Programming by Term Rewriting

Lee A. Barnett   David A. Plaisted

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC

{lbarnett,plaisted}@cs.unc.edu

Term rewriting systems have a simple syntax and semantics and allow for straightforward proofs of correctness. However, they lack the efficiency of imperative programming languages. We define a term rewriting programming style that is designed to be translatable into efficient imperative languages, to obtain proofs of correctness together with efficient execution. This language is designed to facilitate translations into correct programs in imperative languages with assignment statements, iteration, recursion, arrays, pointers, and side effects. Versions of the language with and without destructive operations such as array assignment are developed, and the relationships between them are sketched. General properties of a translation that suffice to prove correctness of translated programs from correctness of rewrite programs are presented.

1 Introduction

The development of large-scale software systems often includes finding and resolving possibly a large number of bugs. While software testing can sometimes detect the presence of bugs, in general it cannot be used to prove that software is entirely correct with respect to its intended purpose. In systems, or components of systems, where such correctness is critical, such as in systems controlling transportation infrastructure or medical devices, it is crucial that the underlying algorithms are formally verified; proven to meet their specifications. A number of methods for formally verifying software, and creating formally verified programs, have been developed [3]. While the use of formal verification is on the rise, many software projects are still designed and deployed without undergoing verification. One possible reason is that many verification tools may need extensive specialized knowledge to use, be incompatible with some software packages, require that a specific programming language be used, or in some way add a large amount of time to the software development process.

Term rewriting systems [1] provide a formalism for expressing computational rules with a simple syntax and semantics, allowing for properties of these rules to be investigated in a natural way. This paper presents an abstract programming system, based on term rewriting, which is designed to facilitate both proofs of correctness and translation of programs into mainstream imperative programming languages. The system allows programs to be specified in a way which is close to logical notation, making possible straightforward techniques for proving correctness with respect to specifications, but is similar to an imperative language in style, so that programs can be translated more directly into imperative programming languages. The system is untyped, and designed to be simple and have few features, in order to be both easier for the typical programmer to understand and use, and more easily translatable into other languages. Complex operations such as AC-unification and higher-order unification are not included for this reason.

The purpose of such a system is to reduce the cost of formal verification by focusing on the correctness of algorithms which may exist as components of many different software systems, implemented in many different languages. Instead of having to prove the correctness of such an algorithm in each system,

Submitted to:
TERMGRAPH 2018

each of which may require a different verification tool, an abstract program expressing the algorithm can be proved correct once, and then translated into each different language for use in each software system. To this end, abstract programs which have been proved correct could be included in a publicly-available library of abstract programs, each of which can be translated into correct programs in different languages.

This paper focuses on the syntax for expressing programs in this formalism, and general properties of the programs and their translations that suffice to prove correctness of translated programs from the correctness of rewrite programs. At this point, only programs expressing pure algorithms are considered, and a detailed consideration of the translation mechanism is not presented.

2 Syntax

Standard terminology for term rewriting systems is assumed. The symbols f,g,h are used to represent function symbols from a signature Σ. Symbols a,b,c are used for constants, while x,y,z are reserved for variables from a countably infinite collection V of variables. We use $Ter(\Sigma,V)$ to refer to the set of terms over Σ and V; the symbols r,s,t,u,v,w are used to denote terms. The ground terms over Σ are elements of $Ter(\Sigma,\emptyset)$.

2.1 Constructor Term Rewriting Systems

The signature Σ is assumed to be partitioned into a set C of *constructor symbols* or *constructors*, and a set D of *defined functions* or *non-constructors*. The *constructor terms* are the elements of $Ter(C,V)$. A *constructor term rewriting system* (or CTRS) R over Σ is defined as a set of rewrite rules $r \to s$, such that all variables occurring in s also occur in r, and the left-hand side r of each rule has the form $f(t_1,\ldots,t_n)$, for $f \in D$ and $t_1,\ldots,t_n \in Ter(C,V)$.

If R is a CTRS, then R can be viewed as expressing a functional program, with elements of C corresponding to program data, and elements of D to functions. In particular, a *procedure* for $f \in D$ is a subset of rules of R for which f occurs in the left-hand side of each rule. If a procedure consists of all such rules in R, then it is called the *R-procedure* for f. The program described by R then is given as a collection of procedures, as is typical in the declarative programming paradigm.

