From line numbers to origins

Paul Klint

September 4, 1995

Dedicated to Frans Kruseman Aretz.

Abstract

Many aspects of computer science were first shown to me by Frans Kruseman Aretz in the period 1969–1971.

The first computer architecture I ever saw was the OEBRA (for Onbestaanbare Elektronische Binaire Rekenautomaat): a very simple machine originally designed by A. van der Sluis. Frans explained it in full detail by giving all hardware diagrams as well as an interpreter for OEBRA assembly language programs (written in Algol60). Later on he showed how a real PDP8 (with a main memory of 4K 12 bit words) could assemble programs by exclusively using paper tape as storage medium (for representing the assembler itself, the source program, the intermediate output of the assembler, and, finally, the binary version of the source program).

On another occasion we visited the machine room of the EL-X8 of the Mathematisch Centrum then located at the Tweede Boerhaevestraat 48 in Amsterdam. Acting like a professional piano player, Frans used the switch register for entering a program that would set the *complete* memory (28K 27 bit words) of the machine to zero's. Before pushing the "run" button, however, he asked each student how long it would take to execute this program. Probably due to selective amnesia, I do not remember what my guess was, but the very short flash of the lights on the console registers while executing the program impressed me deeply. This X8 was a really fast machine with its 2.5μsec cycle time (= 0.4 MHz)!

The first operating system I saw was the Milli operating system for the X8 (written by Frans Kruseman Aretz in ELAN, the assembly language of the X8). The first compiler I saw was the Algol60 compiler for the X8 (written, again, by Frans Kruseman Aretz in Algol60). His standard approach was to give a brief global overview of the issues and techniques involved and then start a meticulous explanation at the source code level of each system he was discussing.

I remember various small, but very instructive, programs he treated during his courses: for instance, a Lisp interpreter in Algol60 (highlighting variable binding and recursive evaluation), and an interpreter for Turing machines (using Algol60's call-by-name mechanism to represent the infinite tape of the Turing machine). A puzzle he gave me once was to determine the number of logical functions of three variables that can be built using at most one inverter gate.

Frans supervised my Master's thesis [Kli73] concerning a semi-automatic proof of the termination of the X8 Algol60 compiler as well as my PhD thesis [Kli82] on the design of string manipulation languages. Clearly, I owe very much to him and I am glad to contribute the following paper on the occasion of his retirement from the University of Eindhoven.

This contribution is devoted to another theme of common interest (which can, for instance, be verified by inspecting the bibliographical notes in [WG84], p323): maintaining references to a program's source text during its execution. [Kru71] deals with a technique for generating the minimal number of instructions for correctly maintaining line numbers at run time. [Kli79] deals with a technique for concisely encoding line numbers in abstract machine instructions.

I give a new formalization of *origin tracking* [vDKT93], a technique for maintaining references between the normal form and the initial term of a term rewriting process.

1 Introduction

Term rewriting systems are frequently used to execute algebraic specifications: the equations in the specification are interpreted as rewrite rules and a given initial term is reduced according to these rules;

if no further reductions are possible we have obtained an irreducible term (the *normal form*) constituting the answer of the computation.

A typical function in a specification (such as an evaluator, type checker or translator) operates on the abstract syntax tree of a program (which is part of the initial term). During the term rewriting process, pieces of the program such as identifiers, expressions, or statements, recur in intermediate terms.

In [vDKT93] we have introduced the notion of "origin tracking": by establishing reverse links from the normal form to the initial term of the term rewriting process we obtain information that is important for interactive tools like error reporters (associate an error message with a part of the source program) and animators (visualize the statement currently being executed).

The existing formalization of origin tracking proceeds in two stages. First, relations are defined for elementary reduction steps $T_i \rightarrow T_{i+1}$. Next, these relations are extended to complete reduction sequences $T_0 \rightarrow T_1 \rightarrow ... \rightarrow T_n$. In particular, relations are established between subterms of an intermediate term T_i, and subterms of the initial term T_0. The process of incrementally computing origins is called origin tracking. In Appendix B of [vD94] an ASF+SDF specification is given following along these lines. Here, I want to simplify the two stage approach by concentrating on the information that has to be propagated for elementary reduction steps only. In this way, we get a direct formalization of origin tracking itself. The basic intuition of this new approach is to attach unique labels to the initial term and to define how these labels are propagated during rewriting.

