A tree transformation facility using typed rewrite systems

Charles Farnum*

July 1, 1988

Abstract

Tree rewrite systems with typed variables can be used to represent many tree transformations in a more compact form than systems with untyped variables. By building on the work of Eduardo Pelegrí-Llopart, it is possible to generate linear-time optimal solutions to SET-REACHABILITY — a generalization of REACHABILITY with a possibly infinite goal set — for a useful class of typed rewrite systems. The algorithms developed can also handle some untyped systems that are not in BURS, such as systems with rules of the form $X \rightarrow a(X)$.

An experiment involving a rewrite system for instruction selection for the Motorola 68000 produced table sizes an order of magnitude larger than those produced by an untyped rewrite system for the same task. It is not clear whether this table size can be limited, or if it is an inherent cost of the power given by types. Although discouraging for the instruction selection application, the table sizes are small enough (under 100k bytes) that the techniques may be useful for smaller applications, or in cases where the added expressibility of typed variables outweighs the size explosion of the tables.

*Sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 (monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089), and an NSF Graduate Fellowship.

1 Introduction

A *tree transformation system* is a relation between trees in an input language and an output language. Many problems in compiler writing are naturally modeledusing tree transformation systems: e.g., the input and outputlanguages may be two different intermediate representations, with thetree transformation system defining the possible implementations of theinput language in terms of the output language. Usually the tree transformation system is described in some compact form, and the problem is to find some (possibly optimal) output tree that is related to a given input tree.

Eduardo Pelegrí-Llopart's PhD dissertation [2] focuses on tree *rewrite systems.* A rewrite system is a collection of functions, called rules, from trees to trees. An individual rule contains an *input pattern* used to specify a set of input trees — those trees which the pattern *matches* — to which the rule applies, and an *output pattern* which constructs an output tree, possibly using pieces of the input tree. A tree t_1 can be *rewritten* into a tree t_n if there is a sequence of trees $t_1, \ldots, t_n$ where t_{i+1} is obtained from t_i by replacing some subtree with its image under a rule in the rewrite system. A tree transformation system can be obtained from a rewrite system by defining two trees to be related if one can be rewritten into the other.

Given a rewrite system R, a goal tree g, and an input tree t, the REACHABILITY problem is to find a sequence of rewrites from t to g, or show that there is no such sequence. Pelegrí showed how to solve the REACHABILITY problem efficiently in the case that R and g are fixed and only t varies, for rewrite systems in the class BURS. His solution yields optimal rewrite sequences, given a linear cost function on the rewrite sequences.

In a REACHABILITY problem, the main item of interest is the rewrite sequence from the tree to the goal; some meaning outside of the rewrite system is attached to individual rules, and is used to solve the particular problem (e.g., code generation) at hand. Often the original problem calls for a tree to be generated. If this is the case, the SET-REACHABILITY problem may be more useful: given a rewrite system R, a set of goal trees G, and an input tree t, find a sequence of rewrites from t to any tree in G, or show that there is no such sequence.

This report describes CRSTNA[1], a system that produces an efficient solver of SET-REACHABILITY given an R and G satisfying certain restrictions. CRSTNA extends Pelegrí's work by

- solving SET-REACHABILITY, a generalization of REACHABILITY,
- extending the power of rewrite rules by allowing a more powerful type of input pattern, and
- accepting a wider variety of rewrite systems.

Familiarity with Pelegrí's dissertation is not required to use CRSTNA or to verify the theory in this report, but it covers the uses and motivation behind rewrite systems in much greater detail. This report assumes that readers are familiar with his work, or with other recent work in rewrite systems. In particular, this report stops short of providing a complete description of the theory and implementation, since the final parts of the solution are identical to Pelegrí's.

2 A User's View

In this section, we informally describe the kinds of rewrite systems CRSTNA is intended to process, and consider the limitations imposed by the algorithms used in CRSTNA.

2.1 Specifying trees, patterns, and rewrite rules

Trees in CRSTNA are rooted, ordered acyclic graphs labeled with *operators*. Each operator has an associated integer called its *arity*; a node labeled with the operator a must have as many children as the arity of a. We use the notation $a[c_1, \ldots, c_n]$ to denote the tree with the root labeled by a with children c_1 through c_n. For example, the tree $+[reg_1[], reg_2[]]$ might represent a machine add instruction.

A *pattern* is a tree in which some of the operators are *variables*. CRSTNA only allows patterns in which all of the variables have zero arity, and each variable occurs at most once.[2] A variable has associated with it a set of trees called its *type*. A pattern ϱ *matches* a tree t if

[1]Compositional Rewrite System Tool with Natural Automata.

[2]linear n-patterns, in Pelegrí's terminology.

- ϱ and t have the same label at the root, and each child of ϱ's root matches the corresponding child of t, or

- ϱ is labeled by a variable at the root, and t is a member of the type of the variable.

Variables are denoted with upper-case letters; their type may be specified by following the variable name with a colon and the name of the type. Variables with no specified type have the universal type (the set of all trees). Thus, the pattern X matches all trees; $+[X, reg_2[]]$ matches $+[reg_1[], reg_2[]]$ and $+[+[reg_1[], reg_2[]], reg_2]$, among others. If t is the type $\{reg_1, reg_2\}$, then $\mathit{add1}\,[X : t[]]$ matches $\mathit{add1}\,[reg_1[]]$ but not $\mathit{add1}\,[reg_3[]]$.

A *rewrite rule* is specified with an input pattern and an output pattern, and is written $\alpha \rightarrow \beta$, where α is the input pattern and β the output pattern. Each variable in the output pattern must occur exactly once in the input pattern. $\alpha \rightarrow \beta$ specifies the function r defined as follows:

- If α does not match t, $r(t)$ is undefined.

- If α matches t, $r(t)$ is β with variables in β replaced by sub-trees in t that are matched by the corresponding variables in α.

For example,

$$+[X[], Y[]] \rightarrow +[Y[], X[]]$$

maps the tree $+[1[], reg_1[]]$ into $+[reg_1[], 1[]]$, and

$$+[X[], 1[]] \rightarrow \mathit{add1}\,[X[]]$$

maps $+[reg_1[], 1[]]$ into $\mathit{add1}\,[reg_1[]]$.

The types used in a rewrite system are specified with a set of *type instances*. A type instance is a statement of the form

$$\varrho \text{ is-a } t,$$

where ϱ is a pattern and t is a type name; it states that any tree matching ϱ is an element of the type named by t. This is a recursive definition; ϱ is allowed to have variables with t as their type, or with types specified in terms of t. For example, given the set of type instances

$$\begin{array}{rcl} \mathit{true}[] & \text{is-a} & t \\ \mathit{not}[X : t[]] & \text{is-a} & f \\ \mathit{not}[X : f[]] & \text{is-a} & t, \end{array}$$

f is the set of trees with an odd number of nodes labeled *not* and with one leaf, labeled *true*.