2.2 COC Systems

The abstract programming system described in this paper specifies a program as a collection of CTRS procedures, but is designed for such a program to be translated into imperative programming languages. As such, additional properties are required of a CTRS used for this purpose in order to facilitate the translation. Corresponding to the lack of repeated formal parameters in imperative languages, each rule $r \to s$ in R is required to be *left-linear*, meaning that no variable occurs in r more than once. So that each term can be rewritten in at most one way, any pair of rules $r_1 \to s_1$ and $r_2 \to s_2$ in R are required to be *disjoint*, meaning that r_1 and r_2 are not unifiable. A CTRS which satisfies these properties is called *orthogonal*.

A system R is *complete* if, for all $f \in D$ which occur in rules of R, and all $t_1,\ldots,t_n \in Ter(C,\emptyset)$, the term $f(t_1,\ldots,t_n)$ is reducible by the R-procedure for f. In a complete CTRS, all non-constructor ground terms are reducible, and as a consequence, all normal forms of ground terms are constructor terms. This corresponds to the fact that all inputs can be processed by programs written in a typical imperative language.

If R is a CTRS specifying a program to be translated into an imperative language, then R should satisfy these properties; this way, the program given by R is structurally closer to an imperative program. These requirements are summed up by the following definition.

Definition 2.1 *A* COC system *is a CTRS that is complete and orthogonal.*

All COC systems are confluent, as a consequence of orthogonality. The following proposition results from these properties.

Proposition 2.2 *If R is a COC system and R is terminating, then each ground term s has a unique normal form $t \in Ter(C, \emptyset)$.*

2.3 Programs

We can now define a program in this formalism.

Definition 2.3 *A* program *is a COC system which consists of a finite collection of procedures.*

In other words, a COC system P is a program if P can be partitioned into finitely many procedures for $f \in D$. The letters P, Q, R will denote programs.

The semantics of such a program can be succinctly described. Let $R^=$ denote the set of equations $r = s$ for all rules $r \to s$ in any collection of rewrite rules R. If P is a program then the *declarative semantics* $\mathscr{D}(P)$ of P is $P^=$. It is easily seen that if $s \overset{*}{\Rightarrow}_R t$ then $\mathscr{D}(P) \models (s = t)$.

As a result of the properties of COC systems given above, and the fact that constructor terms are irreducible, we have the following result.

Theorem 1 *For a program P, there are no two distinct $s, t \in Ter(C, \emptyset)$ such that $\mathscr{D}(P) \models (s = t)$.*

Proof. This follows from the confluence of P, and from the fact that all constructor terms are irreducible.

3 Examples

In this section, examples are given to show how programs can be specified in this formalism. We assume rewrite rules for the binary Boolean operators $\wedge$, $\vee$, and $\neg$ are given, which will be used in infix notation. The set C of constructors is assumed to include constants $true$ and $false$, as well as tupling operators $\langle\ \rangle$.

3.1 Procedures

The following procedures are given to be included in programs where they are convenient. In the following, i, m, and n are positive integers, $f, g \in C$ are constructors, $c, d \in C$ are distinct constructor constants, and for each $f \in C$ there is a unique constant $c_f \in C$.

If-then-else (in infix notation):	Top:
(if $true$ then x else y) $\to x$	$\text{top}(c) \to c$
(if $false$ then x else y) $\to y$	$\text{top}(f(x_1, \ldots, x_n)) \to c_f$

Eq:

$\text{eq}(c, c) \to true$

$\text{eq}(c, d) \to false$

$$\text{eq}(f(x_1, \ldots, x_m), g(y_1, \ldots, y_n)) \quad \to \quad m = n \wedge \text{eq}(\text{top}(f(x_1, \ldots, x_m)), \text{top}(g(y_1, \ldots, y_n))) \wedge \text{eq}(x_1, y_1) \wedge \cdots \wedge \text{eq}(x_m, y_m)$$

<u>Arg</u>: $arg(i, f(x_1, \ldots, x_i, \ldots, x_n)) \to x_i$

<u>Replace</u>: $replace(f(x_1, \ldots, x_i, \ldots, x_n), i, y) \to f(x_1, \ldots, y, \ldots, x_n)$

Other rules are included as well; for these, if the input is not of the expected term, then the result will be a constructor constant *error* indicating that an error has occurred. Also, multiple instances of some of the above rules are necessary. In particular, one instance of the third rule for eq above is included for each pair of constructors f and g, where at least one of f and g is not a constant. One instance of the rule for arg above is given for each possible i, and one instance of the rule for replace above is given for each pair of non-constant $f \in C$ and i.

