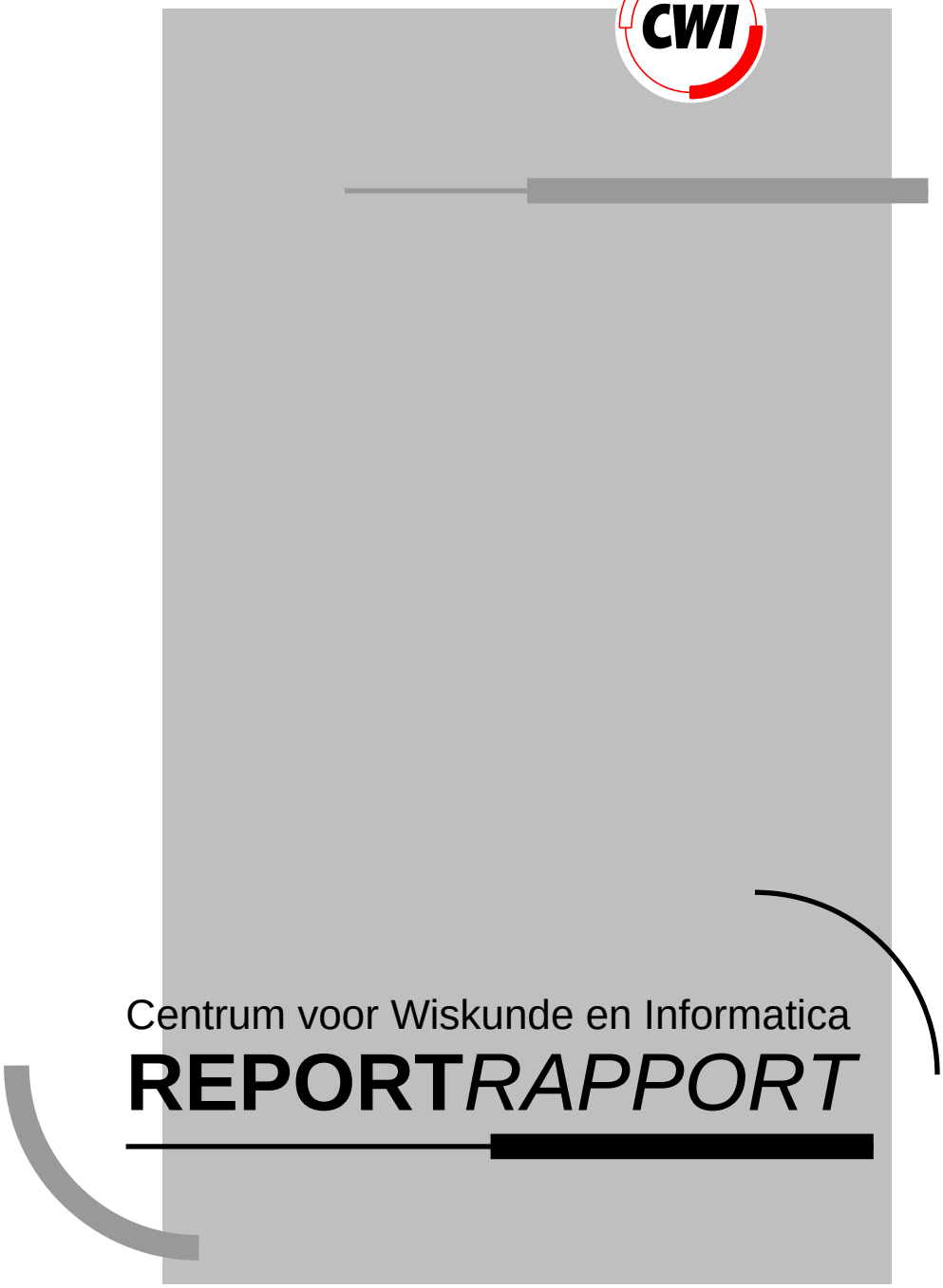

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Origin Tracking in Primitive Recursive Schemes

A. van Deursen

Computer Science/Department of Software Technology

CS-R9401 1994

Origin Tracking in Primitive Recursive Schemes

Arie van Deursen

CWI

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

Email: arie@cwi.nl

Abstract

Algebraic specifications of programming languages can be used to generate language-specific programming support tools. Some of these can be obtained in a straightforward way by executing language specifications as term rewriting systems. More advanced tools can be obtained if the term rewriting machinery is extended with *origin tracking*. Origin tracking is a technique which automatically establishes a relation between subterms of the result value (normal form) and their *origins*, which are subterms of the initial term. For specifications having a syntax-directed nature, as formalized by the class of so-called *primitive recursive schemes*, high-quality origins can be established. The definition, properties, extensions, and implementation of these so-called *syntax-directed origins* are discussed.

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, programming language semantics, programming environments, program generation, origin tracking, program schemes, primitive recursion.
Note: Partial support has been received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II: GIPE II) and from the Netherlands Organization for Scientific Research (NWO), project *Incremental Program Generators*.

1. Introduction

One of the benefits of formal definitions of programming languages is that programming support tools can be generated (semi-)automatically from them. One way to achieve this is based on algebraic specifications [BHK89, Wir90]. The syntax of a language is defined in a signature, and properties of the language, such as static or dynamic semantics, are described by equations. The syntax can be used to derive parsers, and equations can be executed as term rewriting systems (TRSs) [Klo92], giving rise to elementary tools such as type checkers or evaluators. Combining these tools with syntax-directed editors and proper user-interface results in programming-environment generation from algebraic language specifications [Kli93].

In order to enhance the level of sophistication of the tool generators, term rewriting can be extended with a facility called *origin tracking* [Ber92, DKT93]. We will illustrate the need for origin tracking by means of a small example.

