INFORMATION AND CONTROL 71, 186–217 (1986)

Function Definitions in Term Rewriting and Applicative Programming*

CHILUKURI K. MOHAN AND MANDAYAM K. SRIVAS

Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794-4400

The frameworks of unconditional and conditional Term Rewriting and Applicative systems are explored with the objective of using them for defining functions. In particular, a new operational semantics, *Tue-Reduction*, is elaborated for conditional term rewriting systems. For each framework, the concept of *evaluation* of terms invoking defined functions is formalized. We then discuss how it may be ensured that a function definition in each of these frameworks is meaningful, by defining restrictions that may be imposed to guarantee *termination*, *unambiguity*, and *completeness* of definition. The three frameworks are then compared, studying when a definition may be translated from one formalism to another.

1. INTRODUCTION

In this paper we study the use of *Term Rewriting Systems* and *Conditional Term Rewriting Systems* for defining and evaluating functions. We formulate sufficient criteria for ensuring that programs written using these formalisms define functions, and compare such function definitions to those in an applicative language. We study the transformation of a function definition expressed in one language to an equivalent one in another.

The formalism of Term Rewriting Systems (TRS) [HuOp80] has traditionally been used for theorem proving in equational theories, eg., [Hsia82]. Equality axioms have been expressed as systems of rewrite rules, completed by the use of the Knuth–Bendix procedure [KnBe70], which has also been applied to several aspects of equational reasoning, eg., [Ders83]. Properties of TRS like *termination* and *confluence* have been investigated and several useful sufficient criteria and results obtained [Ders79, JoLR82, Lesc84]. TRS which satisfy these properties have been used for solving the word problem: two terms are repeatedly reduced to their *normal forms* which must be identical for terms equal within the equational theory being simulated by the TRS [Huet80].

* Research supported by National Science Foundation Grant MCS8401624.

0019-9958/86 $3.00

Attempts have been made to increase the expressive power of TRS by incorporating conditions into rewrite rules, and different approaches have been used to characterize the operational semantics of Conditional Term Rewriting Systems (CTRS) [BrDJ78, Lank79, BeKl82, PlEE82, Kapl83, Remy83]. The *T*-Unification procedure for finding complete sets of unifiers for terms with respect to a rewriting system [Fay79, Hull80] has been extended to a semidecision procedure for the computation of unifiers for terms with respect to a CTRS [Kapl84]. We propose a mechanism called *Tue-Reduction* for evaluations using CTRS containing rewrite rules much more general than in previous work. In our formalism, the condition and the right-hand side (rhs) of a rule may contain variables not occurring in the left-hand side (lhs). Conditions in rules may contain some literals that are to be *satisfied* and others that are to be proved *unsatisfiable.* Occurrence of equality predicates is treated as the problem of *T-unification* of the terms being equated.

Some attempts have been made to formulate a programming language using a combination of term rewriting and logic programming techniques [Ders84, DePl85]. The term rewriting formalism is similar to the more traditional applicative language formalism, eg., LISP [McCa60, Hend80], in that a computation in both is performed by value rather than by assignment. But the term rewriting language is more flexible because it allows nondeterminism, and is also more declarative. This increased flexibility can be a boon as well as a bane. It makes the language more expressive, but also makes it harder to ensure program correctness. Hence, it is useful to formulate sufficient conditions that ensure that a program correctly defines a function. These conditions should be easy to check as well as easy to comply with by a programmer.

Term rewriting techniques have been used to specify abstract data types, and the properties of such specifications have been studied in detail [KaMu82, Sriv82, Pada83, KaSr85]. Attempts have been made to characterize the desirable properties of abstract data type specifications, eg., *Sufficient Completeness* [GuHo78], *Full Specification* [Muss80], *Definition Principle* [HuHu80], *Relative Completeness* [Remy82]. In this paper, we use similar techniques to formulate *Fully Defined* functions in rewrite systems. We outline methods of verifying *termination*, *unambiguity*, and *completeness* properties for function definitions in different formalisms. We generalize the ideas of well-founded orderings for verifying termination of unconditional systems [Ders85], hierarchical conditional specifications [ReZh84] and *fair* conditional systems [Kapl84] to *pseudo-hierarchies* which allow some degree of circularity in the definitions without sacrificing termination. The idea of *nonoverlapping* rewrite systems [HoOd82] is generalized to enable verification that a defined function maps each constructor term argument to a unique result. An inference mechanism is

described to verify that a function is totally defined. This is similar to the methods used in [Thie83, Koun85, JoKo85] for specifications using unconditional TRS.

