generated animator is described in [25]. In order to improve origin tracking for *syntax-directed* specifications (typically translators or type checkers), an extension for *primitive recursive schemes* is proposed in [9]. An origin-like relation, called *dynamic-dependence* relation is studied by Field and Tip [12]. They show that the dependence tracking technique is useful in the context of program slicing.

The study of origins was pioneered by Bertot [4, 5], who was concerned with origins in natural semantics, (orthogonal) term rewriting, and the (untyped) λ-calculus. He describes a language for the definition and representation of origins. In his setting, origins are unitary (consisting of at most one path). Secondary origins are represented by *marking functions*. This work was done in the framework of the CENTAUR system [6]. In particular, the specification language TYPOL [16] has been extended with *subject tracking* [8].

Closely related to origins are *residual maps*, *descendants*, or *labelings* [20, 15, 21, 13], which are used to study reduction strategies. Residuals indicate which redexes survive if a particular redex is contracted. One can think of this as giving interesting parts in the initial term a particular color, and then looking how this color survives during reduction. An interesting combination of origins and labeling systems is presented by Bertot [5] where he investigates how origins for TRSs can be used to simulate labeling systems for the λ-calculus. The labels of [20] suggest that alternative representations for origins containing more structure than the (simple) sets of paths could be fruitful.

Nipkow's definition of higher-order TRSs requires the rewrite rules to satisfy several syntactic constraints [22]. We have discussed origins using the more liberal setting of Wolfram [26]. Obviously, the same origins can be established for Nipkow's HRSs. The nicer matching behavior of Nipkow's HRSs will probably have a favorable effect on the origins. The mapping between Nipkow's HRSs and Klop's combinatory reduction systems (CRSs) [18] as described in [24] can be the basis for a definition of origins for CRSs.

Another issue is the study of origins as transformations on HRSs. Tip has conducted such experiments for the first-order case. For the higher-order case, it may be useful to use specifications of the λ-calculus with explicit substitutions as in [1].

6. Conclusions

Origin tracking for higher-order specifications is considerably more difficult than establishing origin relations for the first-order case. Various conversions to be performed, both as reductions and as expansions, have to be taken into account. Nevertheless, we have found a satisfactory origin scheme, which is applicable to arbitrary higher-order term rewriting systems

There is, however, still some future work to do. The most important thing is to gain experience with these origins. More specifications of realistic problems and their applicability for origin tracking should be studied.

Finally, after having seen many variants of origin tracking, it may be worthwhile to investigate the possibility of generalizing to some kind of origin *scheme*. This may clarify and ease future discussions of further extensions of origin tracking.

Acknowledgments This paper would not have been possible without Jan Heering's support and advice. Comments of Jan Heering, Femke van Raamsdonk, Susan Üsküdarlı, Machteld Vonk and the reviewers have helped improve the presentation.

References

1. M. Abadi, L. Cardelli, P.-L. Currien, and J.-J. Lévy. Explicit substitutions. In *Proceedings of the 17th conference on Principles of Programming Languages*, pages 31–46, 1990.

2. H.P. Barendregt. *The Lambda Calculus; its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathatematics.* North-Holland, 1984.

3. 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.

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

5. Y. Bertot. Origin functions in lambda-calculus and term rewriting systems. In J.-C. Raoult, editor, *Proceedings of the 17th Colloquium on Trees in Algebra and Programming (CAAP '92)*, volume 581 of *LNCS.* Springer-Verlag, 1992.

6. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the ACM SIG-*

SOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 24(2).

7. A. Church. A formulation of a Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

8. Th. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, 1988.

9. A. van Deursen. Origin tracking in primitive recursive schemes. In H.A. Wijshoff, editor, *Conference Proceedings Computing Science in the Netherlands CSN'93*, pages 132–143, 1993.

10. A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special Issue on Automatic Programming.

11. T.B. Dinesh. Type checking revisited: Modular error handling. In *Proceedings of the Workshop on Semantics of Specification Languages*, Utrecht, 1993. Springer-Verlag, LNCS. To Appear.

12. J. H. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. Technical report, Centrum voor Wiskunde en Informatica (CWI), 1994. To appear.

13. J.H. Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems.* PhD thesis, Cornell University, 1991.

14. J. Heering. Second-order algebraic specification of static semantics. Technical Report CS-R9254, Centrum voor Wiskunde en Informatica (CWI), 1992. Extented version to appear, 1994.

15. G. Huet and J.-J. Lévy. 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*, pages 395–443. MIT Press, 1991.

16. 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 *LNCS*, pages 22–39. Springer-Verlag, 1987.

17. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

18. J.W. Klop. *Combinatory Reduction Systems.* Number 127 in Mathematical Center Tracts. Mathematisch Centrum, Amsterdam, 1980.

19. J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, pages 1–116. Oxford University Press, 1992.

20. J.-J. Lévy. An algebraic interpretation of the $\lambda\beta$K-calculus and a la-

belled λ-calculus. In C. Böhm, editor, *λ-Calculus and Computer Science Theory*, number 37 in LNCS. Springer-Verlag, 1975.

21. L. Maranget. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proceedings of the Eighteenth conference on Principles of Programming Languages POPL '91*, pages 225–269, 1991.
22. T. Nipkow. Higher-order critical pairs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349. IEEE Computer Society Press, 1991.
23. V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, Amsterdam, March 1994.
24. V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems, 1994. This proceedings.
25. F. Tip. Animators for generated programming environments. In P. Fritzson, editor, *Proceedings of the First International Workshop on Automated and Algorithmic Debugging AADEBUG'93*, LNCS. Springer-Verlag, 1993.
26. D.A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993.