First, a straightforward formalization of term rewriting is given in Section 2 and then we extend it in Section 3 with origin tracking.

2 Term rewriting

2.1 Terms

First, we define the basic syntactic structure of terms. Observe that functions are always unary, but they may operate on lists. We assume (but dot not show) a module `Layout` containing definitions for comments and white space.

Module BasicTerms
imports Layout$^{(2.1)}$
exports
  sorts FUN VAR TERM
  lexical syntax
    [a-z][A-Za-z0-9\—]* → FUN
    [A-Z][A-Za-z0-9\—]* → VAR
  context-free syntax
    VAR → TERM
    nil → TERM
    TERM ";" TERM → TERM {**right**}
    FUN "(" TERM ")" → TERM
    "(" TERM ")" → TERM {**bracket**}
  variables
    Var [*0-9'*]* → VAR
    Fun [*0-9'*]* → FUN
    T [*0-9'*]* → TERM
equations
Ensure that lists are always right-associative.

$$(T_1;\ T_2);\ T_3 \ =\ T_1;\ T_2;\ T_3 \qquad \text{[list-1]}$$

Next, we introduce functions for classification, selection (i.e., decomposition) and replacement of terms. We assume (but do not show) a definition of the module `Booleans`.

Module Terms

imports BasicTerms$^{(2.1)}$ Booleans$^{(2.1)}$
exports
context-free syntax

is-fun(TERM)	→ BOOL
is-non-empty-list(TERM)	→ BOOL
is-nil(TERM)	→ BOOL
fun(TERM)	→ FUN
arg(TERM)	→ TERM
head(TERM)	→ TERM
tail(TERM)	→ TERM
repl-arg(TERM, TERM)	→ TERM
repl-list(TERM, TERM, TERM)	→ TERM

equations

Classification functions on terms.

is-fun(*Fun*(*T*)) = true [is-fun-1]
is-fun(*T*) = false **otherwise** [is-fun-2]

is-non-empty-list(T_1; T_2) = true [is-non-empty-list-1]
is-non-empty-list(*T*) = false **otherwise** [is-non-empty-list-2]

is-nil(nil) = true [is-nil-1]
is-nil(*T*) = false **otherwise** [is-nil-2]

Selection functions on terms.

fun(*Fun*(*T*)) = *Fun* [fun-1]
fun(*T*) = `error` **otherwise** [fun-2]

arg(*Fun*(*T*)) = *T* [arg-1]
arg(*T*) = nil **otherwise** [arg-2]

head(T_1; T_2) = T_1 [head-1]
head(*T*) = nil **otherwise** [head-1]

tail(T_1; T_2) = T_2 [tail-1]
tail(*T*) = nil **otherwise** [tail-2]

Replacement functions on terms. Given a function application (or a list) and a new argument (or two new list elements) construct a new function application (or list). In this manner replacement operations are made explicit and can be extended later on (see Section 3.2).

repl-arg(*Fun*(T_1), T_2) = *Fun*(T_2) [repl-arg-1]
repl-arg(T_1, T_2) = T_1 **otherwise** [repl-arg-2]

repl-list(T_1; T_2, T_1', T_2') = T_1'; T_2' [repl-list-1]
repl-list(*T*, T_1', T_2') = T_1'; T_2' **otherwise** [repl-list-2]

The definition of a replacement function for complete terms (`repl-term`) will be given in Section 2.4.

2.2 Substitutions

Substitutions define a mapping from variables to terms and are represented as lists of the form $[Var_1 \rightarrow Term_1, ..., Var_n \rightarrow Term_n]$. Three operations are defined for substitutions: composition of two substitutions ($\sigma_1 \circ \sigma_2$), application of a substitution σ to a term T (T^σ), and checking that a variable Var is in the domain of a substitution σ ($Var \in \sigma$).

