Origin Tracking for Higher-Order Term Rewriting Systems

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

CWI – P.O. Box 4079, 1009 AB 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 to higher-order term rewriting systems. This extension is discussed, covering a definition and some alternatives, as well as an assessment with respect to existing specifications.

1 Origin Tracking

When algebraic specifications are being 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 can be useful to have some more 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, can it be possible to identify a set of subterms in the initial term which in some sense were *responsible* for its the creation?

Trying to capture how intermediate and final terms originate from the initial term is formalized in a notion called *origin tracking* [Ber91, Ber92, DKT93]. Origin tracking is based on so-called *residuals*, which have been used successfully in more theoretically oriented papers [HL91, Mar91] for reasoning about optimal reduction strategies in TRSs.

1.1 Applications

Our motivation to work on origin tracking was that we needed it for the automatic generation of tools from algebraic specifications of programming languages. As an example,

*Supported by the European Communities, ESPRIT project 2177: Generation of Interactive Programming Environments II — GIPE II; and by the Netherlands Organization for Scientific Research – NWO, project *Incremental Program Generators*.

†Supported by the European Communities, ESPRIT project 5399: Compiler Generation for Parallel Machines - COMPARE

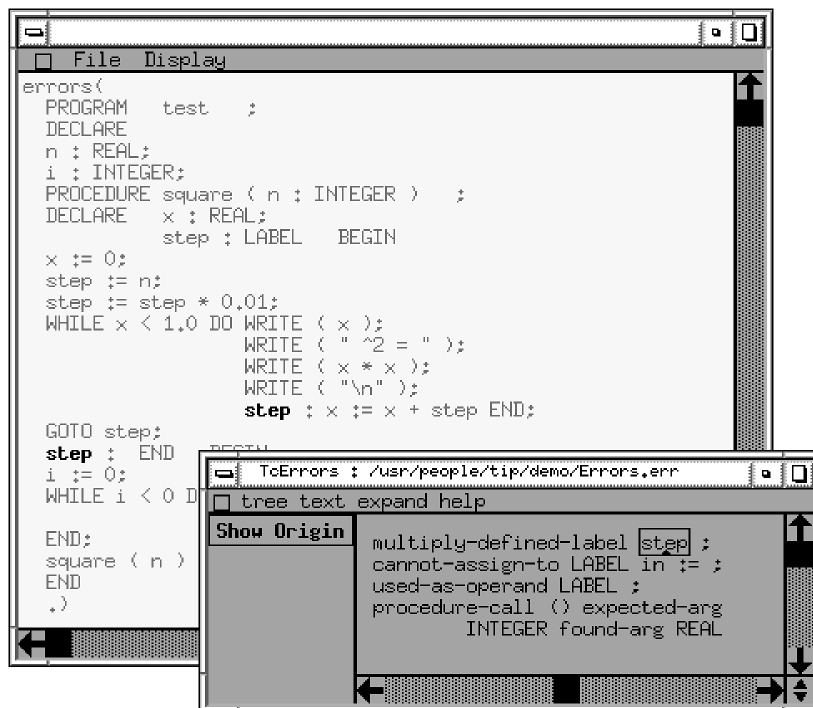


Figure 1: Example of a generated environment using origin tracking.

let us take an algebraic specification of a type checker for some programming language. Assume that the specification can be executed using rewriting, and that the type-check function is called tc. In order to typecheck a program P, a term p is constructed representing P and the term $tc(p)$ is reduced to its normal form, which we assume to be a list $[E_1, ..., E_n]$ of error messages. Just carrying out the reduction will only give this list. Doing this in combination with origin tracking will give additional information: For each error E_i it is indicated which statement, expression, identifier, or other part of the initial term $tc(p)$ was responsible for the generation of E_i.

In the ASF+SDF programming environment generator[1] [BHK89, Kli93] origin tracking has been implemented. It is used to derive error reporters from algebraic specifications of the static semantics of programming languages. In Figure 1 a generated editor for a simple programming language is shown. The programmer asked for a typecheck which resulted in a list of four error messages. He asked for more information concerning the error message "multiply-defined-label `"step"`" by clicking the "Show Origin" button. This caused the relevant occurrences of "step" to be highlighted in the original program.