Consider 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 type check 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 ($n \geq 0$). Just carrying out the reduction will only give this list, but doing it in combination with origin tracking will give additional information: For each error E_i the origin tracking mechanism indicates which statement, expression, identifier, or other part of the initial term $tc(p)$ was responsible for the generation of E_i. In other words, the *origins* of each E_i in the initial term $tc(p)$ are identified.

Origin tracking has actually been implemented, so we can illustrate this by the programming environment shown in Figure 1. A programmer entered a program (in the large window), and invoked a type check which resulted in a list of four messages (in the small window). He asked for more information concerning the error message "multiply-defined-label `step`" (in the small window) by putting his *focus* (the small box around `step`) on a piece of this message and by subsequently clicking on the "Show Origin" button. This caused the relevant occurrences of "`step`" in the original program to be highlighted in the large window. Note that not *all* occurrences of `step` are highlighted but only those actually related to the error message.

This was an application of origin tracking in the field of *error handling*: algebraically specified type checkers can automatically be extended to show the error positions. Another typical application is program *animation*, where a program is executed in such a way that the programmer can see what is happening in his program. Such animators can be generated from a specification of the dynamic semantics of the language. An animator generator can use origin information to map structures needed during execution to constructs occurring in the actual program (see [Tip93] for an example of animator generation).

Origin tracking is a general technique. For an arbitrary specification, it establishes relations between subterms in a normal form and subterms in the initial term, where the latter subterms are called the *origins* of the former. The details of how to compute these origins are presented in Section 3. It need not be easy to define an origin function that is generally applicable and always computes the right origins. Even though the origins from [DKT93] have been used successfully, there are many specifications for which they are insufficient.

The extension of origin tracking we are proposing is based on the observation that many functions occurring in first-order algebraic specifications are defined using some form of primitive recursion. Such definitions are formalized in the class of so-called *Primitive Recursive Schemes* (PRSs) [CF82]. The extra knowledge concerning the definition of certain functions that is available in a PRS allows us to derive special origins for these functions. We present our proposal in full detail in Section 5. Since pure PRSs are quite rigid in their requirements, we also discuss a further generalization to arbitrary specifications with a syntax-directed nature in Section 6. We refer to these origins as *syntax-directed origins*

The latter name, syntax-directed, is not a coincidence. PRSs are proven to be equivalent to attribute grammars [CF82], which in turn are a formalization of syntax-directed definitions. When transposed to the algebraic specification framework, attribute grammars define a function by primitive recursion over an abstract-syntax tree.

In summary, our paper presents a theoretical notion, origins in primitive recursive

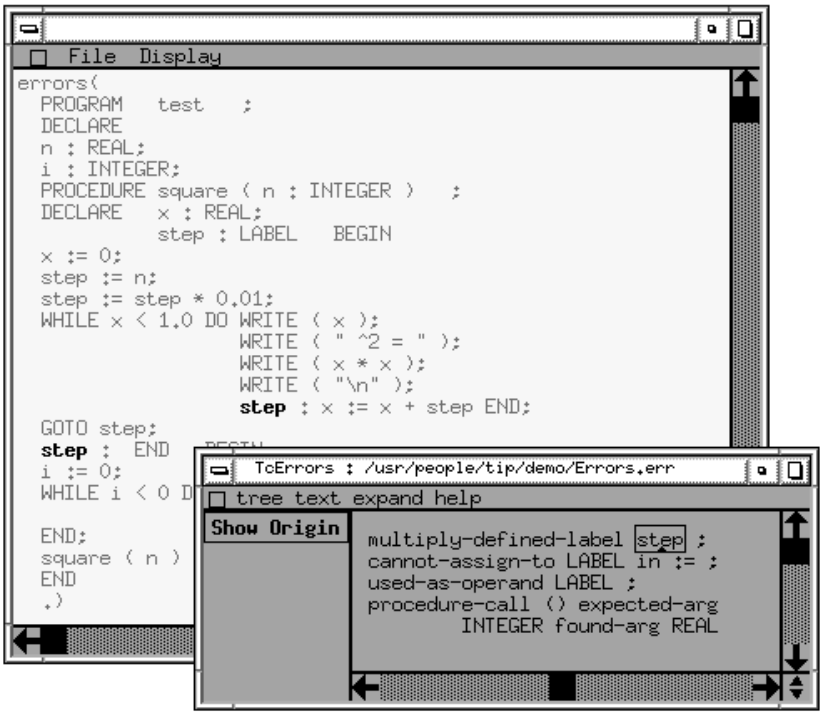

Figure 1: Example of a generated environment using origin tracking.

schemes, with a practical goal in mind: high-quality tool generation from formal language definitions. New in our paper are not only kernel Sections 5 and 6, but also parts of Section 3: the presentation style of Sections 3.2 and 3.4 is rather different from that in [DKT93], and the problem analysis in Sections 3.5 and 3.6 was not given in [DKT93]. The definition of PRSs in Section 4 was taken from [Meu92, CF82].

2. Related Work

The study of origins was pioneered by Bertot [Ber90, Ber92, Ber93]. He investigated applications of origin tracking to source-level debugging given a specification in natural semantics style [Kah87, Ber90]. Furthermore, he considered the relation between origins for the λ-calculus and for TRSs [Ber92], and introduced a formal framework to reason about origin functions [Ber93]. Bertot focused on orthogonal, unconditional TRSs, where an origin consists of at most one subterm occurrence. Part of his work was implemented in the Centaur system [BCD+89]; in particular the notion of a *subject* occurring in the specification language Typol is akin to syntax-directed directed origins.

Bertot's ideas were picked up in [DKT93], where origins were extended to sets of occurrences and defined for arbitrary TRSs with conditional rules. Moreover, an implementation in the ASF+SDF programming environment generator [Kli93] was described. An extension to higher-order term rewriting systems was given in [DD93].

Practical experience with origin tracking is described by Dinesh and Tip [Din93, Tip93]. Dinesh presents a specification style for the definition of static semantics of programming languages based on abstract interpretation. He shows how origins can be used to generate

type checkers with good error pinpointing facilities. Tip discusses an algebraic specification of an interpreter for a Pascal-based language, and explains how origin tracking can be applied to obtain an animator for this language.