Module Substitutions
imports Terms(2.1) Booleans(2.1)
exports
sorts SUBSTITUTION ONE-SUBS
context-free syntax

VAR ↦ TERM	→ ONE-SUBS
"[" {ONE-SUBS ","}* "]"	→ SUBSTITUTION
SUBSTITUTION ∘ SUBSTITUTION	→ SUBSTITUTION
TERM "^" SUBSTITUTION	→ TERM
VAR ∈ SUBSTITUTION	→ BOOL
SUBSTITUTION "(" VAR ")"	→ TERM

exports
variables

Subs [0-9']*"*"	→ {ONE-SUBS ","}*
σ [0-9']*	→ SUBSTITUTION

equations

Composition of substitutions.

$$[Subs_1^*] \circ [Subs_2^*] = [Subs_1^*, Subs_2^*] \quad \text{[c-1]}$$

Variable occurs in substitution.

$$Var \in [Subs_1^*, Var \mapsto T, Subs_2^*] = \text{true} \quad \text{[in-1]}$$

$$Var \in \sigma = \text{false} \quad \textbf{otherwise} \quad \text{[in-2]}$$

Retrieve variable associated with a variable in a given substitution.

$$[Var \mapsto T, Subs^*](Var) = T \quad \text{[sv-1]}$$

$$Var_1 \neq Var_2 \Rightarrow [Var_1 \mapsto T, Subs^*](Var_2) = [Subs^*](Var_2) \quad \text{[sv-2]}$$

$$[](Var) = Var \quad \text{[sv-3]}$$

Apply substitution to a term.

$$Var \in \sigma = \text{true} \Rightarrow Var^\sigma = \sigma(Var) \quad \text{[as-1]}$$

$$Var \in \sigma = \text{false} \Rightarrow Var^\sigma = Var \quad \text{[as-2]}$$

$$\text{is-fun}(T) = \text{true} \Rightarrow T^\sigma = \text{repl-arg}(T, \text{arg}(T)^\sigma) \quad \text{[as-3]}$$

$$\text{is-non-empty-list}(T) = \text{true} \Rightarrow T^\sigma = \text{repl-list}(T, \text{head}(T)^\sigma, \text{tail}(T)^\sigma) \quad \text{[as-4]}$$

$$\text{is-nil}(T) = \text{true} \Rightarrow T^\sigma = T \quad \text{[as-5]}$$

2.3 Matching

Matching two terms T_1 and T_2 yields zero or more substitutions $\{\sigma_1, \sigma_2, ...\}$ such that $T_1^{\sigma_i} \equiv T_2$ $(i = 1, 2, ...)$ holds. If a match yields $\{\}$ (the empty set of substitutions), terms T_1 and T_2 do not match.

For the current paper it would suffice to introduce a unitary matching function that either succeeds (yielding a single substitution) or fails. The definition given here is more general and can also handle (non-unitary) list matching.

Module Match

imports Substitutions$^{(2.2)}$

exports

sorts MATCH

context-free syntax

SUBSTITUTION → MATCH
MATCH & MATCH → MATCH
"(" MATCH ")" → MATCH {**bracket**}
"{" {MATCH ","}* "}" → MATCH

match(TERM, TERM) → MATCH

variables

μ [0-9']* → MATCH
μ [0-9']*"*" → {MATCH ","}*
μ [0-9']*"+" → {MATCH ","}+

equations

Nested lists of matches may be flattened.

$$\{\mu_1^*, \{\mu_2^*\}, \mu_3^*\} = \{\mu_1^*, \mu_2^*, \mu_3^*\} \quad \text{[lm-1]}$$