More details concerning the application of origin tracking to error reporting are given in [Din93]. 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, or it can be used to link intermediate steps in an interpreter to the program executed given a specification of an evaluator, thus aiding in the generation of program animators [Tip93].

[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.

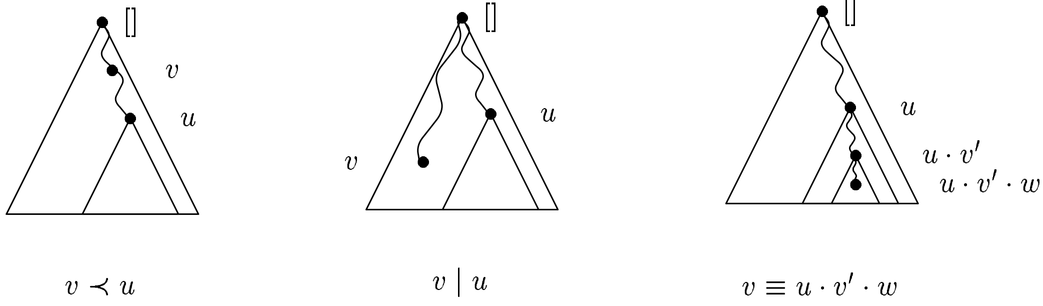


Figure 2: Relative positions of v with respect to contractum position u

1.2 Preliminaries: First-Order Rewriting

Before defining origins more rigorously, we borrow some preliminary definitions concerning first-order term rewriting from [Klo91, HL91]. Given an alphabet containing *variables* and *function symbols* each equipped with an *arity* (natural number), a set of *terms* is constructed: all variables are terms, and if f is an n-ary function symbol and $t_1, ..., t_n$ $(n \geq 0)$ are terms, then $f(t_1, ..., t_n)$ is a term.

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 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*. For the concatenation of reduction steps $t_0 \to t_1 \to ... \to t_n$ we also write $t_0 \to^* t_n$ $(n \geq 0)$.

Subterms are characterized by *occurrences* (paths), which are either equal to [ ] for the entire term or to a sequence of integers (the *branches*) $[n_1, ..., n_m]$ representing the access path to the subterm. E.g., occurrence [1, 2] 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 $u \preceq v$ 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 $\mathcal{O}_{var}(t)$ of occurrences denoting variables and a set $\mathcal{O}_{fun}(t)$ denoting function symbols.

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.

1.3 Definition of the Origin Function

We give the definition of origins as described in [DKT93], following the presentation of [Ber91]. 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 three 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 $t_1 \rightarrow^* t_n$ $(n \geq 0)$ by considering the origin functions for the individual steps in the complete reduction $t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_n$. Let $o_i : \mathcal{O}(t_i) \rightarrow \mathcal{P}(\mathcal{O}(t_{i-1}))$ be the origin function associated with rewrite step $t_{i-1} \rightarrow t_i$ $(0 < i \leq n)$. Recursively define $org^j : \mathcal{O}(t_j) \rightarrow \mathcal{P}(\mathcal{O}(t_0))$ 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 $t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_n$ $(n \geq 0)$.

For orthogonal (left-linear and non-overlapping) TRSs the origin function is the reversal of the well-known notion of *descendant* or *residual* [HL91]; 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.