The latter two papers were part of the inspiration to start working on specialized origins for PRSs. Syntax-directed origins will widen the class of algebraic specifications for which origin tracking can be applied easily and successfully. In particular, the patterns used to specify the animation behavior for Tip's animator become much simpler, and the adaptation of the abstract syntax proposed by Dinesh to improve his origins becomes unnecessary.

On the theoretical side, origins are related to so-called *residuals* or *descendants* [HL91, Mar91], which are used in the search for optimal reduction strategies. Currently, Field and Tip are extending residuals to *creation/residuation tracking* using ideas from incremental rewriting as described by [Fie93].

The notion of a *program scheme* [Cou90] is a general device to understand control structures: loops, iterations, goto's and so on are translated to functions defined recursively by equations. Primitive Recursive Schemes were introduced by Courcelle and Franchi-Zannettacci in order to understand attribute grammars (AGs). They gave a one-to-one correspondence between PRSs and AGs [CF82]. As an example of an application of this mapping, Van der Meulen has used techniques for incremental computations in to obtain the effect of incremental rewriting [Meu92].

One of the aims of our origin tracking technique is to get, in an algebraic framework, error messages with good location information associated with it. In AGs, error messages typically correspond to attributes of type string. The error position is identified by printing the message close to the text position of the grammar node the attribute belongs to. In the Synthesizer Generator [RT89] the language describer can give unparsing (pretty print) rules to indicate where the error message should be printed. If there is no error, the message attribute contains the invisible empty string.

To compare this with our approach, consider an identifier `x` of type string in an expression like `- x`, causing a message like `integer instead of string expected`. In an AG this message will be associated with the `-` node, which gives useful information. In our approach, the message can have origins to (1) the position where the type inconsistency was detected which is the `-` node as in AGs, (2) the place where `x` was declared of type `string`, and (3) the position where `x` was not used with type `string`. Typically, the signature for error messages will include a "`_ instead of _ expected`" symbol, and origins are associated with the two arguments as well as with the top function symbol.

Although not intended for this purpose, our technique could be used to enhance error location in AGs, provided the attribute evaluation mechanism is in some sense based on term rewriting.

3. Principles of Origin Tracking

We present some basic notions concerning term rewriting and ordinary origin tracking. The problems with existing origins as well as the difficulties when extending them, are discussed.

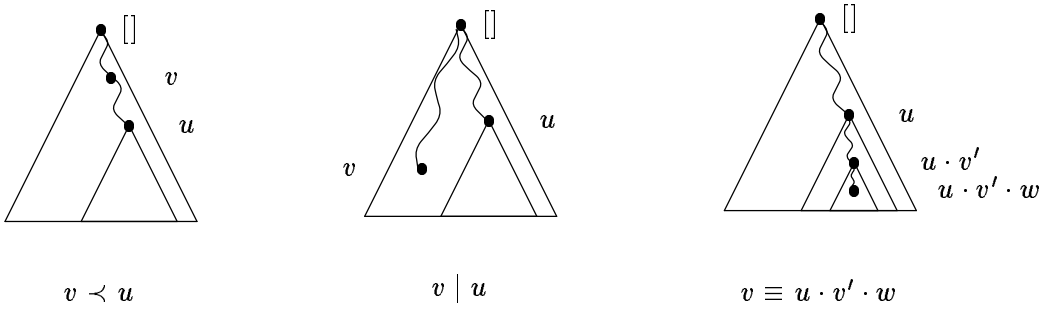

Figure 2: Relative positions of v with respect to contractum position u

3.1 Preliminaries

Before defining origins, we borrow some preliminary definitions concerning first-order term rewriting from [Klo92, HL91]. A term t can be *reduced* to a term t' according to a *rewrite rule* $r : \alpha \to \beta$ by identifying a context $C[]$ and subterm s in t such that $t \equiv C[s]$, and by finding a substitution σ such that $s \equiv \alpha^\sigma$. Then $t \equiv C[\alpha^\sigma]$ rewrites to $C[\beta^\sigma] \equiv t'$ by one *elementary reduction*, written $t \to t'$. We call α^σ the *redex*, and β^σ the *contractum*. For multi-step reductions $t_0 \to t_1 \to \cdots \to t_n$ 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 $[n_1, \ldots, n_m]$ $(m \geq 1)$ 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. Occurrences are concatenated by the (associative) $\cdot$ operator. If u, v, w are occurrences and $u = v \cdot w$, then v is *above* u, written $v \preceq u$. Also, if $w \neq []$ then we write $v \prec u$. 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 denoted by $\mathcal{O}(t)$. Similarly, $\mathcal{O}_{var}(t)$ is the set of occurrences of variables in t, and $\mathcal{O}_{fun}(t)$ the set of function (or constant) symbols in t. The number of elements in a set O of occurrences will be written $|O|$.

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.

3.2 Definition of the Origin Function

Let $t \xrightarrow{u,\sigma}_r t'$, where r is a rule $\alpha \to \beta$, be an elementary reduction step. With each step we associate a function *org-step* : $\mathcal{O}(t') \to \mathcal{P}(\mathcal{O}(t))$ mapping occurrences in t' to sets of occurrences in t. Let $v \in \mathcal{O}(t')$. We define *org-step* by distinguishing the following cases: (see also Figure 2):

- (Context)

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

- (Common Variables)

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

$$\textit{org-step}(v) = \{u \cdot v'' \cdot w \mid \alpha/v'' \equiv X\}$$

(Note that $v'' \in \mathcal{O}_{var}(\alpha)$ is the occurrence of X in the left-hand side α of r).

- For the time being, we will assume that $\textit{org-step}(v) = \emptyset$ for the remaining case, i.e., where v denotes a function symbol in the right-hand side of r (see also Section 3.6).