Determine the "and" of two matches. The equations for matches consisting of a single substitution are:

$$\frac{Var \in \sigma = \mathsf{true},\ \sigma(Var) = T}{[Var \mapsto T, Subs^*] \ \& \ \sigma = [Subs^*] \ \& \ \sigma} \quad \text{[mand1]}$$

$$\frac{Var \in \sigma = \mathsf{true},\ \sigma(Var) \neq T}{[Var \mapsto T, Subs^*] \ \& \ \sigma = \{\}} \quad \text{[mand2]}$$

$$\frac{Var \in \sigma_1 = \mathsf{false},\ [Subs^*] \ \& \ \sigma_1 = \sigma_2}{[Var \mapsto T, Subs^*] \ \& \ \sigma_1 = [Var \mapsto T] \circ \sigma_2} \quad \text{[mand3]}$$

$$[] \ \& \ \sigma = \sigma \quad \text{[mand4]}$$

$$\sigma \ \& \ [] = \sigma \quad \text{[mand5]}$$

The equations for matches consisting of a list of substitutions are:

$$\mu \ \& \ \{\} = \{\} \quad \text{[mand-6]}$$

$$\{\} \ \& \ \mu = \{\} \quad \text{[mand-7]}$$

$$\{\mu_1\} \ \& \ \mu_2 = \{\mu_1 \ \& \ \mu_2\} \quad \text{[mand-8]}$$

$$\mu_1 \ \& \ \{\mu_2\} = \{\mu_1 \ \& \ \mu_2\} \quad \text{[mand-9]}$$

$$\{\mu_1, \mu_2^*\} \ \& \ \mu_3 = \{\mu_1 \ \& \ \mu_3, \{\mu_2^*\} \ \& \ \mu_3\} \quad \text{[mand-10]}$$

$$\mu_1 \ \& \ \{\mu_2, \mu_3^*\} = \{\mu_1 \ \& \ \mu_2, \mu_1 \ \& \ \{\mu_3^*\}\} \quad \text{[mand-11]}$$

Match terms T_1 and T_2.

$$\mathsf{match}(\mathit{Var},\ T)\ =\ \{[\mathit{Var} \mapsto T]\} \qquad \text{[match-1]}$$

$$\frac{\mathsf{is\text{-}fun}(T_1) = \mathsf{true},\ \mathsf{is\text{-}fun}(T_2) = \mathsf{true},\ \mathsf{fun}(T_1) = \mathsf{fun}(T_2)}{\mathsf{match}(T_1,\ T_2) = \mathsf{match}(\mathsf{arg}(T_1), \mathsf{arg}(T_2))} \qquad \text{[match-2]}$$

$$\frac{\mathsf{is\text{-}nil}(T_1) = \mathsf{true},\ \mathsf{is\text{-}nil}(T_2) = \mathsf{true}}{\mathsf{match}(T_1,\ T_2) = \{[]\}} \qquad \text{[match-3]}$$

$$\frac{\mathsf{is\text{-}non\text{-}empty\text{-}list}(T_1) = \mathsf{true},\ \mathsf{is\text{-}non\text{-}empty\text{-}list}(T_2) = \mathsf{true}}{\mathsf{match}(T_1,\ T_2) = \mathsf{match}(\mathsf{head}(T_1), \mathsf{head}(T_2))\ \&\ \mathsf{match}(\mathsf{tail}(T_1), \mathsf{tail}(T_2))} \qquad \text{[match-4]}$$

$$\mathsf{match}(T_1,\ T_2)\ =\ \{\}\quad \textbf{otherwise} \qquad \text{[match-5]}$$

2.4 Rewrite Rules

Rewrite rules (in their simplest form) look like $T_1 \to T_2$. We introduce the decomposition functions `lhs` and `rhs` to retrieve the sides of a rule and we define term replacement.

Module Rules

imports Terms(2.1) Substitutions(2.2)
exports
sorts RULE TRS
context-free syntax

TERM "→" TERM	→ RULE
lhs(RULE)	→ TERM
rhs(RULE)	→ TERM
"{" {RULE ","}* "}"	→ TRS
repl-term(TERM, SUBSTITUTION, RULE)	→ TERM

variables

Rule	→ RULE
Rule [0-9']* "*"	→ {RULE ","}*
R [0-9']* "*"	→ {RULE ","}*
TRS [0-9']*	→ TRS

equations

Selector functions for left hand side and right hand side of a rule.