3.2 Programs

In the following programs, if the input is not of the expected term, then the result will be a constructor constant *error*, indicating that an error has occurred.

List Append. The append function on lists can be written as follows, where $cons(x, y)$ is the list y with x added to the front, and NIL is the empty list:

$$append(cons(u, v), w) \to cons(u, append(v, w))$$
$$append(NIL, w) \to w$$

For this program, the only constructors needed are *cons* and *NIL*. If there are other constructors, then *append* would also have to be given a definition on them, to achieve completeness. If sorted term rewriting were used, then it would be sufficient to specify that the arguments of *append* must be of the appropriate sort.

List Length. Following is a program that computes the length of a list:

$$length(cons(u, v)) \to 1 + length(v)$$
$$length(NIL) \to 0$$

Binary Search. Here is the essential part of a binary search program, omitting arithmetic functions. Here i and j give the lower and upper bounds of the search, x is the array being searched, and y is the element being looked for. The program returns a pair $\langle b, k\rangle$ where b is $true$ if the element is found, $false$ otherwise, and k gives the location of the element if it is found:

```
$bsearch(i, j, x, y) \to$
  if $i > j$ then $\langle \text{false}, \_\_ \rangle$ else
    if eq$(i, j)$ then (if eq(arg$(i, x), y)$ then $\langle true, i \rangle$
      else $\langle false, \_\_ \rangle$) else
    if $y < arg(\lfloor (i+j)/2 \rfloor, x)$
      then $bsearch(i, \lfloor (i+j)/2 \rfloor, x, y)$
      else $bsearch(\lfloor (i+j)/2 \rfloor, j, x, y)$
```

4 Computation

A program in a typical imperative language will specify explicitly the order in which steps are executed, but programs expressed in the formalism presented here are essentially declarative. To make translation of such programs into an imperative language more direct, the order in which rules of a program should be applied can be specified, so that execution of the translated program can mirror the chosen order.

4.1 Evaluation Strategies

A *computation sequence* or *execution sequence* through P is a sequence $t_1, t_2, t_3 \ldots$ of terms such that the *starting term* t_1 is ground, and $t_i \Rightarrow_P t_{i+1}$ for all i. Each term t_i is called a *snapshot* of the sequence. Since the starting term is required to be ground, all snapshots in a sequence are ground. The set of computation sequences through P is written $\mathscr{C}_P$. If P is terminating, then all computation sequences through P are finite, and the final term t_n in each sequence is irreducible.

Definition 4.1 *An* evaluation strategy *for a program P is a function $\sigma : Ter(\Sigma, \emptyset) \to \mathscr{C}_P$ such that the starting term of $\sigma(s)$ is s.*

The *input* to a program P is given as a ground term s. If P is terminating, then the *output* of P on input s with evaluation strategy σ is the final term of $\sigma(s)$. Notice that all irreducible terms in a COC system are constructor terms, so any output will always be a ground constructor term.

Define $P_\sigma(s,t)$ for a program P to mean that if s is the input to P with evaluation strategy σ, then t is the output. As a consequence of the confluence of any program P in this formalism, we have the following:

Proposition 4.2 *For any evaluation strategies σ, σ' for a program P, for any input s it holds that $P_\sigma(s,t)$ if and only if $P_{\sigma'}(s,t)$.*

4.2 I/O Assertions

Partial correctness of a program P with a specified evaluation strategy σ can be defined relative to an *input predicate I* and an *output predicate O*.