This function *org-step* covers elementary reductions. It is generalized to a function org^* for multi-step reductions $t_0 \to t_1 \to \cdots \to t_n$ $(n \geq 0)$ by considering the origin functions for the individual steps. Let us call, for the i-th elementary reduction $t_{i-1} \to t_i$ $(0 < i \leq n)$, the associated origin function $\textit{org-step}_i$. Recursively define $org^j : \mathcal{O}(t_j) \to \mathcal{P}(\mathcal{O}(t_0))$ for $0 \leq j \leq n$, and $v \in \mathcal{O}(t_j)$:

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

Then org^* is equal to org^n for multi-step reduction $t_0 \to^* t_n$.

Given a multi-step reduction $t_0 \to t_n$, with associated function org^* and occurrence $u \in \mathcal{O}(t_n)$, the set $O = org^*(u)$ is called *the origin set* of t_n/u, and the elements of O are called *the origins* of t_n/u. Often it is natural to relax the difference between sets and elements. If no confusion is possible we will, for example, use "subterm s has an origin" to indicate that the origin set of s is non-empty, and "subterm s has multiple origins" to state that the origin set of s contains more than one occurrence.

3.3 Example

As an example, consider the specification of Figures 3, 4 and 5 (inspired by [Meu88, Bro92]). The language designer has specified a translation of a simple language to assembly code. Signature describing the syntax of source and target language are given in Figures 3 and 4. Figure 5 contains the signature and equations for the actual translation functions. At the moment, we can ignore the underlining and bold face fonts used for the symbols (see Section 4). A reduction of term "tr-expr(const(4) + const(3))", which results in the normal form "push(4) @ push(3) @ add @ null", is shown in Figure 6.

The dotted lines in the figure indicate the origin relations established for this reduction. In the first rewrite step, equation [3] is applied. Since variables E_1 and E_2 occur both in its left- and right-hand side, origin relations are established between their instantiations, i.e., between the occurrences of "const(4)" and "const(3)" respectively. In the remaining rewrite steps, in particular those where equation [2] is applied, only the origins for the constants "3" and "4" survive, as indicated by the dotted lines in the figure.

3.4 Properties

Origins are very similar to the better-known *residuals* or *descendants* [HL91], which are used to study the survival of redexes during reductions. Let $A : t \to^* t'$ be a reduction, let $v \in \mathcal{O}(t)$ and $v' \in \mathcal{O}(t')$, and let org^*_A be the origin function for A. Then we have for orthogonal (left-linear and non-overlapping) TRSs:

Property 1 *Assume reduction A is performed in an* orthogonal *TRS, and let $\backslash A$ be Huet and Lévy's residual mapping for reduction A. Then $v \in org^*_A(v') \Leftrightarrow v' \in v\backslash A$.*

Sorts:	EXP STAT ...		
Functions:	**if _ then _ else _ fi**:	EXP × STAT × STAT	→ STAT
	const:	INT	→ EXP
	_ + _ :	EXP × EXP	→ EXP
	...		

Figure 3: Abstract syntax of simple statements and expressions.

Sorts:	ASSEMBLY COMMAND LABEL		
Functions:	null:		→ ASSEMBLY
	_ @ _ :	COMMAND × ASSEMBLY	→ ASSEMBLY
	cjump:	LABEL	→ COMMAND
	jump:	LABEL	→ COMMAND
	lab:	LABEL	→ COMMAND
	push:	INT	→ COMMAND
	add:		→ COMMAND
	...		
	0,1,2,3:		→ LABEL
	_ ^ _ :	LABEL × LABEL	→ LABEL

Figure 4: Part of the abstract syntax for a simple assembly language.

For left-linear TRSs, we can say something about the size of the origin sets.

Property 2 *Assume reduction A is performed in a* left-linear *TRS. Then for every $v' \in \mathcal{O}(t')$ we have $|org_A^*(v')| \leq 1$.*

For arbitrary TRSs, we can only say that the sets or smaller than the number of nodes in the initial term.

Property 3 *For every $v' \in \mathcal{O}(t')$ we have $0 \leq |org_A^*(v')| \leq |\mathcal{O}(t)|$*

Finally, for arbitrary TRSs, origins only point to syntactically identical terms:

Property 4 *For every $v \in org_A^*(v')$ we have $t/v \equiv t'/v'$.*

Note, however, that these properties need not hold for the extensions we will be proposing (see Section 5.4).

3.5 Limitations

The origin definition just presented establishes the most fundamental origin relations, and works for all algebraic specifications. Experiments in practical situations show that these origins can already be quite useful [Din93, Tip93].

Functions:

tr-stat:	STAT × LABEL	→	ASSEMBLY
tr-exp:	EXP	→	ASSEMBLY
...			
_ ; _ :	ASSEMBLY × ASSEMBLY	→	ASSEMBLY
...			

Variables:

E_1, E_2:	EXP	N:	INT
S_1, S_2:	STAT	*Alist*:	ASSEMBLY
L:	LABEL	C:	COMMAND

Equations:

[1] tr-stat(**if** E **then** S_1 **else** S_2 **fi**, L) =

tr-exp(E)	;	%% condition
cjump(L)	@	
tr-stat(S_2, $1\hat{}L$)	;	%% else part
jump($3\hat{}L$)	@	
lab(L)	@	%% then part
tr-stat(S_1, $2\hat{}L$)	;	
lab($3\hat{}L$)	@	
null		

[2] tr-expr(**const**(N)) = push(N) @ null

[3] tr-expr(E_1+E_2) = tr-expr(E_1) ; tr-expr(E_2) ; add @ null

[4] null ; *Alist* = *Alist*

[5] (C @ *Alist*) ; *Alist'* = C @ (*Alist* ; *Alist'*)

...

Figure 5: Example specification of a simple translation. Equation [1] defines the translation of an if-statement. Equations [2] and [3] specify the compilation of expressions, and [4] and [5] define concatenation of assembly programs.