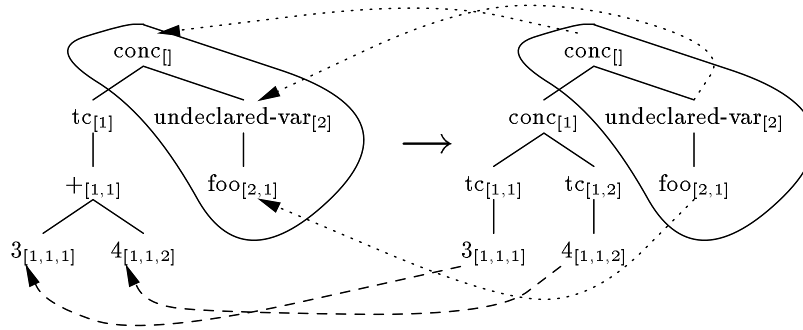
1.4 Example

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] within 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. Because of variable E_1 we have $org([1,1,1]) = \{[1,1,1]\}$, and according to 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$(X, X) \rightarrow X$" 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", e.g., was good, but it is less obvious that the empty set is the best origin for the two



Rewrite rule: $\mathrm{tc}(E_1 + E_2) \rightarrow \mathrm{conc}(\mathrm{tc}(E_1), \mathrm{tc}(E_2))$
Substitution: $\{E_1 \mapsto 3,\ E_2 \mapsto 4\}$
Context: $\mathrm{conc}(\square, \text{undeclared-var}(\mathrm{foo}))$

Dashed Lines: Origins for Common Variables
Dotted Lines: Context Origins.

Figure 3: Origins established for one rewrite step.

occurrences of "tc". Here we summarize some issues we should be aware of when dealing with (extensions of) origins.

Typically, having only origins based on the Common Variables case is not enough. These will only establish origins for literal recurrences of terms, not for newly created function symbols. Therefore, in addition to relations according to variables, links 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, however, will not do either. This would result in origin sets that are too big to give accurate information. On the other hand, 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 on just Contexts / Common Variables as *primary origins*. These are "beyond doubt" and needed in any kind of application. Moreover, we will deal with *secondary origins*, where the emphasis is on relations established because of function symbols occurring in left- and right-hand sides of rewrite rules. Proposals for secondary origins may be biased towards particular applications, more focusing 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 [Hee92]. 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 would be available [Hee92, Section 2.2]. Moreover, they indicated that a simple origin

scheme based on primary origins only rule would be inadequate.

This paper tackles these problems. First, we briefly summarize the definitions of higher-order rewriting in Section 2, together 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 some related work and draw some conclusions.

2 Higher-Order Term Rewriting

For the definition of Higher-Order Term Rewriting Systems (HRSs), we follow [Wol93, Nip91, OR93]. The main difference with the first-order case that terms in HRSs are constructed according to the simply-typed λ-calculus [Chu40].

2.1 The Simply-Typed λ-Calculus

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

Occurrences in λ-terms are defined as for the first-order terms; the only branches allowed are 1 and 2, which indicate how to navigate through abstractions (one son) and applications (two sons). As an example, Figure 4 shows all occurrences in the term (add $((\lambda N.N)$ zero) zero).

All the 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. 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* [Bar84], 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

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

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 $\lambda x_1 \cdots x_n.H(t_1, \ldots, t_m)$. The constant or variable H is called the *head* of such a term, $H(t_1, \ldots, t_m)$ is the *matrix*, and $\lambda x_1 \cdots x_n$ is the *binder*.

The rule of η-reduction states that terms of the form $\lambda x.(t\ x)$ can be transformed to just t, provided 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 $(\alpha_1, \ldots, \alpha_{m+k} \to \beta)$ $(k > 0)$, then apparently H expects more arguments, and we can add these as extra abstractions: the

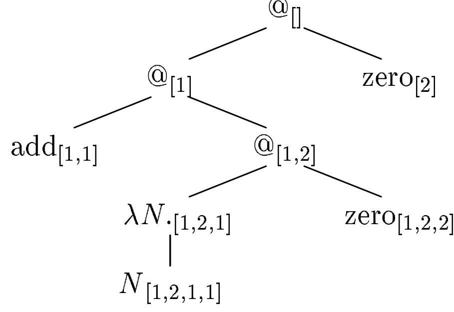


Figure 4: Occurrences in the term "(add $((\lambda N.N)$ zero)) zero".