$$\mathsf{lhs}(T_1 \to T_2)\ =\ T_1 \qquad \text{[lhs-1]}$$

$$\mathsf{rhs}(T_1 \to T_2)\ =\ T_2 \qquad \text{[rhs-1]}$$

Replace a term by an instantiated right-hand side of a rule. This function will be used to replace redexes during rewriting. It is parametrized with a complete rule as opposed to, for instance, only the right hand side of a rule, in order to provide all information that may be relevant for more advanced replacement operations in which the syntactic structure of the complete rule may determine the propagation of certain additional information during replacement. This becomes relevant when extending term replacement for more sophisticated forms of origin tracking.

$$\mathsf{repl\text{-}term}(T, \sigma, \mathit{Rule})\ =\ \mathsf{rhs}(\mathit{Rule})^{\ \sigma}\quad \textbf{otherwise} \qquad \text{[repl-term-1]}$$

2.5 Term Rewriting Systems

Given a term T and a term rewriting system TRS, term rewriting amounts to constructing the sequence of terms $T \equiv T_0 \rightarrow T_1 \rightarrow ... \rightarrow T_n$ such that for each step $T_i \rightarrow T_{i+1}$ the following holds:

- in T_i occurs a subterm T,
- TRS contains a rule $Rule$,
- $match(T, lhs(Rule)) = \{\sigma_1, ..., \sigma_m\}$ $(m \geq 0)$,
- $T' = rhs(Rule)^{\sigma_j}$ (for some σ_j, $0 \leq j \leq m$), and
- $T_{i+1} = T_i[T := T']$ (i.e., the occurrence of T in T_i is replaced by T').

In addition, T_n should be a normal form. In the following specification we use the artifact of a "step": either the constant `nostep` if no step is possible or a pair of the form $[Rule, \mu]$, where $Rule$ is a rewrite rule and μ is a match.

Note that the following definitions can be generalized in several directions to take into account:

- the ordering that is used when searching for applicable rules (e.g., random, textual, or specificity of left hand sides);
- reduction strategy (e.g., innermost, outermost, mixed);
- more advanced features in rules (e.g., list variables, conditions).

Here, we will use textual ordering, innermost reduction, and unconditional rules.

Module TRS

imports Match(2.3) Rules(2.4)
exports
sorts STEP
context-free syntax

"[" RULE "," MATCH "]"	→ STEP
nostep	→ STEP
normalize(TERM, TRS)	→ TERM
normalize1(TERM, TERM, TRS)	→ TERM
normalize-args(TERM, TRS)	→ TERM
find-rule(TERM, TRS, TRS)	→ STEP
apply-rule(TERM, RULE, MATCH, TRS)	→ STEP

equations

Normalize a term to normal form. If it is a non-empty list, normalize the list elements.

$$\frac{\text{is-non-empty-list}(T) = \text{true}}{\text{normalize}(T, \text{TRS}) = \text{repl-list}(T, \text{normalize}(\text{head}(T), \text{TRS}), \text{normalize}(\text{tail}(T), \text{TRS}))} \quad \text{[n1]}$$

Else, if there is a matching rule, apply it and continue normalization. Observe that matters related to the rewriting strategy (e.g., normalization of function arguments) are delegated to `find-rule`.

$$\frac{\begin{array}{c}\text{is-non-empty-list}(T) = \text{false}, \\ \text{find-rule}(T, \text{TRS}, \text{TRS}) = [Rule, \{\mu_1^*, \sigma, \mu_2^*\}]\end{array}}{\text{normalize}(T, \text{TRS}) = \text{normalize}(\text{repl-term}(T, \sigma, Rule), \text{TRS})} \quad \text{[n2]}$$

Otherwise, first normalize the term's arguments and retry normalization.

$$\text{normalize}(T, \text{TRS}) = \text{normalize1}(T, \text{normalize-args}(T, \text{TRS}), \text{TRS}) \quad \textbf{otherwise} \quad \text{[n3]}$$

Normalization halts if the previous term is equal to the current one; otherwise normalization is continued.