In the next section, we present the formalisms of term rewriting, conditional term rewriting, and a style of applicative programming. We describe the operational semantics of each of these formalisms, and show how each can be used to define functions. These formalisms are then compared, addressing the question of when a function definition may be translated from one formalism to another. Conclusions of the comparative study are presented in the last section.

2. Description of Formalisms

In this section, we illustrate the definition of functions in (unconditional) Term Rewriting Systems (TRS), Conditional Term Rewriting Systems (CTRS), and Applicative (LISP-like) Systems.

In each formalism, function definitions use certain *Primitive* operators, consisting of *constructors*, *extractors*, *structure-testers*, and *error* operators.

Primitive Terms are terms containing primitive operators and variables.

Constructors are a set of functions which are sufficient to denote any object of the relevant data type. The constructors may include some nullary *constants* denoted by the symbols *a*, *b*, *c*,..., possibly subscripted. *Constructor terms* are terms in which every symbol occurring is a constructor or a variable.

Extractors are unary functions which return components of the structure of the argument, and are like inverses of constructor functions. *Extractor terms* are terms in which every symbol occurring is an extractor or a variable.

Structure-Testers are Boolean-valued unary functions which may be used to check whether the argument has any particular structure, i.e., whether it has a specific constructor as outermost symbol.

Nullary *Error* operators yield a distinguished "error" value.

Example. The list data structure has constructors NIL and CONS, extractors CAR and CDR, structure-testers ISNULL and ISCONS, and an error-operator **err**. Extractors, structure-testers, and constructors are related by the following axioms which, along with axioms for the Booleans (**true**, **false**) and operations on them, comprise primitive operations embedded at a lower level of specification in any function definition system using the list data structure. Here as elsewhere, we implicitly assume that every defined function invoked on **err** returns **err**, unless otherwise specified.

(1) CAR(CONS(x, y)) = x, CAR(NIL) = **err**,

(2) CDR(CONS(x, y) = y, CDR(NIL) = **err**,

(3) ISNULL(NIL) = **true**, ISNULL(CONS(x, y) = **false**,

(4) ISCONS(NIL) = **false**, ISCONS(CONS(x, y)) = **true**.

In any programming formalism, it is desirable (in most cases) that a function definition satisfies three properties:

(P_1) evaluation of terms invoking the defined function on ground constructor term arguments must *terminate*;

(P_2) the definition must be *unambiguous*, i.e., every evaluation of the same term must yield the same result; and

(P_3) the definition must be *complete*, i.e., evaluation must be possible for **every** invocation of the defined function on ground constructor term arguments.

Our goal is to see how these properties are satisfied using the three formalisms for function definitions under consideration. First, each formalism for function definition and the corresponding evaluation technique are outlined. Restrictions needed to ensure properties (P_1), (P_2), (P_3) are successively described, and the discussion is summarized, illustrating how functions may be *Fully Defined* in each formalism.

2.1. *Term Rewriting Definitions*

We define *Term Rewriting Systems* (TRS) and describe their operational semantics. We show how TRS may be used to define functions, and describe the restrictions required for function definitions in TRS to be meaningful, i.e., for the definition to satisfy the properties (P_1), (P_2), (P_3) stated earlier.

2.1.1. *Reduction using TRS*

DEFINITION. A *Term Rewriting System* is a set of rewrite rules, each rule consisting of an ordered pair of terms (lhs $\rightarrow$ rhs).