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 one χ-reduction at occurrence u, we write this as $t \rhd_{\chi,u} t'$, or sometimes as $t' \lhd_{\chi,u} t$, where we may omit occurrence u. Repeated χ-reduction is written $t \rhd^*_\chi t'$. Since $\rhd^*_\alpha$ 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 $t\downarrow_{\beta\overline{\eta}} =_\alpha t'\downarrow_{\beta\overline{\eta}}$.

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 $p \to q$ 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 a substitution, and u an occurrence in $\mathcal{O}(t)$ denoting the redex position, is possible if:

- The types of the redex and the left-hand side 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 will turn 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 just assume that in some way it has been decided to apply a rewrite rule under a given substitution.

2.3 Example

Consider the second-order algebraic specification of a simple type checker shown in Figure 5, which was taken from [Hee92]. The idea of this specification is that all simple

expressions (being an identifier, string constant, or natural constant) are replaced by a term "tp(τ)", where τ is the type of that simple expression. This replacement is done in equations [1], [2], and [3]. Next, type correct expressions are reduced to their type, equation [4], and type correct statements are eliminated, equation [5]. The resulting normal form will contain only the incorrect statements.

An initial term (let us refer to it as P_1):

```
program( decls( decl(n,natural),  decls( decl(s,string), emptydecls) ),
         series( assign(s, plus(id(n),id(n))), emptyseries )   )
```

can be reduced according to equation (1) under, e.g., the following substitution σ_1:

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

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

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

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

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

To construct the result P_2 of this one-step reduction, we apply σ_1 to the right-hand side of equation (1), producing some term P_2''. Two more β-reductions are needed to transform P_2' to its β-normal form, which is the desired P_2. We can summarize this first single-step rewrite as follows:

$$P_1 \ \triangleleft_\beta \ P_1' \ \triangleleft_\beta \ P_1'' \equiv l_1^{\sigma_1} \ \rightsquigarrow \ r_1^{\sigma_1} \equiv P_2'' \ \triangleright_\beta \ P_2' \ \triangleright_\beta \ P_2$$

where $\rightsquigarrow$ 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 will also follow this "flow"; origins between P_2 and P_1 are defined using elementary origin definitions between P_2 and P_2', between P_2' and P_2'', etc.

3 Higher-Order Origins

We define origins for higher-order rewriting by (i) indicating how origins are to established for $\triangleright_\alpha$, $\triangleright_\beta$, $\triangleright_\eta$, and $\triangleright_{\overline{\eta}}$ conversion; then (ii) describing how the inverses $\triangleleft_\beta$ and $\triangleleft_{\overline{\eta}}$ 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 will refer to as *primary origins*. In the next section we will discuss various proposals and heuristics to extend these origins.

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
	series	: STAT, STATS	$\rightarrow$ STATS
	emptyseries	:	$\rightarrow$ 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:	D	: DECL $\rightarrow$ DECLS	X	: ID
	τ	: TYPE	S_0	: ID $\rightarrow$ STATS
	S	: STATS	N	: NAT
	R	: STRING		

equations:
[1] program(D(decl(X,τ)), $S_0(X)$) = program(D(decl(X,τ)), S_0(tp(τ)))
[2] nat(N) = id(tp(natural))
[3] str(R) = id(tp(string))
[4] plus(id(tp(natural)), id(tp(natural))) = id(tp(natural))
[5] series(assign(tp(τ), id(tp(τ))), S) = S

Figure 5: Part of the static semantics specification

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: a series of n branches b we abbreviate 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 $[1^{j-1}]$ and the path to t_i is $[1^n, 1^{m-i}, 2]$. See, e.g., the picture of a term in β normal form at the left of Figure 6.

3.1 Conversions

Let t, t' be terms, $u \in \mathcal{O}(t)$, and let χ be any of $\{\alpha, \beta, \eta, \overline{\eta}\}$. Assume $t \triangleright_{\chi,u} t'$. 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: $org(v) = \{v\}$.