$$\text{normalize1}(T,\ T,\ \text{TRS}) \ = \ T \qquad \text{[n1-1]}$$

$$\text{normalize1}(T,\ T',\ \text{TRS}) \ = \ \text{normalize}(T',\ \text{TRS}) \quad \textbf{otherwise} \qquad \text{[n1-2]}$$

Normalize the arguments of a term.

$$\frac{\text{is-fun}(T) = \text{true}}{\text{normalize-args}(T,\ \text{TRS}) = \text{repl-arg}(T,\ \text{normalize}(\text{arg}(T),\ \text{TRS}))} \qquad \text{[na-1]}$$

$$\frac{\text{is-non-empty-list}(T) = \text{true}}{\text{normalize-args}(T,\ \text{TRS}) = \text{repl-list}(T,\ \text{normalize}(\text{head}(T),\ \text{TRS}),\ \text{normalize}(\text{tail}(T),\ \text{TRS}))} \qquad \text{[na-2]}$$

$$\text{normalize-args}(T,\ \text{TRS}) = T \qquad \textbf{otherwise} \qquad \text{[na-3]}$$

Find a rule that can be applied to a given term. The first rule that can be applied is used.

$$\frac{\text{apply-rule}(T,\ \mathit{Rule},\ \{\text{match}(\text{lhs}(\mathit{Rule}),\ T)\},\ \text{TRS}) = [\mathit{Rule},\ \mu]}{\text{find-rule}(T,\ \text{TRS},\ \{\mathit{Rule},\ \mathit{Rule}^*\}) = [\mathit{Rule},\ \mu]} \qquad \text{[fr-1]}$$

$$\frac{\text{apply-rule}(T,\ \mathit{Rule},\ \{\text{match}(\text{lhs}(\mathit{Rule}),\ T)\},\ \text{TRS}) = \text{nostep}}{\text{find-rule}(T,\ \text{TRS},\ \{\mathit{Rule},\ \mathit{Rule}^*\}) = \text{find-rule}(T,\ \text{TRS},\ \{\mathit{Rule}^*\})} \qquad \text{[fr-2]}$$

$$\text{find-rule}(T,\ \text{TRS},\ \{\}) \ = \ \text{nostep} \qquad \text{[fr-3]}$$

Apply a rule to a term. Given a term T and a rule *Rule*, can *Rule* be used to reduce T? Here, we will use a fixed, innermost strategy, but these definitions can easily be extended to cover other strategies as well.

$$\frac{\begin{array}{c}T' = \text{normalize-args}(T,\ \text{TRS}),\\ \text{match}(\text{lhs}(\mathit{Rule}),\ T') = \{\mu_1^+\}\end{array}}{\text{apply-rule}(T,\ \mathit{Rule},\ \{\mu^+\},\ \text{TRS}) = [\mathit{Rule},\ \{\mu_1^+\}]} \qquad \text{[ar-1]}$$

$$\text{apply-rule}(T,\ \mathit{Rule},\ \mu,\ \text{TRS}) \ = \ \text{nostep} \quad \textbf{otherwise} \qquad \text{[ar-3]}$$

2.6 An example: list reversal

Consider the reversal of a list of two elements, defined as follows (taken from [vD94]):

```
normalize(
 rev(cons(one(nil) ;cons(two(nil);cons(two(nil);null(nil))))),
 { rev(null(nil))        -> null(nil),
   rev(cons(E;L))        -> append(rev(L);cons(E;null(nil))),
   append(null(nil);L)   -> L,
   append(cons(E;L1);L2) -> cons(E;append(L1;L2))
 }
)
```

Note that our term syntax does not support constants. As a result, all constants have to be written as functions applied to an empty argument list: we have to write `one(nil)`, `two(nil)`, and `null(nil)` instead of `one`, `two` and `null`.

The above initial term will yield the normal form:

```
cons(two(nil) ; cons(two(nil) ; cons(one(nil) ; null(nil))))
```

3 Origin Tracking

3.1 Background

As already explained in the introduction, origin tracking amounts to establishing reverse links between the normal form and the initial term of a term rewriting process. In [vDKT93], all rewriting steps $T_0 \rightarrow T_1 \rightarrow ... \rightarrow T_n$ are taken into account and *backward* relations are established between the normal form T_n and the initial term T_0. Four relations are distinguished: *Common Variables*, *Redex-Contractum*, *Context*, and *Common Subterms*.