CRSTNA requires a set of type instances, a set of rewrite rules, and a goal type; it then produces an efficient algorithm that, given an input tree, finds a sequence of rewrites from the input tree to a goal tree. If costs are associated with rewrite rules, CRSTNA will find a rewrite sequence with minimum cost (where the cost of a sequence is the sum of the costs of the individual rules). As a final example, the following specification might be used to remove additions of zero, and replace additions of one by increment operators, in simple expression trees:

$$\begin{array}{rcl} g & & \text{is the goal type} \\ \mathit{ident}[] & \text{is-a} & g \\ +[X:g[],Y:g[]] & \text{is-a} & g \\ \mathit{add1}\,[X:g[]] & \text{is a} & g \end{array}$$

$$\begin{array}{rcl} +[X[],Y[]] & \rightarrow & +[Y[],X[]] \\ +[X[],1[]] & \rightarrow & \mathit{add1}\,[X[]] \\ +[X[],0[]] & \rightarrow & X[] \end{array}$$

2.2 Restrictions on types

Sets of type instances are equivalent in power to labeled bottom-up finite state automata. The types that can be specified are precisely the *recognizable* tree languages.[3] In order for CRSTNA to solve SET-REACHABILITY, the types must be *closed* under the rewrite system, i.e., if T is a type used in the system, then for all t in T, if t can be rewritten into some tree t', t' must be in T.

This might seem to be a severe restriction on the power of CRSTNA, but it is perfectly acceptable for applications in which the types of a tree (i.e., those types of which it is a member) say something about the semantics of a tree. In this case, enforcing the closure of the type system corresponds to insisting that a rewrite rule can only enrich, not destroy, the semantic information corresponding to a tree.

[3]Bottom-up finite state automata, or BFSAs, are a natural generalization of string automata to trees; the class of tree languages that can be recognized by BFSAs are called *recognizable*, as are the string languages that can be recognized by string automata.

2.3 Restrictions on rewrite rules

CRSTNA generates linear time solvers; they compute all of the information needed to solve SET-REACHABILITY in a single bottom-up pass over the input tree. This strongly limits the rewrite systems which CRSTNA can handle. There is no simple characterization of rules which are or are not acceptable; the interaction between different rules is of major importance. In practice, we have found two general problems that can occur:

1. There exists a sequence of rules which can pass information arbitrarily far down the tree. For example, the rule $a[b[X[]]] \rightarrow a[a[X[]]]$, applied successively to a tree of the form $a[b[b[\ldots]]]$ passes the information that there is an a at the root arbitrarily far downward.

   This will always be a problem in any system which attempts to track all interesting rewrites in a single bottom-up pass over the tree, since it is impossible to know if a particular rewrite is possible without information arbitrarily high in the tree.

2. There exists a sequence of rewrites which can be reapplied at the root (but not at subtrees) an unbounded number of times, depending on the size of the tree. This requires keeping track of an unbounded amount of information while traversing the tree, and thus will also always be a problem for any solution which does its computation in a single bottom-up traversal.

These two problems can be formalized, and I believe they form an exhaustive list; a future goal is to prove CRSTNA can handle any system that has neither of these problems.

3 Implementation

We here describe the theory behind CRSTNA, and a few of the details of the implementation.

3.1 Typed rewrite systems

We make use of the following notations. The domain of a function f is written $\mathcal{D}(f)$. A sequence with elements e_1 through e_n is written $e_1 \cdots e_n$;

a singleton sequence is often denoted by its single element when it is clear by context that a sequence is required. The concatenation of two sequences s_1 and s_2 is written either as $s_1 s_2$ or as $s_1 // s_2$. The length of a sequence s is denoted *length*(s). $s_1 \cdots \widehat{s_i} \cdots s_n$ denotes the sequence $s_1 \cdots s_{i-1} // s_{i+1} \cdots s_n$. The *head* of a sequence is the first element; the *tail* is the sequence with the first element removed. Tuples (sequences with fixed numbers of elements) are enclosed in angle brackets, with elements separated by commas. Functions are often described with "defining equations": $\triangleq$ reads "is defined to be", and $x \triangleq y$ means that if y is defined, then x is defined to be y.

The trees used in CRSTNA are rooted, ordered, and labeled, where the label at a node determines the arity of that node. Formally, we will define a tree to be a mapping from positions in the tree to operators; positions will be described by sequences of integers, with the empty sequence corresponding to the root of the tree and the sequence $p//i$ corresponding to the i'th child of the node at position p. $t@p$ is the subtree of t at position p, and $t \overset{@p}{\leftarrow} t'$ is the tree formed by replacing the subtree of t at p with t'.

Definition 1 *An* **operator set** $\mathcal{O}$ *is a pair* $\langle O, N \rangle$ *where* O *is any set and* N *is a mapping from* O *to the non-negative integers. The members of* O *are called* **operators**. *The* **arity** *of an operator* o *is* $N(o)$. *An operator with arity* n *is called an* n*-ary operator; an operator with arity zero is called a nullary operator.*

A **position** *is a sequence of positive integers.* $\mathcal{P}$ *denotes the set of all positions.*

A **tree shape** *is a set* P *of positions where, for all positions* p *and* q, $p//q \in P$ *implies* $p \in P$, *and for all positions* p *and integers* $i > 1$, $p//i \in P$ *implies* $p//(i-1) \in P$.

A **tree** t **over an operator set** $\mathcal{O} = \langle O, N \rangle$ *is a mapping from a tree shape* P *to* O *where, for all* $p \in P$ *and integers* i, $p//i \in P$ *iff* $0 < i \leq N(t(p))$. $\mathcal{T}_{\mathcal{O}}$ *denotes the set of trees over* $\mathcal{O}$.

For any function f *with* $\mathcal{D}(f) \subset \mathcal{P}$, $f@p$ *is the function*

$$(f@p)(q) \triangleq f(p//q).$$

$f \overset{@p}{\leftarrow} g$ *is the function*

$$(f \overset{@p}{\leftarrow} g)(q) \triangleq \begin{cases} g(s) \text{ if } \exists\, s \ni : q = p//s, \\ f(q) \text{ otherwise.} \end{cases}$$

We sometimes use the notation $a[t_1, \ldots, t_n]$ to denote the tree t with $t(\varepsilon) = a$ and $t@i = t_i$.

In Pelegrí's thesis, and in earlier sections, patterns may have *variables* as operators. A variable can be formally defined as a pair of a name and a type. In linear patterns (where any given variable appears at most once), only the type of the variable is important in determining which trees are matched by the pattern; thus, it is possible to have several different patterns, all of which are equivalent in terms of matching trees. This causes many theorems to be more difficult to state and prove than is inherently necessary.

We therefore use *wildcards* instead of variables in our formal work. A wildcard is like an anonymous variable; it consists of simply a type:

Definition 2 *A* ***pattern over an operator set*** *$\mathcal{O} = \langle O, N\rangle$ is a tree over the operator set $\langle O \cup 2^{\mathcal{T}_\mathcal{O}}, N'\rangle$, where*

$$N'(x) \triangleq \begin{cases} N(x) & \text{if } x \in O, \\ 0 & \text{if } x \in 2^{\mathcal{T}_\mathcal{O}}. \end{cases}$$

A member of $2^{\mathcal{T}_\mathcal{O}}$ is called a ***wildcard****.*

$\mathcal{W}(\varrho) = \{q \mid \varrho(q) \in 2^{\mathcal{T}_\mathcal{O}}\}$ is the set of ***wildcard positions*** *of the pattern ϱ.*

A pattern ϱ ***matches*** *a tree t if, for all $p \in \mathcal{D}(\varrho)$, $\varrho(p) \in O$ implies $t(p) = \varrho(p)$ and $\varrho(p) \in 2^{\mathcal{T}_\mathcal{O}}$ implies $t@p \in \varrho(p)$.*

For a pattern ϱ, $L(\varrho)$ is the set $\{\, t \mid \varrho \text{ matches } t\,\}$.