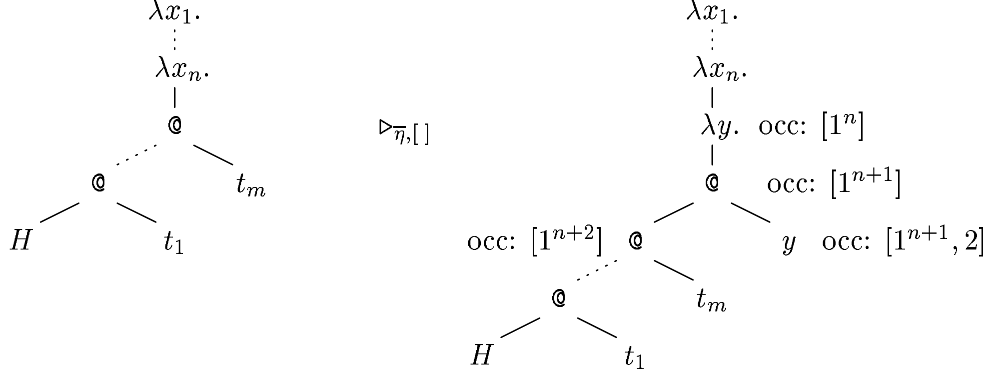


Figure 6: $\overline{\eta}$-Expansion.

- $\chi = \beta$;

  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 indicated by the dotted lines.

- $\chi = \eta$.

  In η-reduction one just forgets about a λ. 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 those old parts, while the origin of the new λ is the empty set:

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

  1. For $v' \prec [1^n]$, $org(u \cdot v') = \{u \cdot v'\}$.
  2. For $v' \in \{[1^n], [1^{n+1}], [1^{n+1}, 2]\}$, $org(u \cdot v') = \emptyset$.
  3. For $v' \succeq [1^{n+2}]$, let $v' \equiv [1^{n+2}] \cdot v''$. Then $org(u \cdot [1^{n+2}] \cdot v'') = \{u \cdot [1^{n+1}] \cdot v''\}$.

  The tree representation of $\overline{\eta}$-expansion shown in Figure 6 shows where occurrences $[1^n], [1^{n+1}], [1^{n+1}, 2]$ come from.

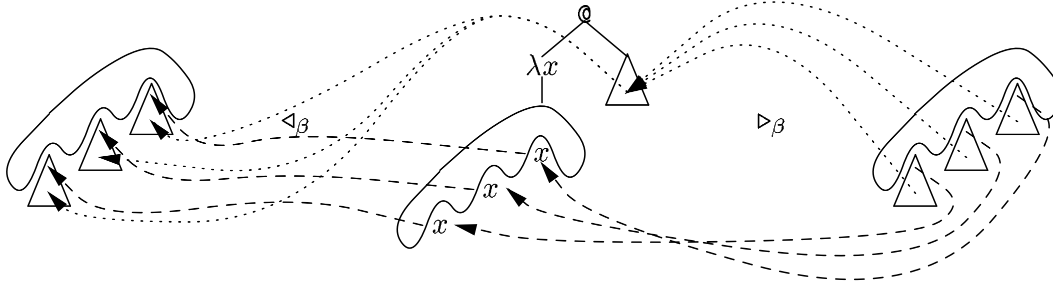


Figure 7: β-reduction in both directions.

It will turn out helpful to introduce the following concepts. 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 it is possible that an occurrence has 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 for α is identical, for η it is unitary, for $\overline{\eta}$ it is forgetful, and for β, finally, it is unitary and many-to-one. 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

As we have seen while discussing the example in Section 2.3, reversed β and $\overline{\eta}$-reductions also need to take place. The origin functions for $\triangleright_{\{\alpha,\beta,\eta,\overline{\eta}\}}$ defined in the previous section 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-conversion 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 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. This is illustrated by the dotted lines of Figure 7. Unfortunately, the origin definition for $\triangleleft_\beta$ is one-to-many, but in Section 4 we will discuss ways to deal with this.