Definition 4.3 *A program P with evaluation strategy σ is* partially correct *with respect to I and O if for all inputs $s \equiv f(t_1, \ldots, t_n)$, where $f \in D$ and $t_1, \ldots, t_n \in Ter(C, \emptyset)$, it holds that*

$$I(t_1, \ldots, t_n) \wedge P_\sigma(f(t_1, \ldots, t_n), u) \to O(t_1, \ldots, t_n, u).$$

As a result of the previous proposition, if P is partially correct with a particular evaluation strategy σ, then P is *partially correct*: partially correct with any evaluation strategy. Thus one way to prove to the partial correctness of a program P is by means of its denotational semantics $\mathscr{D}(P)$.

Theorem 2 *If it holds that $I(t_1, \ldots, t_n) \wedge (\mathscr{D}(P) \models (f(t_1, \ldots, t_n) = u)) \wedge (u \text{ is irreducible}) \to O(t_1, \ldots, t_n, u)$, then P is partially correct with respect to I and O.*

Proof. Suppose $P_\sigma(f(t_1, \ldots, t_n), u)$; then $\mathscr{D}(P) \models f(t_1, \ldots, t_n) = u$. Also, u must be irreducible. Thus $I(t_1, \ldots, t_n)$ together with this implies $O(t_1, \ldots, t_n, u)$, so P_σ is partially correct with respect to I and O.

Total correctness is partial correctness together with termination. Many techniques for proving termination are known in the rewriting community.

To verify the binary search program, because it is terminating, one can do induction on the length of a computation sequence. All recursive calls will have shorter computation sequences. Therefore, assuming that the recursive calls are correct, one shows that the binary search program is correct.

In general, for any terminating recursive collection of procedures, one can specify separate input and output predicates for each procedure, and use a similar inductive approach to show that each procedure is correct assuming that all the recursive calls in it are correct.

5 Equivalence Relation on Terms

In an implementation of rewriting, it is convenient to store all occurrences of the same subterm only once and have pointers to this subterm. This idea also impacts translations of the programs into imperative languages. This identification of repeated subterms can be formalized by an *equivalence relation* on subterm occurrences in a program snapshot. The intention is that equivalent subterm occurrences would all be stored in the same location. Another way of achieving this is by representing terms as graphs, as in graph rewriting, but this is not considered here. The treatment of repeated subterms will also influence destructive operations, which are necessary for efficient implementations.

Equivalence classes are named using *identifiers*. A *decorated function symbol* is a pair $id : f$ where f is an ordinary *plain* function symbol and id is the name of an equivalence class. A *decorated term* is a term in which all the function symbols are decorated function symbols. If $id : f(t_1,\ldots,t_n)$ is a decorated term, then it is in the id equivalence class. Decorated terms must only identify subterms that are syntactically equivalent, so that if $id : f(s_1,\ldots,s_m)$ and $id : g(t_1,\ldots,t_n)$ are two decorated subterms in a program snapshot, then $f = g$, $m = n$, and $s_i \equiv t_i$ for all i. If s is a decorated term then $s \rhd t$ indicates that the *plain* term t is s with all the equivalence class identifiers removed.

5.1 Decorated Rewriting

The rewrite relation can be extended to a *decorated rewrite relation* $\Rightarrow^d$ on decorated terms. For this, a *decorated substitution* is of the form $\{t_1/x_1,\ldots,t_n/x_n\}$ where the t_i are decorated terms. Suppose $t[u]$ is a decorated ground term with u as a subterm. Suppose $u \rhd u'$ and u' unifies with the left hand side r of a rule $r \to s$ in R. Thus there is a substitution θ such that $u' \equiv r\theta$. Let r^d and θ^d be a decorated term and a decorated substitution such that $u \equiv r^d\theta^d$. Let s^d be a decorated s where the identifiers of $s'\theta^d$ (for s' a non-variable subterm of s^d) are chosen so that only syntactically identical subterms of $t[s^d\theta^d]$ are in the same equivalence class. Then $t[u] \Rightarrow^d_R t[s^d\theta^d]$. This implies that if the right-hand side of a rewrite rule has repeated variable occurrences, then all occurrences of a term replacing a given repeated variable will be equivalent.

The above description applies if there is only one term in the equivalence class of u. If there is more than one such term, then they all have to be rewritten together. Indicating all the occurrences of u in t by $t[u,u,\ldots,u]$, $t[u,u,\ldots,u] \Rightarrow^d_R t[s^d\theta^d, s^d\theta^d,\ldots,s^d\theta^d]$. This kind of rewriting, when all identical redexes are rewritten at the same time, is called *parallel rewriting*.