Two patterns ϱ_1 and ϱ_2 are ***equivalent****, denoted $\varrho_1 \equiv \varrho_2$, if $L(\varrho_1) = L(\varrho_2)$.*

Our definition allows the wildcards to be arbitrary sets of trees. This general definition creates problems for practical implementations, not the least of which is specification of the wildcards. We will constrain wildcards by associating them with individual states of a *bottom-up finite state automata* (BFSA):

Definition 3 *Given an operator set $\langle O, N\rangle$, a* ***deterministic bottom-up finite state automata****, or BFSA, is a pair $\langle S, \delta\rangle$ where S is a finite set of* ***states*** *and δ, the* ***transition function****, is a function $\delta\colon O \times S^* \to S$ where $\delta(a, s)$ is undefined for all s with $\mathit{length}(s) \neq N(a)$.*

Given an operator set $\langle O, N\rangle$, a tree t and a BFSA $\mathcal{A} = \langle S, \delta\rangle$, the ***state assigned to*** *t by $\mathcal{A}$, denoted by $\mathcal{A}(t)$, is $\delta(t(\varepsilon), \mathcal{A}(t@1) \cdots \mathcal{A}(t@N(t(\varepsilon))))$.*

For a state s, $L(s)$ is the set $\{ t \mid \mathcal{A}(t) = s \}$.

A state s is **useful** *if $L(s)$ is non-empty. We henceforward assume that every state of $\mathcal{A}$ is useful.*

Given a BFSA $\mathcal{A}$, a wildcard W is **single-state recognizable** *(SSR) if there exists a state s in $S_{\mathcal{A}}$ with $W = L(s)$. RECOG is the set of wildcards that are SSR by some BFSA.*

Given a BFSA $\mathcal{A}$, a pattern ϱ is **wildcard-SSR** *if, for each wildcard position w of ϱ, $\varrho(w)$ is SSR.*

If ϱ is wildcard-SSR by $\mathcal{A}$, the **state assigned to** *ϱ by $\mathcal{A}$ is $\mathcal{A}(\varrho)$ where*

$$\mathcal{A}(\varrho) \triangleq \begin{cases} \delta(\varrho(\varepsilon), \mathcal{A}(\varrho@1) \cdots \mathcal{A}(\varrho@N(\varrho(\varepsilon)))) & \text{if } \varrho(\varepsilon) \in O, \\ s & \text{if } \varrho(\varepsilon) = L(s). \end{cases}$$

Patterns that are wildcard-SSR by a BFSA have two main advantages. First of all, whether or not a tree is matched by such a pattern can be computed in a single bottom-up traversal of the tree; Pelegrí showed that this is true for any pattern where the wildcards are members of RECOG. Secondly, such patterns can easily be placed in a normal form. The use of wildcards instead of variables is motivated by the desire that equivalent patterns be equal. This is not the case for arbitrary wildcard-SSR patterns; for example, if $\varrho_1 = a[L(s_1)[]]$ and $\varrho_2 = a[b[L(s_2)[]]]$ where $L(s_1) = L(b[L(s_2)[]])$, then ϱ_1 and ϱ_2 are equivalent but not equal. $L(s_1) = L(b[L(s_2)[]])$ implies $\langle b, s_2 \rangle$ is the only pair with $\delta(b, s_2) = s_1$, i.e., there is only one transition in the automaton leading to the state s_1. This suggests the following definition of a normal form for patterns:

Definition 4 *Given a BFSA $\mathcal{A}$ with transition function δ, a* **transition** *is a pattern $a[L(s_1)[], \ldots, L(s_n)[]]$ where $\delta(a, s_1 \cdots s_n)$ is defined. The transition α* **leads to** *the state $\mathcal{A}(\alpha)$.*

A pattern ϱ is in **$\mathcal{A}$-normal form** *if it is wildcard-SSR by $\mathcal{A}$ and, for all wildcard positions p of ϱ, there are two different transitions α_1 and α_2 leading to $\mathcal{A}(\varrho@p)$.*

Proposition 1 *If ϱ_1 and ϱ_2 are in $\mathcal{A}$-normal form, then $\varrho_1 = \varrho_2$ iff $\varrho_1 \equiv \varrho_2$.*

Proof If the patterns are equal, they are equivalent.

If they are equivalent, we will prove they are equal by induction on the height of the patterns. Suppose that the children of ϱ_1 and ϱ_2 are equal (this is vacuously true for the base case). If $\varrho_1(\varepsilon) \in O$ and $\varrho_2(\varepsilon) \in O$, then clearly the operators must be the same for the patterns to be equivalent, so the patterns are equal. If $\varrho_1(\varepsilon)$ and $\varrho_2(\varepsilon)$ are both wildcards, they must correspond to the same state in $\mathcal{A}$ in order to match the same trees, and so the patterns are equal. Otherwise, suppose WLOG that $\varrho_1(\varepsilon)$ is a wildcard and $\varrho_2(\varepsilon) \in O$.

Since ϱ_1 is in $\mathcal{A}$-normal form, there must be two different transitions ϱ and ϱ' leading to $\mathcal{A}(\varrho_1)$. It can be shown that $L(\varrho) \cup L(\varrho') \subseteq L(\varrho_1)$ and that $L(\varrho) \cup L(\varrho') \not\subseteq L(\varrho_2)$; but this implies $\varrho_1 \not\equiv \varrho_2$, a contradiction. □

Proposition 2 *If ϱ is wildcard-SSR by $\mathcal{A}$, there exists a $\varrho' \equiv \varrho$ in $\mathcal{A}$-normal form.*

Proof Suppose $\varrho(\varepsilon)$ is a wildcard. We will proceed by induction on the height of a minimal height tree in $L(\varrho)$. If there are at least two transitions leading to ϱ, then ϱ is in $\mathcal{A}$-normal form by definition. Otherwise, $\varrho \equiv a[L(s_1)[], \ldots, L(s_n)[]]$ for some operator a and wildcards $L(s_i)$, each of which have equivalent $\mathcal{A}$-normal forms by the induction hypothesis; therefore, ϱ has an equivalent $\mathcal{A}$-normal form.