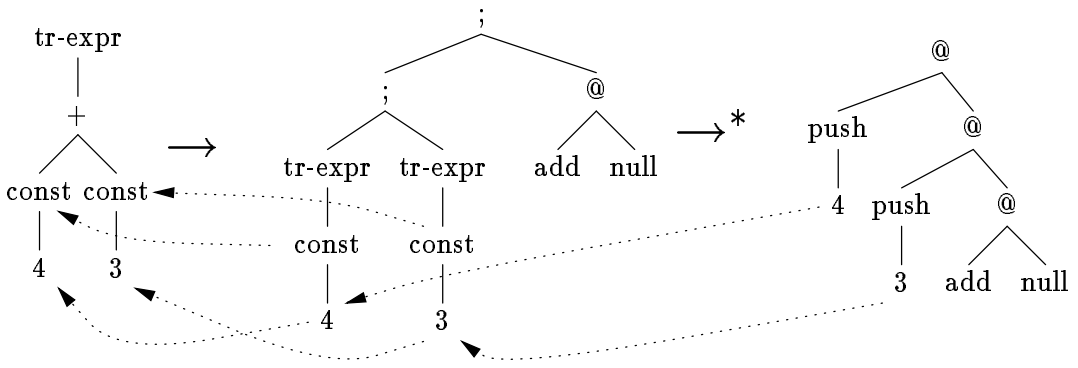

Figure 6: Part of a reduction performing the translation of an expression. The dotted lines indicate origin relations.

Nevertheless, for several specifications it must be possible to establish more and better origins. This is illustrated by the simple example we have seen in Figure 6. There are no arrows from nodes "add" or "null" to the initial term. In other words, these nodes have an empty origin, even though there seem to be good candidate occurrences in the initial term; e.g., it seems intuitively plausible to link "add" to "+".

This problem shows up clearly in equation [1] as well. Origins are established for all variables E, S_1, S_2, and, L, but none of the function symbols in the right hand side, e.g. "jump", "cjump", "lab", ..., get an origin. This is undesirable since these are the symbols that will occur in the resulting normal form.

In general, the problem is that function symbols introduced in the right-hand side of a rule have an origin consisting of the empty set of occurrences. This is unattractive, since it provides very little information on why such a function symbol has been created. This problem is mentioned briefly in [DKT93], where it is noticed that difficulties arise in specifications "having the flavor of translating terms from one representation to another". For these specifications "more origin relations between both sides of the equations have to be established".

We will propose an extension which solves this problem. It will establish good origins for function symbols that are created during rewriting in the context of primitive recursive schemes

3.6 Extending Origins

Having noticed the limitations of the existing scheme, one may wonder why it is so difficult to present a suitable extension. Ideally, origins should meet the following requirements:

(A) One would like the origin sets to be as specific as possible. Thus, rather than having an origin which, e.g., states that this assembly instruction originated from the set of all statements in the source program, one would like to know exactly which statement was responsible. In other words: *Keep the origin sets small.*

(B) However, having the empty set as origin provides little information. Moreover, some applications require that multiple origins be established; e.g., for error handling purposes one would like to have origins both to a declaration of an identifier and its conflicting usage. Thus: *Do not make the origin sets too small.*

(C) Moreover, the higher a path points in the initial term, the smaller the information content. For instance, having an origin to the top-node of the initial term will only point out that the normal form somehow has resulted from the initial term. This does not provide very much information. Hence: *Keep the origins deep.*

(D) On the other hand, origins that point too deep may be misleading as well. If, again in an error-handling example, an expression "plus(E_1, E_2)" has incompatible argument types, the origin for a message indicating this should point to either the plus or both the top nodes of E_1 and E_2, but not to a very deep subexpression occurring within E_1. Therefore: *Do not make the origins too deep.*

A proposal to extend origin tracking should find a compromise between these conflicting requirements.

4. Primitive Recursive Schemes

A Primitive Recursive Scheme (PRS) is a program scheme formalizing the notion of functions defined inductively (using primitive recursion) over some structure. A typical example of a PRS is a type checker defined inductively over the syntax of a programming language. Definitions of PRSs can be found in [CF82, Meu92]. We follow [Meu92]:

(i) A PRS is a five-tuple $\langle G, \Phi, S, E_\Phi, E_S \rangle$, with G, S signatures, Φ a set of functions, and E_Φ, E_S sets of equations. A PRS corresponds to an algebraic specification $\langle \Sigma, E \rangle$ with signature $\Sigma = G \cup S \cup \Phi$ and equations $E = E_\Phi \cup E_S$;

(ii) All functions in G are free constructors (i.e., there are no equations containing only functions from G);

(iii) The type of the first argument of each ϕ in Φ is a sort from G, and the types of all other arguments (the *parameters* of ϕ) as well as the output type are sorts of S;

(iv) E_Φ consists of the Φ-*defining equations*: For each constructor $p : X_1 \times \cdots X_n \to X_0$ in G and each function ϕ in Φ, E_Φ contains exactly one defining equation:

$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau \tag{4.1}$$

(v) All equations in E_Φ are strictly decreasing in G: i.e., in equation (4.1), the only G-terms allowed in τ are $x_1, ..., x_n$;

(vi) All equations are left-linear (all variables in the left-hand side of each equation are distinct).

A typical example of a PRS is the specification shown in Figures 3, 4 and 5. The G-signature of this PRS consists of the **boldface** function symbols (introduced in Figure 3), defining the grammar of a simple programming language. The Φ-functions of this PRS are underlined functions tr-stat and tr-expr (introduced in Figure 5). They are defined using primitive recursion in equations [1], [2], and [3]. These equations satisfy the requirements (iv), (v), (vi) for Φ-defining equations. The S-signature consists of the functions of Figure 4 as well as function _ ; _ from Figure 5, defining result values as well as auxiliary functions.