Here, we introduce a labeling of the initial term and define how these are propagated in the forward direction during rewriting. Labels appearing in the normal form correspond to the desired origin information. The presentation proceeds in three steps. First, we introduce the notion of *origin sets*: sets of labels of the form $\{L_1, L_2, ...\}$. Next, we extend the syntactic form of terms in order to permit the use of origin sets as labels for (sub)terms. Typically, a term `f(a;g(b))` may be labeled as follows: `{lab1}: f(a;g({lab2,lab3}: b))`. Finally, we extend the definitions of the access functions for terms in order to properly preserve origin sets.

It turns out that in this new approach the relations *Common Variables*, and *Context* become implicit since they are taken care of by the labeling. For reasons of simplicity (and space), we will not introduce the *Common Subterms* relation. Using the terminology of [vD94], we therefore restrict the presentation to *primary origins* only.

3.2 Specification of origin tracking

Module Origins
imports $\text{TRS}^{(2,5)}$
exports
sorts ORG-SET
context-free syntax

"{" {FUN ","}* "}"	→ ORG-SET
ORG-SET "∪" ORG-SET	→ ORG-SET
ORG-SET ":" TERM	→ TERM
org-set(TERM)	→ ORG-SET
propagate(TERM, ORG-SET)	→ TERM
collect(TERM)	→ ORG-SET
collect1(TERM)	→ ORG-SET

priorities
TERM ";" TERM → TERM < ORG-SET ":" TERM → TERM

hiddens
variables

O [0-9']*	→ ORG-SET
O [0-9']* "*"	→ {FUN ","}*

equations
Define origin sets as sets of function symbols.

$$\{O_1^*\} \cup \{O_2^*\} = \{O_1^*, O_2^*\} \quad \text{[un-1]}$$
$$\{O_1^*, Fun, O_2^*, Fun, O_3^*\} = \{O_1^*, Fun, O_2^*, O_3^*\} \quad \text{[un-2]}$$

Define the cases that an empty origin set, respectively a multiple origin sets, is associated with a term.

$$\{\} : T = T \quad \text{[empty-org]}$$
$$O_1 : O_2 : T = O_1 \cup O_2 : T \quad \text{[dup-org]}$$

Extend the classification functions on terms.

$$\text{is-fun}(O : T) = \text{is-fun}(T) \quad \text{[is-fun-3]}$$
$$\text{is-non-empty-list}(O : T) = \text{is-non-empty-list}(T) \quad \text{[is-list-3]}$$
$$\text{is-nil}(O : T) = \text{is-nil}(T) \quad \text{[is-nil-3]}$$

Extend the selection functions on terms.

$$\text{fun}(O : T) = \text{fun}(T) \quad \text{[fun-3]}$$
$$\text{arg}(O : T) = \text{arg}(T) \quad \text{[arg-3]}$$
$$\text{head}(O : T) = \text{head}(T) \quad \text{[head-3]}$$
$$\text{tail}(O : T) = \text{tail}(T) \quad \text{[tail-3]}$$

Extend the replacement functions on terms.

$$\text{repl-arg}(O : T_1, T_2) = O : \text{repl-arg}(T_1, T_2) \quad \text{[repl-arg-2]}$$

$$\text{repl-list}(O : T_1, T_2, T_3) = O : \text{repl-list}(T_1, T_2, T_3) \quad \text{[repl-list-2]}$$

Replace a term T by an instantiated right-hand side of a rule. First, collect the origin sets associated with T and then propagate them to the instantiated right-hand side.

$$\text{repl-term}(T, \sigma, \mathit{Rule}) = \text{propagate}(\text{rhs}(\mathit{Rule})^{\sigma}, \text{collect}(T)) \quad \text{[repl-term-2]}$$

Retrieve the origin set directly associated with a term. If no origin set is associated with it, return the empty set.