If $\varrho(\varepsilon)$ is not a wildcard, then replacing each of its wildcards with an equivalent pattern in $\mathcal{A}$-normal form yields an equivalent pattern in $\mathcal{A}$-normal form. □

Given patterns, we can now define *rewrite rules*. A rewrite rule is a partial function mapping trees to trees. The domain of the rule is specified by an *input pattern*, and the range by an *output pattern*; the function itself is specified by relating wildcard positions in the output pattern to wildcard positions in the input pattern:

Definition 5 *A* **rewrite rule** *r is of the form $\alpha \xrightarrow{w} \beta$ where α and β are patterns called the* **input pattern** *and* **output pattern**, *respectively, and w is a 1–1 function $w: \mathcal{W}(\beta) \rightarrow \mathcal{W}(\alpha)$ where $\alpha(w(p)) = \beta(p)$ for all $p \in \mathcal{W}(\beta)$.*

A **rewrite system** *is a collection of rewrite rules.*

A rewrite rule is in **$\mathcal{A}$-normal form** *if both the input pattern and the output pattern are in $\mathcal{A}$-normal form. A rewrite system is in* **$\mathcal{A}$-normal form** *if every rewrite rule is in $\mathcal{A}$-normal form.*

The rewrite rule $r = \alpha \xrightarrow{w} \beta$ is **applicable to** t *if α matches t. If r is applicable to t, $r(t)$, the* **application of** r **to** t, *is the tree*

$$\beta \overset{@p_1}{\leftarrow} (t@w(p_1)) \cdots \overset{@p_n}{\leftarrow} (t@w(p_n))$$

where $\mathcal{W}(\beta) = \{p_1, \ldots, p_n\}$.

Two rules r_1 and r_2 are **equivalent**, *written $r_1 \equiv r_2$, if, for all trees t, $r_1(t) = t'$ iff $r_2(t) = t'$.*

Proposition 3 *If r_1 and r_2 are both rules in $\mathcal{A}$-normal form, then $r_1 \equiv r_2$ iff $r_1 = r_2$.*

Proof If the rules are equal, they are equivalent.

If the rules are equivalent, then they have equal domains; this implies that their input patterns are equivalent, and since they are in $\mathcal{A}$-normal form, they must be equal. Likewise, they have equal ranges, and therefore their output patterns are equal. Suppose $w_1 \neq w_2$; then there is some $p \in \mathcal{W}(\beta)$ with $w_1(p) \neq w_2(p)$. Since β_1 is in $\mathcal{A}$-normal form, there are two different transitions ϱ and ϱ' leading to $\mathcal{A}(\beta_1@p)$; this implies that there are two different trees t_1 and t_2 matching $\beta_1@p$. Let t be a tree matching α_1 with $t@w_1(p) = t_1$ and $t@w_2(p) = t_2$. It is easy to verify that such a tree exists and that $r_1(t) \neq r_2(t)$, a contradiction. □

Due in part to this nice property, we will from now on assume that all rewrite systems are in $\mathcal{A}$-normal form. In section 3.2, we will show that these systems are equivalent in power to systems in which wildcards are allowed to be any member of RECOG.

We are now ready to define SET-REACHABILITY:

Definition 6 *A* **rewrite application** *is a pair $\langle r, p\rangle$ of a rule r and a position p. $\langle r, p\rangle$ is* **applicable to** t *if r is applicable to $t@p$. If $\langle r, p\rangle$ is applicable to t, then $\langle r, p\rangle(t)$, the* **application of** $\langle r, p\rangle$ **to** t, *is the tree $t \overset{@p}{\leftarrow} r(t@p)$.*

A **rewrite sequence** *is a sequence of rewrite applications; a rewrite sequence $\langle r_1, p_1\rangle \cdots \langle r_n, p_n\rangle$ is* **applicable to** t *if $\langle r_1, p_1\rangle$ is applicable to t and $\langle r_2, p_2\rangle \cdots \langle r_n, p_n\rangle$ is applicable to $\langle r_1, p_1\rangle(t)$. A rewrite sequence is* **valid** *if it is applicable to some tree t.*

If every rewrite application in a sequence τ is of the form $\langle r_i, p//q_i\rangle$ for some position p, then $\tau@p$ denotes the rewrite sequence $\langle r_1, q_1\rangle \cdots \langle r_n, q_n\rangle$.

Let τ and ϕ be valid rewrite sequences. If $\tau(t) = t'$ implies $\phi(t) = t'$, then ϕ **covers** *τ, denoted $\phi \supset \tau$.*

Let τ be a valid rewrite sequence. If there exists a rewrite rule r such that, for all trees t, $\langle r, \varepsilon\rangle(t) = t'$ iff $\tau(t) = t'$, then r is the **composition** *of τ.*

Let R be a rewrite system over $\mathcal{O}$, and G be a subset of $\mathcal{T}_{\mathcal{O}}$. The SET-REACHABILITY **problem** *is, given R, G, and a tree t over $\mathcal{O}$, to find a rewrite sequence τ such that $\tau(t) \in G$, or to show that there is no such sequence.*

In order to solve SET-REACHABILITY efficiently, we constrain the number of rewrite sequences that must be considered. If we can show that some sequence τ is covered by a different sequence ϕ, then there is no need to consider τ; ϕ will provide an adequate solution for SET-REACHABILITY whenever τ would. Our first "pruning" of the set of all rewrite sequences will be to consider only those sequences in *compositional normal form* (CNF):

Definition 7 *Let τ be a valid rewrite sequence. τ is in* **compositional normal form at** *ε if it is in the form $\tau_1 \cdots \tau_n \tau_*$ such that*

(1) for $1 \leq i \leq n$, all rewrite applications in τ_i have positions whose head is i, and

(2) $\tau_* = \langle r_1, p_1\rangle\langle r_2, p_2\rangle \cdots \langle r_m, p_m\rangle$ where $\langle r_1, p_1\rangle = \langle \alpha_1 \xrightarrow{w_1} \beta_1, \varepsilon\rangle$ and, for $1 < i \leq m$, $\langle r_1, p_1\rangle \cdots \langle r_i, p_i\rangle$ has a composition $\alpha_i \xrightarrow{w_i} \beta_i$ with $\beta_{i-1}(p_i)$ in O.

τ is in **compositional normal form** *(CNF) if it is in CNF at ε, $\langle r_2, p_2\rangle \cdots \langle r_m, p_m\rangle$ is in CNF, and, for all i, $\tau_i@i$ is in CNF.*

Our solution to SET-REACHABILITY will involve a single bottom-up traversal of the tree computing all of the "interesting" rewrite sequences (those that we cannot show are covered by others) to be applied at each position. Given this approach, the bottom-up nature of the CNF (i.e., all rewrites at subtrees are done before rewrites at the root) is necessary. It also places a strong constraint on the rewrite systems which can be handled: it cannot be possible for the absence of one rewrite to enable a second rewrite arbitrarily far down in the tree. We enforce this constraint by insisting that the rewrite sequence be *type-closed*:

Definition 8 *Two patterns ϱ_1 and ϱ_2 are* **similar**, *written $\varrho_1 \sim \varrho_2$, if $\mathcal{D}(\varrho_1) = \mathcal{D}(\varrho_2)$, $\mathcal{W}(\varrho_1) = \mathcal{W}(\varrho_2)$, and $\varrho_1(p) \neq \varrho_2(p)$ implies $p \in \mathcal{W}(\varrho_1)$.*