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

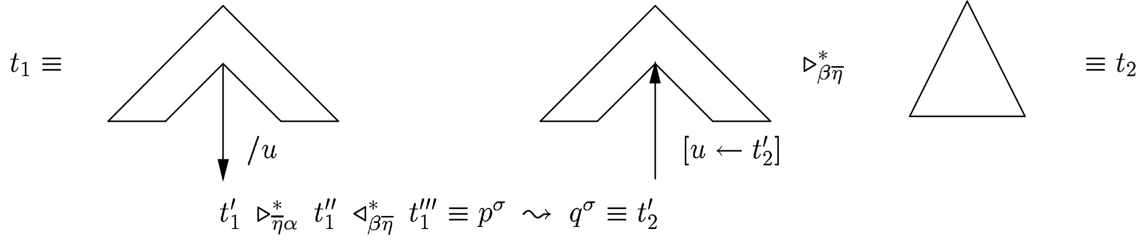


Figure 8: All conversions for one reduction step $t_1 \to t_2$, applying rule $p \to q$ at occurrence u in t_1 under substitution σ.

3.3 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 obviously is a forgetful definition, but in Section 4 this situation may improve. As in the first-order case, it is also possibly one-to-many (in the case of non-left-linearity). The Context case will be dealt with in the next section.

3.4 Rewrite Steps

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

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

$$P_1 \triangleleft_\beta P_1' \triangleleft_\beta P_1'' \equiv l_1^{\sigma_1} \rightsquigarrow r_1^{\sigma_1} \equiv P_2'' \triangleright_\beta P_2' \triangleright_\beta P_2$$

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, and 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 D and 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 look very intuitive). All occurrences of "natural" in P_2 have their origin to the declaration it came from (sounds alright).

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

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

Now 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 not yet optimal for use 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 these similarities within rewrite rules as well. Considering a rewrite rule $p \to q$, 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: it should contain the smallest possible number of holes and none of the holes h_j, h'_j ($1 \leq j \leq m$) should start with a non-empty context C' such that $h_j =_\alpha C[\overline{h}_1, \ldots \overline{h}_n]$ and $h'_j =_\alpha C[\overline{h'}_1, \ldots \overline{h'}_n]$. As an example, equation [1] of Figure 5 has a common context C = "program(D(decl(X,τ)), $S_0(\square)$)", 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 only point 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 variable occurring in the common context. For example, in equation [1] again, the origin of X at the right will only point to its

counterpart under the "decl" at the left, 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 $H = @(h_j, h_{j+1})$ and $H' = @(h'_j, h'_{j+1})$. As an example, equation [2] in applicative form reads @(nat,N) = @(id, @(tp, natural)). It would be counter-intuitive to regard the top-application as a common context $@(\square, \square)$ with two holes: h_1 = nat, h'_1 = id, and h_2 = N, h'_2 = @(tp, natural).

Note that this extended context case could 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 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 [DKT93]. 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 both a subterm of h'_j and of 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 looked for in the *uninstantiated* left- and right-hand side. This rule can in some cases lead to wild connections, but has proven its usefulness for the first-order case already [DKT93, Din93]. However, 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 [Din93].

   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. If after application of the head-to-head and common subterms case there still are constants in h'_j with an empty origin, obtain the set of all constant occurrences at the left-hole h_j as its origin set.

4.3 Abstraction and Concretization Degree

Recall from Section 3.2 that $\lhd_\beta$ conversions are one-to-many. Assume that $t' \lhd_\beta t$ with $t \equiv ((\lambda x.t_1)\ t_2)$. We will 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 to those with a higher abstraction degree.