In summary, a PRS contains a set Φ of functions defined inductively over the abstract syntax trees of G. Context information is passed downward using the parameters of the Φ-functions. The effect of the Φ-functions is a mapping of G-terms to S-terms. Equations over S may be used to define further computations or simplifications of resulting terms.

5. Origins in PRSs

5.1 Introduction

So how can we define origins in PRSs? Let us first try to understand what is going on in a PRS. A large G-term (typically the abstract syntax tree of some program) is processed by Φ-functions; different Φ-functions operate on different G-constructors p, e.g., there will be one Φ-function to translate an if-statement, another one to translate an assignment, and so on. To see what is going on, consider a Φ-defining equation $\phi(p(x_1, ..., x_n), y_1, ..., y_m) = \tau$. The right-hand side τ is a formula to compute a particular value for some grammar node p. It consists of:

(1) Variables (the x_i occurring in τ) representing subconstructs of the current node p.

(2) Variables (the y_j occurring in τ) representing context information;

(3) Function symbols initiating computations over subconstructs of the current node (Φ-functions in τ, with some x_i as first argument)

(4) Function symbols from S indicating how to "synthesize" the result value from the ingredients mentioned above, or how to construct context information to be passed as parameters to the Φ-functions occurring in τ.

This division is reflected in the origins we define. The Common Variables case (Section 3.2) is used to take care of (1) and (2). For case (3) we have a Φ-function ϕ' operating on a subconstruct, and we will give ϕ' an origin to its first argument, that is, to some x_i. Finally, for case (4) the new function symbols are created when working on the $p(x_1, ..., x_n)$ grammar node in the left-hand side; therefore, these new function symbols will obtain an origin to the p node in the left-hand side. These origins caused by Φ-functions traversing the abstract syntax tree are the kernel of the PRS-origins. The remaining origins, concerning equations over S, simply propagate these "Φ-origins", which is achieved by giving all new function symbols in the right-hand side of a rewrite rule from E_S an origin to the top-symbol of the left-hand side. A precise definition is given in Section 5.3.

5.2 Example

As an example, let us study again the reduction of "tr-expr(const(4) + const(3))" according to the equations in Figure 5. The PRS origins for this reduction are shown in Figure 7.

The relations between the constants "3" and "4" in the normal form and initial term are established because of Common Variable N when applying equation [2] of Figure 5.

The relations between "push" and "const" result from reductions according to Φ-defining equation [2]: the S-function symbol "push" gets the G-argument "const(N)" of Φ-function "tr-expr" as origin. Likewise, S-function symbols "_ ; _", "_ @ _", "add" and "null" introduced in Φ-defining equation [3] get an origin to the "_ + _" G-argument of Φ-function "tr-expr".

Finally, equations [4] and [5] are used to eliminate the concatenation of assembly code operator "_ ; _". New functions introduced in these S-equations receive the top-function symbol of the left-hand side as origin. Since in this case the "_ ; _" operators were introduced by equation [3], these origins point to the "+" function symbol.

5.3 Definition

For a PRS $\langle G, \Phi, S, E_\Phi, E_S \rangle$ and term t, we introduce $\mathcal{O}_\Phi(t)$, $\mathcal{O}_G(t)$, and $\mathcal{O}_S(t)$ as the sets of occurrences denoting a function symbol from Φ, G, and S respectively. To define origins for PRSs, we again consider an elementary reduction $t \equiv C[\alpha^\sigma] \rightarrow C[\beta^\sigma] \equiv t'$. Let u be the occurrence in C of the redex position. The function *prs-org-step* : $\mathcal{O}(t') \rightarrow \mathcal{P}(\mathcal{O}(t))$ maps occurrences in t' to sets of occurrences in t. Define *prs-org-step*(v) by taking the "Common Variables" and "Context" cases of *org-step*, together with the following cases, where v is the occurrence of a function (or constant) symbol in β:

- (Φ-Functions)

  If $v \equiv u \cdot v'$ with $v' \in \mathcal{O}_\Phi(\beta)$ the occurrence of a Φ-function symbol in the right-hand side β then

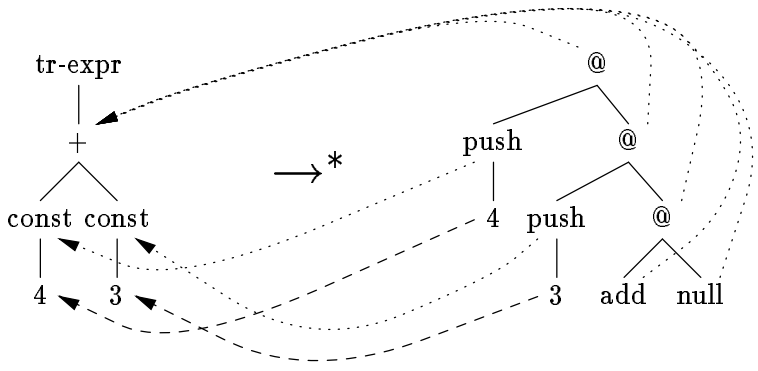

Figure 7: Syntax-Directed Origins for a Simple Reduction

$$\textit{prs-org-step}(v) = \textit{prs-org-step}(\ v \cdot [1]\ )$$

In other words, the origin of a Φ-function is equal to the origin of its G-argument.

- (Synthesizers)

  If $v \equiv u \cdot v'$ with $v' \in \mathcal{O}_S(\beta)$ the occurrence of a function symbol from S in the right-hand side β, and $r \in E_\Phi$ is a Φ-defining equation with left-hand side $\alpha \equiv \phi(p(x_1, ..., x_n), y_1, ..., y_m)$, then

  $$\textit{prs-org-step}(v) = \{u \cdot [1]\}$$

  In other words, the origin is the G-term $p(x_1, ..., x_n)$ as it occurs in the left-hand side.