Two rewrite rules $r_1 = \alpha_1 \xrightarrow{w_1} \beta_1$ and $r_2 = \alpha_2 \xrightarrow{w_2} \beta_2$ are **similar**, *written $r_1 \sim r_2$, if $w_1 = w_2$, $\alpha_1 \sim \alpha_2$, and $\beta_1 \sim \beta_2$.*

A rewrite system R is **type-closed** *if, for all rules $r = \alpha \xrightarrow{w} \beta$ and trees t with r applicable to t, for all positions $p \in \mathcal{W}(\alpha)$, for all rewrite sequences τ in R applicable to $t@p$, there exists a rule $r' \sim r$ such that r' is applicable to $t \overset{@p}{\leftarrow} \tau(t@p)$.*

Section 3.2 gives the reason behind the name "type-closed". Given a type-closed system, we need only consider the CNF rewrite sequences in order to solve SET-REACHABILITY:

Proposition 4 *Let R be a type-closed rewrite system, and let ϕ be a valid rewrite sequence in R. There exists a rewrite sequence τ in CNF that covers ϕ.*

In order to prove this proposition, we need some lemmas.

Lemma 4.1 *Let $\tau = \langle \alpha_1 \xrightarrow{w_1} \beta_1, \varepsilon\rangle\langle \alpha_2 \xrightarrow{w_2} \beta_2, p\rangle$ be a valid rewrite sequence. If $\beta_1(p) \in O$, τ has a composition.*

Proof It is straightforward (but tedious) to confirm that the composition of τ is $\alpha \xrightarrow{w} \beta$ where

$$\alpha(p') \triangleq \begin{cases} \alpha_2(q/\!/s) & \text{if } \exists\, q, s \ni: p' = w_1(p/\!/q)/\!/s \\ \alpha_1(p') & \text{otherwise} \end{cases}$$

$$\beta(p') \triangleq \begin{cases} \beta_2(q) & \text{if } \exists\, q \ni: p' = p/\!/q \text{ and } \beta_2(q) \in O, \\ \beta_1(p/\!/w_2(q)/\!/s) & \text{if } \exists\, q \in \mathcal{W}(\beta_2) \text{ and } s \in \mathcal{P} \ni: p' = p/\!/q/\!/s \text{ and} \\ & \qquad p/\!/w_2(q)/\!/s \in \mathcal{D}(\beta_1), \\ \beta_2(q) & \text{if } \exists\, q \in \mathcal{W}(\beta_2) \ni: p' = p/\!/q \text{ and} \\ & \qquad p/\!/w_2(q) \notin \mathcal{D}(\beta_1), \\ \beta_1(p') & \text{otherwise} \end{cases}$$

$$w(p') \triangleq \begin{cases} w_1(p/\!/w_2(q)/\!/s) & \text{if } \exists\, q \in \mathcal{W}(\beta_2) \text{ and } s \in \mathcal{P} \ni: p' = p/\!/q/\!/s \text{ and} \\ & \qquad p/\!/w_2(q)/\!/s \in \mathcal{D}(\beta_1), \\ w_1(p/\!/q_1)/\!/q_2 & \text{if } \exists\, q \in \mathcal{W}(\beta_2) \text{ and } q_1, q_2 \ni: p' = p/\!/q, \\ & \qquad w_2(q) = q_1/\!/q_2, \text{ and } p/\!/q_1 \in \mathcal{W}(\beta_1), \\ w_1(p') & \text{otherwise} \end{cases}$$

□

The definition of a type-closed rewrite system is all that is needed to yield the following lemma:

Lemma 4.2 *If $r_1 = \alpha \xrightarrow{w} \beta$ is a rewrite rule in a type-closed system R and $p \in \mathcal{W}(\beta)$, then for all rules $r_2 \in R$ and positions q, there exists a rule $r_1' \sim r_1$ such that: $\langle r_2, w(p)//q\rangle\langle r_1', \varepsilon\rangle \supset \langle r_1, \varepsilon\rangle\langle r_2, p//q\rangle$.*

Rewrites in non-intersecting subtrees can be exchanged, since their requirements and effects are completely independent:

Lemma 4.3 *Let $\langle r_1, p_1\rangle\langle r_2, p_2\rangle$ be a valid rewrite sequence. If p_1 is not a prefix of p_2 and p_2 is not a prefix of p_1, $\langle r_2, p_2\rangle\langle r_1, p_1\rangle \equiv \langle r_1, p_1\rangle\langle r_2, p_2\rangle$.*

We can now prove proposition 4:

Proof We construct a series of rewrite sequences $\phi^0, \phi^1, \ldots, \phi^k$, each covering ϕ, such that

(*) ϕ^i is of the form $\phi_1^i \cdots \phi_n^i \phi_*^i$ where ϕ_j^i only has rewrites at positions with j as their head.

Define $\phi^0 = \phi$. $\phi^0 \supset \phi$, and it satisfies (*) with $\phi_j^0 = \varepsilon, \phi_*^0 = \phi$.

Suppose ϕ^i is not in CNF at ε. Let $\langle r_1, p_1\rangle \cdots \langle r_m, p_m\rangle$ be the applications in ϕ_*^i. Let j be the smallest integer such that $\langle r_1, p_1\rangle \underbrace{\cdots}\langle r_j, p_j\rangle$ fails to satisfy condition (2) of definition 7. Let $\tilde{\phi}_*^i = \langle r_1, p_1\rangle \cdots \widehat{\langle r_j, p_j\rangle} \cdots \langle r_m, p_m\rangle$. Due to lemma 4.1, either j is 1 or p_j contains some wildcard position of the composition of $\langle r_1, p_1\rangle \cdots \langle r_{j-1}, p_{j-1}\rangle$ as a prefix; therefore, according to lemma 4.2, there exists some $r_j' \sim r_j$ and position p such that $\langle r_j', p\rangle\tilde{\phi}_*^i$ covers ϕ_*^i; thus $\phi_1^i \cdots \phi_n^i\langle r_j', p\rangle\tilde{\phi}_*^i$ covers ϕ^i.

If $p = \varepsilon$, let $\phi_j^{i+1} = \phi_j^i$ and $\phi_*^{i+1} = \langle r_j', p\rangle\tilde{\phi}_*^i$. Otherwise, $p = l//q$ for some integer l. According to lemma 4.3, $\phi_1^i \cdots \phi_l^i\langle r_j, p\rangle \cdots \phi_n^i\phi_*^{i+1} \equiv \phi_1^i \cdots \phi_n^i\langle r_j, p\rangle\phi_*^{i+1}$. Let $\phi_k^{i+1} = \phi_k^i$ for $k \neq l$, $\phi_l^{i+1} = \phi_l^i\langle r_j, p\rangle$, and $\phi_*^{i+1} = \tilde{\phi}_*^i$.

Let $\phi^{i+1} = \phi_1^{i+1} \cdots \phi_n^{i+1}\phi_*^{i+1}$; $\phi^{i+1} \supset \phi$ and satisfies (*).