Equivalence class names also have to be assigned to subterms of the input term. That is, the input term has to be decorated. This can be done arbitrarily subject to the rule that only syntactically identical subterms can be in the same equivalence class. It may be convenient in some cases to have a *copy* operation such that $copy(s)$ produces a new copy of s with all subterms in new equivalence classes.

6 Assignment Statements

In a program expressed in this formalism, it is convenient to allow a single assignment style of programming on the right-hand sides of rewrite rules, similar to the imperative style of programming.

Definition 6.1 *An* assignment *is an expression of the form $x \leftarrow t$, for a variable x and term t.*

Assignments are used to define *aterms* inductively using the following syntax:

$\langle$ aterm $\rangle$:= $\langle$ term $\rangle$
$\langle$ aterm $\rangle$:= $\langle$ assignment $\rangle$; $\langle$ aterm $\rangle$

$\langle$ aterm $\rangle$:= $\langle$ function symbol $\rangle$ ($\langle$ aterm $\rangle, \ldots, \langle$ aterm $\rangle$)
An aterm $x \leftarrow t;\ E[x]$ is considered as a representation for the term $E[t]$ with the understanding that all occurrences of x are replaced by t.

The formalism is extended to allow a program to have rewrite rules whose right-hand sides are aterms. This is similar to the *let* construct found in some programming languages. The assignments do not influence the rewriting relation, but if the variable on the left-hand side of an assignment statement occurs more than once in the remainder of the program, then all these occurrences will be replaced by equivalent terms. An example of this is the aterm $x \leftarrow g(c); f(x,x)$, representing $f(g(c),g(c))$ with the two occurrences of $g(c)$ equivalent. So assignment statements can influence the equivalence relation and thus the decorated rewriting relation. Other than this, the program with assignment statements can be considered as equivalent to the version without them.

6.1 Variable bindings

There are restrictions on where variables may appear in terms having assignment statements. These restrictions are defined in terms of binding or scoping of variables.

In an expression of the form $x \leftarrow t;\ E[x]$, all occurrences of x in E are *bound* by the assignment statement $x \leftarrow t$. However, occurrences of x in t are not bound by this assignment statement. A variable occurrence that is bound by some assignment statement is said to be *bound*. A variable that is not bound is *free*.

The binding rule for variable occurrences is this: In a rewrite rule $r \rightarrow s$, if any variable x appears in s but not in r then all its occurrences in this rewrite rule must be bound, except that any occurrences of a variable x on the left hand side of an assignment statement $x \leftarrow t$ must be free. Also, if a variable appears in r in a rewrite rule $r \leftarrow s$ then all its occurrences in s must be free. These restrictions imply that in a sequence $x_1 \leftarrow t_1;\ \ldots;\ x_n \leftarrow t_n;\ E$, x_i cannot appear in $x_j \leftarrow t_j$ for $j < i$. All the possibilities for x_i being bound or free in $x_j \leftarrow t_j$ end up violating some restriction on binding. These restrictions also imply that in an assignment statement $x \leftarrow t$, x cannot appear in t. This prohibits assignment statements such as $i \leftarrow i+1$.

6.2 Order of eliminating assignment statements

Because an aterm $x \leftarrow t;\ E[x]$ is considered a representation for $E[t]$, assignments can be *eliminated* by replacing $x \leftarrow t;\ E[x]$ by $E[t]$. If there is more than one assignment statement, then the question arises whether the final result depends on the order in which the assignment statements are eliminated. It turns out that the result of eliminating assignment statements in a rewrite rule does not depend on the order in which they are eliminated, and even the equivalence class names are not affected. Thus a rewrite rule containing assignment statements unambiguously represents one without them, so that there is a unique *assignment-free* form of a program. For this, the terms can be considered as decorated terms to make it clear that the final result, including the names of the equivalence classes, does not depend on the order of elimination. Also, the process of elimination must terminate because the number of assignment statements in $E[t]$ is one less than the number in $x \leftarrow t;\ E[x]$ because t does not contain any assignment statements.