- (Auxiliary Symbols)

  Finally, if rule $r \in E_S$, and $v \equiv u \cdot v'$ with $v' \in \mathcal{O}_G(\beta) \cup \mathcal{O}_S(\beta)$ the occurrence of a function symbol from G or S in the right-hand side, then

  $$\textit{prs-org-step}(v) = \{u\}$$

  In other words, the origin is equal to the top-symbol of the left-hand side.

At first sight the definition of *prs-org-step* for the Φ-Functions case may seem a little dangerous since *prs-org-step* appears at both sides of the equality sign. However, the first argument of a Φ-function must — by definition of a PRS — be a G-term, for which the *prs-org-step* function is directly defined in the remaining cases. When we know that rule r actually is a Φ-defining equation $\phi(p(x_1, ..., x_n), y_1, ..., y_m) = \tau$ we can even make a stronger statement: A Φ-function ϕ' occurring at position v in right-hand side τ must — again by definition of a PRS — have one of the $x_i (1 \leq i \leq n)$ as its first argument. The occurrence of that x_i in the left-hand side is $[1, i]$, so for this case we can define *prs-org-step* alternatively as $\textit{prs-org-step}(v) = \{u \cdot [1, i]\}$.

The function *prs-org-step* can be extended to a function *prs-org** covering multi-step reduction similar to the extension of *org-step* to *org** (see Section 3.2).

Note that a consequence of this definition is that a possible implementation should be able to recognize whether or not a particular function symbol is a Φ-function symbol.

5.4 Properties

In Section 3.4 we mentioned four desirable characteristics (labeled (A), (B), (C), and (D)) for extensions of origins. To what extend did we manage to meet these requirements? Concerning the size of the origin sets (requirements (A) and (B)), origin sets in pure PRSs always contain exactly one element:

Property 5 *Let t, t' be terms, $A : t \to^* t'$ a reduction in a PRS, and let prs-org$^*_A$ be the origin function for this reduction. For all $v \in \mathcal{O}(t')$ we have $|\text{prs-org}^*_A(v)| = 1$.*

Proof: This follows from the facts that (1) PRSs are left-linear, (2) none of the various cases for function symbol origins in PRSs overlap, and (3) because every individual case yields exactly one origin.

This means that requirement (A) to keep the origin sets small is met. Requirement (B), however, is only partly met. Although unpleasant sets containing no origins at all are excluded (which contrasts with the primary origins, see Property 2), situations were multiple origins are nice (error handling) are treated in an unsatisfactory manner.

PRS origins try achieve the proper depth of (requirements (C) and (D)), by focusing on the G-terms. Function symbols need to synthesize new values obtain an origin to the closest surrounding Φ-function. The following property states that origins established during reduction of t according to left-hand side $\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m)$, either result from copying subterms of some y_i (first case), or are the result of introducing a new function symbol in some right-hand side, and thus point to a subterm of the G-term which is the first argument of ϕ.

Property 6 *Let t, t' be terms, $t \neq t'$, let $A : t \to^* t'$ be a reduction in a PRS, and let prs-org$^*_A$ be the origin function for this reduction. Assuming that the top operator of t is a Φ-function, we have: For all $v \in \mathcal{O}(t')$, let prs-org$^*_A(v) = \{u\}$. Then either:*

- *$[j] \preceq u$, with $j \geq 2$, and $t'/v = t/u$; or*
- *$[1] \preceq u$.*

Proof: Direct from the definition of *prs-org-step* and the fact that t has a Φ-function as its top node

6. Syntax-Directed Origin Tracking

The origins in the previous section were defined for the clean PRS case. Here we study the consequences of relaxing the PRS requirements and of extending the rewrite mechanism.

6.1 Relaxing the PRS Requirements

Some requirements of the five (ii) to (vi) in Section 4 are non-restrictive, and others can be relaxed easily:

(ii) Equaions over G-terms can be useful, as in "repeat(S, E) = seq$(S,$ while$($not$(E), S))$". We can handle such equations by relating each function symbol in the right-hand side to the top of the left-hand side. Hence equations over G-terms are treated as equations over S-terms (the Auxiliary Symbols case).

(iii) The restriction to recursion over the *first* argument is not essential.

(iv) The fixed left-hand side $\phi(p(x_1, ..., x_n), y_1, ..., y_m)$ of Φ-defining equations can be annoying when writing large specifications. Deeper patterns at the x_i or y_j positions can be allowed without problems (allowing non-trivial matching at these positions).

(v) Right-hand sides of Φ-defining equations are not allowed to contain any function symbol from G. This can be a problem when writing, e.g., an operational semantics of a while loop, where the while constructor will appear in the right-hand side again. For origin tracking purposes, it is possible to link every new G-term to the $p(x_1, ..., x_n)$ node at the left-hand side. Thus, G-symbols introduced in right-hand sides of Φ-defining equations can be treated as S-symbols introduced in such right-hand side (the Synthesizer case).

(vi) Linearity of left-hand sides is not essential. Allowing non-linear patterns causes origins to contain multiple paths, which (Section 3.6, (B)) can be useful under certain circumstances.

Relaxing requirements (ii) and (v) can make some origins less precise. For instance, the "not(E)" in the right-hand side of the equation mentioned under (ii) will have an origin to the entire repeat statement.

6.2 Common Subterms

In addition to the Common Variables and Context cases, [DKT93] introduced a "Common Subterms" case.

Let $t \xrightarrow{u,\sigma}_r t'$, where r is a rule $\alpha \to \beta$, be an elementary reduction step. Let $v \in \mathcal{O}(t')$:

- (Common Subterms)

  If $v \equiv u \cdot v'$ with $v' \in \mathcal{O}_{fun}(\beta)$ the occurrence of a function symbol (or constant) in the right-hand side β of r, then

  $$\textit{org-step}(v) = \{u \cdot v'' \mid \alpha/v'' \equiv \beta/v'\}$$

  Note that common subterms are looked for in the *uninstantiated* sides α and β.