On each step of this construction, one rewrite application in ϕ_*^i either moves from a non-empty position to the empty position or is moved into some ϕ_l^{i+1}. Therefore, the construction must terminate at some point with ϕ^k in CNF at ε. Recursively applying the construction to the ϕ_i^k and the tail of ϕ_*^k produces the desired τ. □

Recall that our solution for SET-REACHABILITY involves a bottom-up traversal of the tree that computes all of the interesting rewrite sequences applicable at a given position. In a CNF rewrite sequence, these sequences are called *compositional local rewrite sequences*:

Definition 9 *Let τ be a valid CNF rewrite sequence. If $\tau = \tau_1 \cdots \tau_n \tau_*$ satisfying the conditions in Definition 7, then the* **compositional local rewrite sequence** *(LRS) assigned by τ to a position p is defined by $C(\tau, p)$, where*

(1) $C(\tau, \varepsilon) \triangleq \tau_*$, *and*

(2) $C(\tau, i//p) \triangleq C(\tau_i@i, p)$.

Pelegrí has devised an algorithm for computing all of the possible local rewrite sequences, and this algorithm is easily adapted for typed rewrite systems and compositional local rewrite sequences. Unfortunately, sometimes there are an infinite number of sequences, even when many of them are unnecessary for solving REACHABILITY. We therefore further constrain the sequences we consider by insisting that they be *efficient:*[4]

Definition 10 *Let G be a set of trees and let τ be a rewrite sequence in CNF of the form $\tau_1 \cdots \tau_n \tau_*$ satisfying the conditions in Definition 7. τ is* **efficient with respect to G** *if*

(1) there is no CNF rewrite sequence τ_*' shorter than τ_* such that, for all trees t with $\tau_*(t) \in G$, $\tau_*'(t) \in G$, and

(2) each τ_i is efficient with respect to $\{\, t@i \mid \tau_*(t) \in G \,\}$.

Proposition 5 *Let R be a rewrite system in $\mathcal{A}$-normal form and let S be a subset of $S_{\mathcal{A}}$. Let $G = \cup_{s \in S} L(s)$, and let t be a tree. If there is a rewrite sequence $\phi \in R$ with $\phi(t) \in G$, there is a rewrite sequence $\tau \in R$ efficient with respect to G with $\tau(t) \in G$.*

Proof Let $\tau = \tau_1 \cdots \tau_n \tau_*$ be a minimal length rewrite sequence in CNF satisfying the conditions in Definition 7, with $\tau(t) \in G$. Such a sequence must exist, since ϕ is covered by some CNF sequence.

Suppose τ is not efficient with respect to G. Then either

[4]This is stricter than Pelegrí's definition of "efficient".

(1) there is a τ'_* shorter than τ_* such that, for all trees t with $\tau_*(t) \in G$, $\tau'_*(t) \in G$, or

(2) there is some τ_i that is not efficient with respect to $\{ t'@i \mid \tau_*(t') \in G \}$.

(1) is impossible, since it implies that $\tau_1 \cdots \tau_n \tau'_*(t) \in G$, contradicting the choice of τ. If (2) is true, then there is some shorter sequence τ'_i that can replace τ_i and still result in a valid rewrite sequence τ' applicable to t. $\mathcal{A}(\tau'(t))$ is independent of t and τ'_i; it only depends on the output pattern of τ_*, and therefore $\tau'(t) \in G$, again contradicting the choice of τ. □

Therefore, in solving SET-REACHABILITY, if the goal set corresponds to a set of states of $\mathcal{A}$, we can restrict our search to rewrite sequences efficient with respect to the goal set and still be assured of finding a solution sequence if there is one.

Given a set of local rewrite sequences, Pelegrí showed how to modify David Chase's algorithm for pattern matching [1] to compute all possible rewrite sequences composed from the local rewrite sequences. We end this section by showing how to compute all possible rewrite sequences that occur in efficient rewrite sequences; Pelegrí's algorithms are then used to construct the SET-REACHABILITY solver.

Definition 11 *Given a rewrite system R in $\mathcal{A}$-normal form and a set G of goal states in $S_{\mathcal{A}}$, construct the following sets:*

$$
\begin{aligned}
O_0 &= \{ L(s)[] \mid s \in G \}, \\
I_0 &= \emptyset, \\
U_0 &= \{ \varepsilon \}, \\
O_{i+1} &= O_i \cup \{ \varrho @ j \mid \varrho \in I_i, j \in \mathcal{D}(\varrho) \}, \\
I_{i+1} &= I_i \cup \{ \varrho \mid \exists \tau \in U_i, \beta' \in O_i, \text{ and positions } p_1, \ldots, p_n \ni: \\
&\qquad \tau \text{ has composition } \alpha \xrightarrow{w} \beta, \\
&\qquad \{ p_1, \ldots, p_n \} = \mathcal{W}(\beta) \cap \mathcal{D}(\beta'), \\
&\qquad \varrho = \alpha \overset{@w(p_1)}{\longleftarrow} \beta'(p_1) \cdots \overset{@w(p_n)}{\longleftarrow} \beta'(p_n), \text{ and} \\
&\qquad L(\beta) \cap L(\beta') \neq \emptyset \} \\
&\quad \cup \{ \gamma \mid \gamma \text{ is a transition leading to } s, \\
&\qquad \text{where } L(s)[] \in O_i \},
\end{aligned}
$$

$$\begin{aligned}
U_{i+1} &= U_i \cup \{\, \tau = \langle r, \varepsilon\rangle \tau' \mid \tau' \textit{ is in CNF, every LRS assigned by } \tau' \\
&\qquad \textit{is in } U_i, \textit{ and } \tau \textit{ is efficient with respect to} \\
&\qquad L(\varrho) \textit{ for some } \varrho \in O_i \,\}, \\
O &= \cup_i O_i, \\
I &= \cup_i I_i, \\
U &= \cup_i U_i.
\end{aligned}$$

The ***useful local rewrite sequences*** *are the members of the set U, and the* ***extended pattern set of*** *$\langle R, G\rangle$ is the union of I and O.*

Proposition 6 *Let R be a rewrite system in $\mathcal{A}$-normal form, and G a set of goal states in $S_{\mathcal{A}}$. If τ is an efficient rewrite sequence with respect to $L(w)$ for some w in G, the local rewrite sequences assigned by τ are useful local rewrite sequences with respect to R and G.*

Proof The detailed proof is quite long; we sketch the main ideas here. We will show that, for any pattern ϱ in O, any rewrite sequence efficient with respect to $L(\varrho)$ is composed of local rewrites sequences in U.

We proceed by induction on the length of τ. Let τ_p be the local rewrite sequence assigned at p by τ, and define the sets $G_\varepsilon = L(w), G_{p/i} = \{\, t@i \mid \tau_p(t) \in G_p \,\}$. Since τ is efficient with respect to G_ε, τ_p is efficient with respect to G_p. It can be shown by induction that G_p is equivalent to $L(\varrho)$ for some output pattern $\varrho \in O$; combining this with the fact that local rewrite sequence τ_p is composed of efficient local rewrite sequences which are in U by the induction hypothesis, the construction of U_{i+1} from U_i ensures that τ_p is also in U. $\Box$

Note that the construction of U may be infinite. CRSTNA has no test for this possibility, and therefore may fail to terminate. I believe that such a test can be constructed based on the ideas discussed in Section 2.3. The extended pattern set constructed simultaneously with U is an adequate replacement for the extended pattern set defined by Pelegrí; it is used in the table construction process.