In practice, however, such a preference may be a bit problematic. First of all, it need not be the case that a substitution with a low abstraction degree can be found at all. Secondly, repeated application of a substitution with abstraction degree 1 need not yield the same result as one application with a high abstraction degree. Finally, repeated application may be more expensive in terms of run time behavior, than a single application with a high abstraction degree. Therefore, we are currently studying two options to remedy these problems, where we consider $P_1 \lhd^*_\beta p^\sigma \rightsquigarrow q^\sigma \rhd^*_\beta P_2$ in its entirety:

1. In order to look for origins of a particular occurrence of a variable X in q, we need to know its corresponding occurrence in P_2. To relate this occurrence to P_1, we again need the corresponding occurrences of Xs in p. We propose that this information can be obtained by using a fresh variable Z appropriately, in place of an occurrence of 'X of interest', $\beta\overline{\eta}$-normalize the respective p^σ_Z and q^σ_Z to obtain $P_{Z,1}$ and $P_{Z,2}$. Occurences of Z in $P_{Z,2}$ are then related to occurrences of Z in $P_{Z,1}$.

   Let $\mathcal{O}_{fun}(t)$ be the set of function symbol occurrences in t. For $z' \in \mathcal{O}_{fvars}(q)$, denoting some variable X on the right-hand side, and $w \in \mathcal{O}_{fun}(matrix(X^\sigma))$, an occurrence in the instantiation of that variable,
   let $q' = (q[z' \leftarrow binders(X^\sigma).Z])^\sigma) \downarrow_\beta$, Z a new variable.

   If $v \equiv u \cdot v' \cdot w \in \mathcal{O}(P_2)$
   where $v' \in \mathcal{O}_{fvars(Z)}(q')$,
   then:

$$\begin{aligned} org(v) = \quad & \{u \cdot v'' \cdot w \mid q'/v' \equiv p'/v'', \\ & p' = \{(p[z \leftarrow binders(X^\sigma).Z])^\sigma\} \downarrow_\beta, \\ & z \in \mathcal{O}_{fvars(X)}(p)\} \end{aligned}$$

   Occurrence $v'' \in \mathcal{O}_{var}(p')$ denotes a recurrence[2] of X in the left-hand side p (via p').

2. Also, the possible *variations* of σ with different abstraction degrees could be taken into account in order to get the *most desirable* origins. One variation that seems to be very desirable, when comparing results to the first-order specification, is when the effect of abstraction degree 1 can be simulated by origin-tracking.

   In order to get origins of this nature, we need to appropriately modify the binding of substitution σ for the purposes of origin definition.

[2]Modifying X in σ instead will not give the same result. E.g., consider the rewrite rule $F(F(Y)) = d(F(Y))$ where in, if F is replaced by some new variable Z, the two occurrences of F merge to the top one.

We define $variants_1(\{\sigma\})$ as a set of substitutions with abstraction degree[3] no greater than 1, which are variants of σ.

If Σ is a set of substitutions, then $variants_1(\Sigma)$ is defined as follows:

- (Abstraction degree(Σ) $\not> 1$): $variants_1(\Sigma) = \Sigma$
- otherwise:
  $\sigma \in \Sigma$ has a mapping $[X \mapsto S]$ with abstraction degree(S) > 1.
  $variants_1(\Sigma) = variants_1((\Sigma - \{\sigma\}) \cup (\sigma \circ [X \mapsto variants_1(S)])$
  $\circ$ and $\mapsto$ are appropriately extended to sets and variants of a term are obtained by substituting a new variable (D) for all instances except one, defined as follows:
  $$\begin{aligned} variants_1(S) = \quad & \{S' \mid S' = binders(S).(matrix(S)[(V_X - \{s\}) \leftarrow D]), \\ & X \in variables(binders(S)), \\ & V_X = \mathcal{O}_{fvars(X)}(matrix(S)), \\ & s \in V_X\} \end{aligned}$$
  where the extension of the replacement function $S[X \leftarrow T]$, to sets of occurences X, replaces all occurences of $x \in X$ by term T.

Then we follow the previous origin definition (in 1 above), for every $\sigma' \in variants_1(\{\sigma\})$ to get the more desirable origins.

We are in the process of finding out whether these proposals can help to improve the origins for $\triangleleft_\beta$ reductions.