In the PRS context, common subterms can be useful in the Auxiliary Symbols case (which also applies to G-terms if (ii) is relaxed). Moreover, if the fixed patterns of left-hand sides of Φ-defining equations are allowed to contain arbitrary patterns (iv), the common subterms case could be useful to find origins S-symbols (or even G-symbols, if (v) is relaxed) occurring in the right-hand side.

6.3 Further Extensions

A detailed account of the use of PRSs is given in [Meu94]. She proposes extensions of PRSs to deal with *conditional* equations as well as with associative *lists*. Syntax-directed origins can easily be extended to deal with these mechanisms as well.

A particularly interesting topic Van der Meulen discusses is the nested or *layered* PRS. A typical example of a layered PRS is a compilation defined by a translation to an intermediate language, followed by a translation to the target language. Both translations will be defined as PRSs. Thus, the S-functions of the first PRS act as G-functions of the second PRS. Syntax-directed origins easily apply to both PRSs.

7. Concluding Remarks

We have discussed an extension of the origin function. We gave a precise definition for pure PRSs, and formulated several properties. Realizing that in practice a specification hardly ever is a pure PRS, we extended our definition such that it applies to arbitrary specifications with a syntax-directed nature. In comparison with existing origin schemes, syntax-directed origins particularly focus on providing good information for created function symbols. This allows the technique to be applied to, e.g., automatic generation of error handlers and source-level debuggers.

We can illustrate the success of syntax-directed origins by considering the type checker for ISO Pascal described in [Deu91]. For about 40% of the error messages, the primary origins are sufficient. These 40% are of the form `Not-a-variable(x)`, i.e., containing some piece of the initial program (in this case the identifier `x`) as part of the message. This piece provides the origin information. But the remaining 60% of the messages are functions like `Integer-type instead of string-type expected`, where all symbols are freshly created (not occurring in the source program). Primary origins could not give origins for these, but sytnax-directed origins can.

On the negative side, our extension is restricted to the class of PRS-like specifications. However, this class is very large. Many specifications, particularly of type checkers, evaluators, compilers, and so on, have a syntax-directed (or, equivalently, homomorphic, compositional, or inductive) character.

Another point of criticism might concern the actual definition of *prs-org*. The "Auxiliary Symbols" case may result in rather imprecise origins, since it simply relates all function symbols at the right to the top-symbol at the left. This strategy is however, at least as safe as possible since it does not lose any information. Moreover, the Auxiliary Symbols case does not form the heart of the syntax-directed origins. This is covered by the "Syntax-Directed Functions" and "Synthesizer" case. Hence, future improvements of origin tracking can be embodied in the Auxiliary Symbols case without changing the nature of the syntax-directed origins.

At the moment, we are implementing syntax-directed origin tracking within the ASF+SDF programming-environment generator [Kli93]. Primary origins are already implemented (see [DKT93]). This implementation tries to overcome the potential loss of reduction speed in the term rewriting machine in several ways. Paths are represented by pointers in directed acyclic graphs (DAGs), sets of paths are encoded by bit maps, and origins are computed by propagating these sets as annotations during rewriting. Moreover, as many computations as possible are performed at "compile time", thus reducing the overhead at reduction time. These measurements are sufficient to make origin tracking feasible in realistic specifications. The extension to syntax-directed origins is currently being implemented. This implementation is eased by the fact that parts of the implementation of *incremental* rewriting [Meu92] can be re-used.

As a preliminary study, we already finished an (algebraic) specification of the origins following the definition of Section 5.3. We were able to conduct some initial experiments using this executable specification, and observed that the syntax-directed origins behaved as expected.

Acknowledgments

I would like to thank T.B. Dinesh, Jan Heering, Paul Klint, and Emma van der Meulen for their careful reading and willingness to discuss origins and PRSs.

REFERENCES

[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* 24(2).

[Ber90] Y. Bertot. Implementation of an interpreter for a parallel language in Centaur. In N. Jones, editor, *ESOP '90 - Proceedings of the Third European Symposium on Programming*, volume 432 of *LNCS*, pages 57–69. Springer-Verlag, 1990.

[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.

[Ber93] Y. Bertot. A canonical calculus of residuals. In G. Huet and G. Plotkin, editors, *Logical Environments*. Cambridge University Press, 1993.

[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.

[Bro92] M. Broy. Experiences with software specification and verification using LP, the Larch Proof Assistant. Technical Report 93, DEC Systems Research Center, 1992. Available by *ftp* from gatekeeper.pa.dec.com: /pub/DEC/SRC/research-reports.

[CF82] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I and II. *Theoretical Computer Science*, 17:163–191 and 235–257, 1982.

[Cou90] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 459–492. Elsevier Science Publishers, 1990.

[DD93] A. van Deursen and T.B. Dinesh. Origin tracking for higher-order term rewriting systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Proceedings of the International Workshop on Higher-Order Algebra, Logic and Term Rewriting HOA93*, Amsterdam, September 1993. To Appear.

[Deu91] A. van Deursen. An algebraic specification for the static semantics of Pascal. In J. van Leeuwen, editor, *Conference Proceedings Computing Science in the Netherlands CSN'91*, pages 150–164, 1991. Full specification available by *ftp* from ftp.cwi.nl:pub/gipe.

[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.

[Fie93] J. Field. A graph reduction approach to incremental rewriting. In C. Kirchner, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 259–273, 1993.

[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 and Methodology*, 2(2):176–201, 1993.

[Klo92] 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.

[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.

[Meu88] E.A. van der Meulen. Algebraic specification of a compiler for a language with pointers. Report CS-R8848, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988.

[Meu92] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. In *Proceedings of the 2nd International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, pages 277–286. Springer-Verlag, 1992.

[Meu94] E. A. van der Meulen. *Incremental Rewriting.* PhD thesis, University of Amsterdam, 1994. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Meu94.ps.Z.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual - Third edition.* Springer-Verlag, 1989.

[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.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 675–789. Elsevier Science Publishers, 1990.