Theorem 3 *The result of eliminating assignment statements from a rewrite rule does not depend on the order in which the statements are eliminated.*

Proof. Consider two assignment statements in an expression of the following form:

$$E[x_1 \leftarrow t_1; E_1[x_2 \leftarrow t_2; E_2]]$$

Here x_1 cannot appear in t_1 but it can appear in E_1, t_2, and E_2, while x_2 cannot appear in t_1 or t_2 or E_1 outside of E_2, though it can appear in E_2, by the scoping rules. Doing the assignment $x_1 \leftarrow t_1$ first yields the expression $E[E_1[x_2 \leftarrow t_2; E_2](t_1/x_1)]$. For simplicity of notation let E_1' be $E_1(t_1/x_1)$, t_2' be $t_2(t_1/x_1)$ and E_2' be $E_2(t_1/x_1)$. Then the result of eliminating the first assignment statement can be written as

$$E[E_1'[x_2 \leftarrow t_2'; E_2']]$$

Eliminating the second assignment statement then yields $E[E_1'[E_2'(t_2'/x_2)]]$. Eliminating the inner assignment statement $x_2 \leftarrow t_2$ first yields $E[x_1 \leftarrow t_1; E_1[E_2(t_2/x_2)]]$. Then eliminating the outer statement $x_1 \leftarrow t_1$ yields $E[E_1[E_2(t_2/x_2)](t_1/x_1)]$ This is the same as $E[E_1'[E_2'(t_2'/x_2)]]$, so the result is the same regardless of the order of elimination of these two statements. One obtains a similar result more easily if the assignment statements are in disjoint subterms as in $E[x_1 \leftarrow t_1, x_2 \leftarrow t_2]$.

Thus assignment statements can be eliminated in either order, so this extends to any set of assignment statements and the result is unique independent of the order of elimination.

7 Destructive Operations

Destructive operations are needed for efficiency in imperative programs, but do not conform to pure term rewriting semantics. An alternate *destructive semantics*, which uses decorated rewriting so that all equivalent subterms are rewritten at once, can be used in order to support such operations in this system. It is important to have a way to relate the two semantics.

We illustrate the problem with a modification to an element of an array. Suppose that $A(s_1, \ldots, s_n)$ represents a one-dimensional array with n elements. Then $replace(t, i, A(s_1, \ldots, s_n))$ replaces the i-th argument of A with a term t; the result is $A(s_1, \ldots, t, \ldots, s_n)$, with t having replaced s_i. In the pure rewriting semantics, this operation creates a completely new term, and other occurrences of $A(s_1, \ldots, s_n)$ are not affected. However, in destructive semantics, the same storage is used for $A(s_1, \ldots, t, \ldots, s_n)$ as was used for $A(s_1, \ldots, s_n)$, and just the i-th element is modified, so all other occurrences of $A(s_1, \ldots, s_n)$ in the same equivalence class are also modified to $A(s_1, \ldots, t, \ldots, s_n)$. Thus the destructive operation may have a *side effect*. A similar situation can occur for a change to a pointer in the middle of a list; other references to this list will also have the pointer changed in destructive semantics, but not in rewriting semantics.

We use the notation $\Rightarrow_{\text{destr}}$ for the effect of an operation with destructive semantics, and $\Rightarrow_{\text{trs}}$ for the effect with rewriting semantics. As before, we indicate the equivalence class of term u by $id : u$. For the array modification example above, we have

$$t[id' : replace(t, i, id : A(s_1, \ldots, s_n)), id : A(s_1, \ldots, s_n)]$$

$$\Rightarrow_{\text{destr}} t[id : A(s_1, \ldots, t, \ldots, s_n), id : A(s_1, \ldots, t, \ldots, s_n)],$$

and

$$t[id' : replace(t, i, id : A(s_1, \ldots, s_n)), id : A(s_1, \ldots, s_n)]$$
$$\Rightarrow_{\text{trs}} t[id' : A(s_1, \ldots, t, \ldots, s_n), id : A(s_1, \ldots, s_n)].$$

In the expression $t[\ldots]$, the terms inside the brackets indicate distinct subterm occurrences in term t.