4.4 Example

With these extensions, the origins for the example of Section 2.3 run smoothly. We will assume that equation [1] is applied with substitutions of abstraction degree 1 only. The extended contexts take care that "program" and "decl" are linked. Moreover, the effect of linking variables in contexts only to that 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 an origin of the "tp" to the "plus". Finally, common subterms will cause, also for equation (4), the "tp(natural)" to be linked to both occurrences of "tp(natural)" in the "plus" expression.

The example given here is only part of the specification discussed in [Hee92]. The origins with extensions create the right 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 for the automatic generation of parts of compilers or programming environments – in particular error handlers, symbolic debuggers, and animators. The extension to primary origins studied in [DKT93] established relations between *common subterms* in left

[3] Abstraction degree of set is the maximum of the abstraction degrees of its components. Abstraction degree(σ) is the maximum of the abstraction degrees of the substituends in σ and the abstraction degree of a term is maximum of the abstraction degrees of the variables in its binder.

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 [Kli93] are discussed as well [DKT93]. The applicability of origins in practice, using a specification of the semantics of a subset of Pascal, is studied by Dinesh and Tip: the static semantics and generated error handler is covered in [Din93], the dynamic semantics and generated animator is described in [Tip93]. In order to improve origin tracking for *syntax-directed* specifications (typically translators or type checkers), an extension for *primitive recursive schemes* is proposed in [Deu93]. Finally, Field and Tip are studying *creation/residuation* tracking, an extension in which *responsibility* for the birth of function symbols is formalized.

The study of origins was pioneered by Y. Bertot [Ber91, Ber92], 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*. Part of his work has been implemented in the CENTAUR system [BCD+89]. In particular, the specification language TYPOL [Kah87] has been extended with *subject tracking* [Des88].

Closely related to origins are *residual maps*, *descendants*, or *labelings* [Lév75, HL91, Mar91, Fie91], 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 [Ber92]: he investigates how origins for TRSs can be used to simulate labeling systems for the λ-calculus. The labels of [Lév75] 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 [Nip91]. We have discussed origins using the more liberal setting of Wolfram [Wol93]. 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) [Klo80] as described in [OR93] can be the basis for a definition of origins for CRSs.

6 Conclusions and Future Work

Origin tracking for higher-order specifications is considerably more difficult than establishing origin relations for the first-order case. All kinds of conversions, which can 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.

Another issue is the study of origins as transformations on HRSs. Tip is conducting such experiments for the first-order case already. For the higher-order case, it may be useful to use specifications of the λ-calculus with explicit substitutions as in [ACCL90].

Finally, after having seen so many variants of origin tracking, it may be worthwhile to investigate whether a generalization to some kind of origin *scheme* can be made. 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. Femke van Raamsdonk and Machteld Vonk commented on earlier drafts.

References

[ACCL90] 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.

[Bar84] H.P. Barendregt. *The Lambda Calculus; its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathatematics*. North-Holland, 1984.

[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).

[Ber91] Y. Bertot. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle*. PhD thesis, INRIA, Sophia-Antipolis, 1991. In French.

[Ber92] 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.

[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.

[Chu40] A. Church. A formulation of a Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

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

[Deu93] A. van Deursen. Origin tracking in primitive recursive schemes. Technical Report CS-R93xx, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1993. To appear.

[Din93] 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.

[DKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special Issue on Automatic Programming.

[Fie91] J.H. Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems.* PhD thesis, Cornell University, 1991.

[Hee92] J. Heering. Second-order algebraic specification of static semantics. In these proceedings, 1992.

[HL91] 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.

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

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[Klo80] J.W. Klop. *Combinatory Reduction Systems.* Number 127 in Mathematical Center Tracts. Mathematisch Centrum, Amsterdam, 1980.

[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.

[Lév75] J.-J. Lévy. An algebraic interpretation of the $\lambda\beta$K-calculus and a labelled λ-calculus. In C. Böhm, editor, *λ-Calculus and Computer Science Theory*, number 37 in LNCS. Springer-Verlag, 1975.

[Mar91] 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.

[Nip91] 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.

[OR93] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In these proceedings, 1993.

[Tip93] 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. To appear.

[Wol93] D.A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993.