Optimality of the rewrite sequence found by CRSTNA has been lightly treated in this section, since most of the complication arises in the work done by Pelegrí. If each rule has an associated non-negative cost, and the cost of a rewrite sequence is the sum of the costs of the rules in the sequence, then the definition of *efficient* can be modified to require a sequence with lowest cost, rather than a shortest sequence; combined with Pelegrí's work, CRSTNA then finds a least cost sequence leading to a tree in the goal set.

3.2 Internalizing the specification

According to the theory in the previous section, the wildcards used by a rewrite system are severely restricted: they must all correspond to individual states in a single BFSA. But CRSTNA's specification language allows wildcards to be arbitrary members of RECOG. In this section we show how the rewrite systems defined previously are equivalent in power to those written in CRSTNA's specification language, and show the motivation for the definition of a type-closed rewrite system.

Proposition 7 *Let $\mathcal{R}$ be the set of all rewrite systems R such that R is in $\mathcal{A}$-normal form for some BFSA $\mathcal{A}$. Let $\mathcal{R}^*$ be the set of rewrite systems in which every wildcard is in RECOG.*

$\mathcal{R}$ is as powerful as $\mathcal{R}^*$, i.e., for every $R^* \in \mathcal{R}^*$ there is a corresponding $R \in \mathcal{R}$ such that, for every tree t and rewrite sequence $\tau^* \in R^*$, there is a rewrite sequence $\tau \in R$ with $\tau(t) = \tau^*(t)$.

Proof Let $R^* \in \mathcal{R}^*$, and $\mathcal{A}$ be a BFSA that simultaneously recognizes every wildcard in R^*, i.e., a wildcard in R^* is equal to $\cup_i L(s_i)$ for some subset $\{\, s_1 \ldots s_n \,\}$ of $S_{\mathcal{A}}$. That some appropriate $\mathcal{A}$ exists is a basic result of BFSA theory.

Let $r^* = \alpha^* \xrightarrow{w} \beta^*$ be a rewrite rule in R^*. We will construct a set of rules R_{r^*} such that, if $r^*(t) = t'$, there exists a rule $r \in R_{r^*}$ with $r(t) = t'$; the union of the sets R_{r^*} for all rewrite rules in R^* forms the desired R.

Let $\{p_1, \ldots, p_n\} = \mathcal{W}(\alpha^*)$, and S_i be the set of states corresponding to $\alpha(p_i)$. Then

$$R_{r^*} = \{\alpha \xrightarrow{w} \beta^* \mid \alpha = \alpha^* \xleftarrow{@p_1} L(s_1)[] \cdots \xleftarrow{@p_n} L(s_n)[] \\ \text{for some } \langle s_1, \ldots, s_n\rangle \in S_1 \times \cdots \times S_n\}.$$

□

The definition of a type-closed rewrite system arises from the notion of a rewrite system with recognizable wildcards in which every type is closed under the system.

Definition 12 *A set of trees T is* **closed under a rewrite system** *R if, for all trees $t \in T$, for all rewrite sequences τ in R applicable to t, $\tau(t) \in T$.*

Proposition 8 *Let R^* be a rewrite system in which every wildcard is a recognizable set closed under R^*. Let R be the rewrite system corresponding to R^* according to Proposition 7. R is type-closed.*

Thus, given a rewrite system R^* whose types are recognizable sets closed under R^*, Proposition 7 shows how to construct a type-closed rewrite system equivalent to R^*, allowing us to use the theory in the previous section to solve SET-REACHABILITY.

Unfortunately, the type system R specified by the user in terms of type-instances may not be closed under the rewrite system. CRSTNA obtains a closed type system by solving SET-REACHABILITY for a related rewrite system R' defined by the following specification:

- Type names in R are treated as nullary operators in R'.
- For each type instance x is-a y, R' has a rule $x' \rightarrow y[]$ where variables in x are replaced with the names of their types to yield x'.
- For each rule $x \rightarrow y$ in R, R' has a rule $y' \rightarrow x'$ where variables in x and y are replaced with the names of their types to yield x' and y'.
- R' has a single type g, distinct from all types in R.
- For each type name x in R, R' has a type instance $x[]$ is-a g.

For example, given the specification

$$\begin{array}{rcl} \mathit{ident}[] & \text{is-a} & \mathit{expr} \\ 1[] & \text{is-a} & \mathit{expr} \\ +[X : \mathit{expr}[], Y : \mathit{expr}[]] & \text{is-a} & \mathit{expr} \end{array}$$

$$+[X : \mathit{expr}[], 1[]] \rightarrow \mathit{add1}\,[X : \mathit{expr}[]]$$

we would create the system

$$\begin{array}{rcl} \mathit{ident}[] & \rightarrow & \mathit{expr}[] \\ 1[] & \rightarrow & \mathit{expr}[] \\ +[\mathit{expr}[], \mathit{expr}[]] & \rightarrow & \mathit{expr}[] \\ \mathit{add1}\,[\mathit{expr}[]] & \rightarrow & +[\mathit{expr}[], 1[]] \end{array}$$

Now if t is a type name in R, let the type named by t be the set of all trees that can be rewritten into t by R'; in our example, trees with *add1* at the root and *expr*'s as children can be rewritten into *expr*, in addition to those originally specified with type-instances.

The resulting type system is the minimal system that contains the original type-instances and is closed under R. Determining these types is simply the SET-REACHABILITY problem for the rewrite system R' with goal type g; the algorithms in this report can solve SET-REACHABILITY for any rewrite system without variables, and therefore can solve this problem for R'.

3.3 Programming details

The system is written in about 2000 lines of Common Lisp. It is a relatively straightforward implementation of the theory presented in this paper; very little optimization was done. The following basic design choices were made:

- Sets of patterns are used heavily in Chase's algorithm; the most common operations on them are intersection, union, and equality testing. For these reasons, an ordered set representation is used, with patterns hashed to ensure that equal patterns are represented by the same data object. Profiling suggests that this was a good choice.

- The implementation of Definition 11 is both important and difficult; CRSTNA spends most of its time in this construction. Since compositions are expensive to compute, CRSTNA keeps a list of compositions that may eventually satisfy the condition required to add a composition to U. It is not clear whether or not this was a good choice; CRSTNA has severe problems with space, but the alternative of repeatedly composing rules, discovering that they are not yet valid, garbage collecting the composition, and composing the rules again does not sound too promising. At the very least, the compositions that are formed should be carefully screened. The current implementation forms any composition that is in bottom-up normal form using useful rewrite sequences that have already been discovered; this is excessive, since in many cases the "useful" rewrite sequences are useless in the particular context.

- The automaton produced by solving SET-REACHABILITY for the rewrite system R' described in the previous section is more powerful than is necessary to determine types; it contains information needed to construct a rewrite sequence from a given tree to its type name, while all that is needed for the type automaton is to know if such a sequence exists. Thus, the automaton has more states than are strictly required.