In general, for an operation f that is destructive on the i-th argument, we have

$$r[f(s_1, \ldots, s_n), s_i, u[s_i]] \Rightarrow_{\text{destr}} r[t, t, u[t]]$$

but $r[f(s_1,\ldots,s_n),s_i,u[s_i]] \Rightarrow_{\mathrm{trs}} r[t,s_i,u[s_i]]$. However, the situation can be more complicated than indicated. For example, there could be more than one equivalence class that is affected by a destructive operation, and the destructive operation may have other arguments besides the affected term. Additionally, some of the affected subterms could end up in the term that results from the destructive operation, and some affected subterms could be subterms of other affected subterms.

We will assume that for any destructive operation there is a sequence

$$\alpha = \langle r_1 \to s_1, r_2 \to s_2, \ldots, r_n \to s_n \rangle$$

of decorated rewrite rules such that if the destructive operation has a side effect that transforms r to s, then there is a rewrite sequence as follows:

$$t_1 \Rightarrow_1^d t_2 \Rightarrow_2^d t_3 \cdots \Rightarrow_n^d t_n$$

where $r \equiv t_1$, $s \equiv t_n$, and $\Rightarrow_i^d$ indicates rewriting with the single rewrite rule $r_i \to s_i$ using decorated rewriting. We also write $t_1 \Rightarrow_\alpha^d t_n$.

Definition 7.1 *The rewrites $r_i \to s_i$ are called* primary modifications*, changes to any other terms as a result are called* secondary modifications*.*

In the array assignment example, there is one primary modification, the rule

$$A(s_1,\ldots,s_n) \to A(s_1,\ldots,t,\ldots,s_n).$$

Equivalent terms s and t will have affected subterms at corresponding locations, and so will both be affected in the same way by the destructive operation. This leads to the following proposition.

Proposition 7.2 *For terms s and t in the same equivalence class, if $s \Rightarrow_\alpha^d u$, then $t \Rightarrow_\alpha^d u$.*

As a result of this, the entire effect of the destructive operation can be summarized by giving its effect on each equivalence class of (constructor) terms, and even by the maximal (size) affected terms.

Destructive semantics is not confluent, so the evaluation strategy has to be defined to give a program a precise meaning with the destructive semantics. Consider the following program, with starting term $f(\langle 1,2\rangle)$.

$$f(x) \to \langle arg(1,x), replace(1,2,x)\rangle$$

If the first element of the tuple is evaluated first we get $\langle 1,\langle 2,2\rangle\rangle$, which agrees with the rewriting semantics. Otherwise, we get $\langle 2,\langle 2,2\rangle\rangle$, which does not, noting that the two occurrences of x on the right hand side of the rule are in the same equivalence class, and assuming replace is destructive.

In general, in any primary modification $r_i \to s_i$, both r_i and s_i will be constructor terms, which is not allowed for ordinary rewriting. These rules $r_i \to s_i$ introduce a critical pair between rules, which leads to non-confluence.

One way to handle this situation is to modify the program P to obtain a program P', such that the rewriting semantics of P' is the same as the destructive semantics of P. An alternative is to develop proof methods for destructive semantics, which still has a fairly simple syntax and semantics. Destructive semantics permits a simpler translation into imperative languages, while rewriting semantics appears to be better for proofs of correctness.

8 Translation Issues

The programs translated from a rewrite program should be efficient and correct. As for program constructs in the translated program, one obtains recursion automatically from term rewriting. Iteration results from last call optimization. Assignment statements can be obtained from the single assignment style of the rewrite programs. When translating, one can assume that compiled procedures are already implemented in the target language.

As for correctness, the idea is that one only needs to prove it once for an abstract rewriting program; then correctness follows for the translated, concrete programs in various languages assuming the translations are correct. This can be shown as follows:

Suppose one has the partial correctness of an abstract program P in the following form: $I(t_1,\ldots,t_n) \wedge f(t_1,\ldots,t_n) \Rightarrow u \wedge (u$ is irreducible$)$ implies $O(t_1,\ldots,t_n,u)$, As before, I is an input assertion and O is an output assertion. Suppose we also have the following relation between the translated program $Tr(P)$ and the original (abstract) program P: If $Tr(f(t_1,\ldots,t_n))$ has output u' then there is an irreducible u such that $f(t_1,\ldots,t_n) \Rightarrow u$ and $Tr(u) = u'$. Let the input and output assertions I' and O' on the translated program $Tr(P)$ be: $I'(t')$ is $(\exists t_1,\ldots,t_n) I(t_1,\ldots,t_n) \wedge Tr(f(t_1,\ldots,t_n)) = f'(t'_1,\ldots,t'_n)$, and $O'(t'_1,\ldots,t'_n,u')$ is