$$\text{org-set}(O : T) = O \quad \text{[org-set-1]}$$
$$\text{org-set}(T) = \{\} \quad \textbf{otherwise} \quad \text{[org-set-2]}$$

Determine the origin set of a redex. If an origin set is attached at the outermost level, return it.

$$\text{collect}(O : T) = O \quad \text{[coll-1]}$$
$$\text{collect}(T) = \text{collect1}(T) \quad \textbf{otherwise} \quad \text{[coll-2]}$$

Otherwise, return the origin information attached to embedded function arguments or list elements.

$$\frac{\text{is-fun}(T) = \text{true}}{\text{collect1}(T) = \text{org-set}(\text{arg}(T))} \quad \text{[coll1-1]}$$

$$\frac{\text{is-non-empty-list}(T) = \text{true}}{\text{collect1}(T) = \text{org-set}(\text{head}(T)) \cup \text{org-set}(\text{tail}(T))} \quad \text{[coll1-2]}$$

$$\text{collect1}(T) = \{\} \quad \textbf{otherwise} \quad \text{[coll1-3]}$$

Propagate a given origin set to the arguments or list elements in a term.

$$\frac{\text{is-fun}(T) = \text{true}}{\text{propagate}(T, O) = O : \text{org-set}(T) : \text{fun}(T)(\text{propagate}(\text{arg}(T), O))} \quad \text{[prop-1]}$$

$$\frac{\text{is-non-empty-list}(T) = \text{true}}{\text{propagate}(T, O) = \text{org-set}(T) : (\text{propagate}(\text{head}(T), O); \text{propagate}(\text{tail}(T), O))} \quad \text{[prop-2]}$$

$$\text{propagate}(T, O) = T \quad \textbf{otherwise} \quad \text{[prop-3]}$$

3.3 Example: list reversal with origin sets

Applying the same rewrite rules as in Section 2.6 to a labeled term

```
normalize(
  rev(cons({a}:one(nil) ;cons({b}:two(nil);cons({c}:two(nil);null(nil))))),
 { rev(null(nil))        -> null(nil),
   rev(cons(E;L))        -> append(rev(L);cons(E;null(nil))),
   append(null(nil);L)   -> L,
   append(cons(E;L1);L2) -> cons(E;append(L1;L2))
 }
)
```

will yield an appropriately labeled normal form:

```
cons({c}:two(nil) ; cons({b}:two(nil) ; cons({a}:one(nil) ; null(nil))))
```

By removing all origin sets we obtain the same normal form as yielded by ordinary rewriting. Also observe that with ordinary rewriting the two occurrences of the constant `two(nil)` could not be distinguished. Using origin tracking, the different origins of these two constants are now explicitly indicated in the normal form.

4 Discussion

This paper presents origin tracking as a straightforward extension of ordinary term rewriting. Clearly, many issues have not been discussed here (e.g., rewriting strategies, conditional rules). However, the approach has several merits:

- The definition of origin tracking is much simpler than the one given in earlier papers.
- It provides a starting point for studying different origin propagation rules.
- It gives guidance to an implementation of origin tracking.

Acknowledgements

Appendix B of [vD94] formed the starting point for this exercise. Arie van Deursen commented on a draft of this paper.

References

[Kli73] P. Klint. Enumerability and termination. Technical report, University of Amsterdam, 1973.

[Kli79] P. Klint. Line numbers made cheap. *Communications of the ACM*, 22:557–559, 1979.

[Kli82] P. Klint. *From Spring to Summer – Design, Definition, and Implementation of Programming Languages for String Manipulation and Pattern Matching.* PhD thesis, Technical University Eindhoven, 1982.

[Kru71] F.E.J. Kruseman Aretz. On the bookkeeping of source-text line numbers during the execution phase of ALGOL 60 programs. In *MC-25 Informatica Symposium*, volume 37 of *Mathematical Centre Tracts*, pages 6.1–6.12, 1971.

[vD94] A. van Deursen. *Executable Language Definitions – Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, Programming Research Group, 1994.

[vDKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.

[WG84] W.M. Waite and G. Goos. *Compiler Construction*. Springer, 1984.