  This is disastrous; the primary reason CRSTNA has space problems is that, given a rule with the pattern $+[X : t[], Y : t[]]$, CRSTNA has to make s^2 copies where s is the number of states corresponding to t. For rewrite systems the size of machine descriptions, it is imperative to minimize the type automaton. CRSTNA has this capability, and it was used in the experiments described below. A representation which optimized the amount of space occupied by a rule at the expense of time manipulating rules might also be a win.

4 Table sizes for a code generator

Types make possible the specification of SET-REACHABILITY, eliminating the need for semantic actions in many applications. They also give the rewrite system designer greater control over when rewrites will be applied. The cost of this greater control is larger tables that must simultaneously track input patterns and tree types.

In order to see if the table sizes were likely to be practical, CRSTNA was used to generate tables for a code generator for the Motorola 68000. The machine description was written from scratch, assuming a low-level intermediate representation (e.g., using machine types and explicit addressing calculations) as input. Costs were not used. The description was written in about a day, and has 149 rewrite rules and 178 type instances; a comparable machine description written for BURS uses 520 rewrite rules. The ability to use types and variables is therefore significant in the ease of writing the machine description. Unfortunately, CRSTNA requires more than 25 megabytes to process this description, which is beyond the capability of our current hardware configuration.

In order to generate at least some tables, the rewrite rules describing register-to-register moves and the register-indexed-indirect addressing modes were removed from the machine description. The register-to-register move instructions greatly increase the number of states in the type automaton corresponding to a single type, making the number of local rewrite sequences explode; the register-indexed-indirect addressing modes involve large patterns with three operands, all of which may be rewritten into registers. Since CRSTNA generates all compositions of local rewrite sequences, this is a deadly combination; it might be possible for a better implementation (which only generated and saved compositions that might eventually become useful) to handle the full description without undue amounts of space.

Even so, the results are not promising for machine descriptions. The smaller description, without the register moves and register-indexed-indirect addressing modes, yields a table with 2,964 states; this compares with 362 states for the untyped version. A simple uncompacted representation of the transition tables occupies about 7,000 bytes; Pelegrí does not give uncompacted transition table sizes, but his compacted transition tables occupy around 4,000 bytes.

Most of the table size is taken up by the state descriptions in Pelegrí's tables, and so the factor of ten increase in the number of states (for a machine description that is less powerful) seems likely to make typed rewrite systems unsatisfactory for code generation. In addition, the increased table generation time makes it very difficult to experiment with descriptions. CRSTNA must solve an untyped system just to get the type automaton, before moving on to the (slower) typed system, so this gap in table-generation speeds is an inherent part of the process.

Thus, the experiments suggest that CRSTNA is unsuitable for code generation. The extra power provided by types makes writing the machine description more convenient, but at the expense of a large increase in table size and in table generation time. Both of these problems may be solvable: the former may either evaporate as memories get larger, or might be solved by noting similarities among the states and storing their representations in a compact form, while the latter could be made manageable by a carefully optimized implementation. But in the meantime, untyped systems seem to provide a better combination of table size and generation speed, with an acceptable amount of difficulty in the description writing.

5 Future work

The main principle in CRSTNA's design has been to accept as many rewrite systems as possible while yielding an automaton that can find an optimal rewrite sequence in a bottom-up traversal of the tree. The following problems still need to be solved before leaving this general area of design.

- Information regarding the possible input trees should be taken into consideration, so that unbounded local rewrite sequences that only occur for impossible input trees can be ignored.
- A humanly understandable characterization of the rewrite systems that can be accepted should be produced. At the very least, a decidability test should be found.
- SET-REACHABILITY currently states that a rewrite sequence must be found. Although this is useful for applications that still need to attach semantic actions to specific rewrites, one of our goals is to eliminate the need for such actions. A different casting of SET-REACHABILITY that only requires an output tree, and not the rewrite sequence producing it, may allow a cleaner theory and/or easier solutions to some of the other problems in this section.
- Different design choices in the Common Lisp implementation need to be explored, to discover if the problems in handling machine descriptions are inherent in the method or simply an artifact of the implementation. In particular, a careful implementation of Definition 11 could vastly increase the size of the description that CRSTNA could process (although it would have no effect on the resulting table sizes).

6 Conclusion

It is possible, in principle, to solve SET-REACHABILITY very efficiently, given a fixed rewrite system and set of goal trees. Unfortunately, table sizes and table generation times make these results impractical for rewrite systems the size of machine descriptions, given our current technology. It is unclear whether further research could significantly improve either table size or generation time.

Our new algorithm for determining the efficient local rewrite sequences greatly increases the number of rewrite systems that can be handled, with or without types; rewrites that increase the size of the tree may be acceptable in some cases, and any rewrite system without variables can be handled. This result may wind up being the most important contribution of CRSTNA.

References

[1] David R. Chase. An improvement to bottom-up tree pattern matching. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, 1987.

[2] Eduardo Pelegrí-Llopart. Rewrite systems, pattern matching, and code generation. Ph.D. Dissertation UCB/CSD 88/423, U. C. Berkeley, 1988.

After printing of the technical report, an error was discovered in the proof of Proposition 5; in fact, the proposition is false. The false portion of the result is not used in the report; this insert should be read in place of the bound pages 15 and 16, up to Definition 11 which remains unchanged.

Recall that our solution for SET-REACHABILITY involves a bottom-up traversal of the tree that computes all of the interesting rewrite sequences applicable at a given position. The interesting sequences are a subset of the *local rewrite sequences*:

Definition 9 *Let τ be a valid CNF rewrite sequence. If $\tau = \tau_1 \cdots \tau_n \tau_*$ satisfying the conditions in Definition 7, then the* **local rewrite sequence** *(LRS) assigned by τ to a position p is defined by $C(\tau, p)$, where*

(1) $C(\tau, \varepsilon) \triangleq \tau_*$, and

(2) $C(\tau, i/\!/p) \triangleq C(\tau_i @ i, p)$.

Given a set U of local rewrite sequences, Pelegrí has shown how to modify David Chase's algorithm for pattern matching [1] to compute all possible rewrite sequences that assign only local rewrite sequences in U. Our goal is to find a set U such that the set of rewrite sequences assigning LRS's in U covers the set of rewrite sequences that rewrite trees into the goal set. U should be as small as possible; in particular, it should be finite. Pelegrí's algorithm finds all local rewrite sequences that do not loop and that produce a subtree which might eventually be written into the goal tree; unfortunately, this can still involve infinite sets of local rewrite sequences (e.g., sequences that expand and then contract a tree). We obtain a finite set by only considering the set of sequences that are *efficient*:[1]

Definition 10 *Let G be a set of trees and let τ be a local rewrite sequence. τ is* **efficient with respect to G** *if there is no local rewrite sequence τ' shorter than τ such that, for all trees t with $\tau(t) \in G$, $\tau'(t) \in G$.*

If an LRS is interesting only because it rewrites some trees into a particular goal set G, we can ignore it if it is not efficient with respect to G; some other efficient LRS can always take its place. CRSTNA uses the following construction of U:

[1]This is stricter than Pelegrí's definition of "efficient".