$$\exists t_1,\ldots,t_n,u\ [Tr(f(t_1,\ldots,t_n)) = f'(t'_1,\ldots,t'_n) \wedge f(t_1,\ldots,t_n) \Rightarrow u \wedge O(t_1,\ldots,t_n,u) \wedge Tr(u) = u'].$$

Then the following theorem holds.

Theorem 4 *$I'(t'_1,\ldots,t'_n)$ and ($f'(t'_1,\ldots,t'_n)$ has output u') imply $O'(t'_1,\ldots,t'_n,u')$.*

Proof. Suppose $I'(t'_1,\ldots,t'_n)$, then $\exists t_1 \ldots t_n\ Tr(f(t_1,\ldots,t_n)) = f'(t'_1,\ldots,t'_n) \wedge I(t_1,\ldots,t_n)$. Then because $f'(t'_1,\ \ldots,t'_n)$ has output u', we know by properties of the translation that there is an irreducible term u such that $f(t_1,\ldots,t_n) \Rightarrow u$ and $Tr(u) = u'$. By partial correctness of the original program, $O(t_1,\ldots,t_n,u)$. Thus there exist $t_1,\ldots,t_n,u$ such that

$$Tr(f(t_1,\ldots,t_n)) = f'(t'_1,\ldots,t'_n) \wedge f(t_1,\ldots,t_n) \Rightarrow u \wedge O(t_1,\ldots,t_n,u) \wedge Tr(u) = u',$$

hence $O'(t'_1,\ldots,t'_n,\ u')$.

For proofs of partial correctness it is necessary to use different axioms that distinguish terms that are equal in the declarative semantics. For example, when adding two numbers m and n, one wants to return a single number p equal to their sum, but in the declarative semantics, $p = m+n$ so the input and output terms cannot be distinguished. This applies both to the original and translated programs. We may want to specify for example that the output is a sequence of bits representing an integer.

9 Conclusion

Term rewriting has long been used to model computation, and as a way of implementing functional languages. In this paper, a system based on term rewriting is developed to express abstract programs which can be translated into programs in mainstream languages. Term rewriting is similarly used in the ASF+SDF Meta-Environment [5], which can be used to generate programs from algebraic specifications. In the system presented here, programs are designed to be similar to imperative programs in style, to permit efficient translation into programs in imperative languages, and remain fairly simple for the typical programmer to write.

To model efficient imperative languages, an equivalence relation on term occurrences can be used, and a version of rewriting called decorated rewriting is developed for this purpose. We are interested in exploring the use of term graph rewriting for this purpose.

The system also permits a single assignment style of programming, and can be adapted to destructive operations with side effects. The pure rewriting language is confluent, but the version with destructive operations is not. We are interested in developing techniques for generating programs in which the confluent rewriting semantics expresses the destructive semantics of other programs, in order to address this issue.

Methods for proofs of correctness of the abstract programs as well as the translated programs have been sketched. The approach given in this paper could be extended to more sophisticated languages such as Coq [2] and other systems [6][4]. Issues of nondeterminism and nontermination have not been addressed. Also, a detailed consideration of the translation mechanism is beyond the scope of this paper.

References

[1] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, Cambridge, England.

[2] Thierry Coquand & Gérard P. Huet (1988): *The Calculus of Constructions*. Information and Computation 76(2-3), pp. 95–120.

[3] Vijay D'Silva, Daniel Kroening & Georg Weissenbacher (2008): *A Survey of Automated Techniques for Formal Software Verification*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27, pp. 1165–1178.

[4] M.J. Gordon & T.F. Melham, editors (1993): *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press.

[5] Paul Klint (1993): *A Meta-Environment for Generating Programming Environments*. ACM Transactions on Software Engineering and Methodology 2, pp. 176–201.

[6] L.C. Paulson (1994): *Isabelle: A Generic Theorem Prover*. Springer Verlag, New York. LNCS Volume 828.