When a subterm s of a given term t matches with the lhs of a rewrite rule under a substitution σ (i.e., $s = (\text{lhs})\sigma$), t may be rewritten to a new term $t[s \leftarrow (\text{rhs})\sigma]$ obtained by replacing the occurrence of s in t by the rhs of the rule, to which the substitution σ has been applied. This is called *reduction* of the term t using the rewrite rule lhs $\rightarrow$ rhs.

EXAMPLE. The one-rule term rewriting systems $(x + y) + z \rightarrow x + (y + z)$

associates terms to the right. Given the term $(a+(b+c))+d$, the following is a reduction sequence using this TRS

$$(a+(b+c))+d \rightarrow a+((b+c)+d) \rightarrow a+(b+(c+d)).$$

2.1.2. *Defining functions in TRS*

A function definition in a TRS contains a set of rewrite rules in each of which the leading symbol of the lhs term is the function being defined.

DEFINITION. The *Term Rewriting Evaluation* (TR-evaluation) of a term is its repeated reduction using the rules of a TRS until the resulting term is a constructor term or is irreducible.

2.1.2.1. *Termination.* A rewrite system is said to be *Terminating* if every possible sequence of reductions from every term is finite. The problem of checking whether an arbitrary unconditional TRS is terminating is undecidable [HuLa78]. Since free variables are arbitrarily instantiable, one necessary condition to ensure termination is that every variable in the rhs of a rule must also appear in the lhs.

A necessary and sufficient condition for a TRS to be finitely terminating is the existence of a well-founded ordering $>$ among ground terms compatible with respect to the operations in the language, such that $(\text{lhs})\sigma > (\text{rhs})\sigma$ in every ground instantiation $(\text{lhs} \rightarrow \text{rhs})\sigma$ of each rule of the TRS. Termination orderings for TRS have been formulated and extensively studied in [Plai78a, Plai78b, Ders79, GuKM82, JoLR82, Lesc84, Ders85]. One convenient and practically useful way of ensuring such an ordering, (allowing recursion to a limited extent), is to organize the function definitions into a *pseudo-hierarchy* with different "levels" containing sets of rules defining distinct functions.

DEFINITION. A term rewriting definition of a function is a *pseudo-hierarchy* if every subterm of the rhs of each rule is either

(a) a term with leading symbol denoting a function hierarchically lower than the function being defined, or

(b) an invocation of a function at the same level (as the function being defined) on an argument-tuple which is $\ll$ (smaller than) the argument-tuple of the lhs of the rule, in some well-founded ordering $\ll$.

PROPOSITION. *A function defined by a pseudo-hierarchical term rewriting system terminates for every invocation of the function defined on constructor terms.*

Several examples of well-founded termination orderings can be found in the

references cited above. The *multiset subterm ordering* defined below is one such example:

(i) If t is a proper subterm of s, then $t < s$.

(ii) Let S and T be the multisets corresponding to the argument-tuples $\bar{s}_l$, $\bar{t}_l$, respectively (as in a rule $f(\bar{s}_l) \to f(\bar{t}_l)$). Then $T \ll S$ iff either T is empty and S is nonempty, or

$$\forall\, t \in T, \qquad [[t \in S \wedge (T - t \ll S - t)] \vee \exists\, s \in S[t < s]].$$

EXAMPLE. Consider the TRS defining *Pairs*, a function that recursively yields a list of pairs of members of its argument lists.

Pairs(NIL, z) $\to z$

Pairs(CONS(x, y), NIL) $\to$ CONS(x, y)

Pairs(CONS(x, y), CONS(u, v)) $\to$ CONS(*Pairs*(x, u), *Pairs*(y, v))

The first two rules are clearly terminating since the rhs contains no recursive invocation of *Pairs*. For the last rule, we apply twice the method described above to verify termination:

$$S = \{\text{CONS}(x, y), \text{CONS}(u, v)\}$$

and

$$T_1 = \{x, u\} \ll S,$$
$$T_2 = \{y, v\} \ll S.$$

Every element in T_1, T_2 is a subterm of some term in S, and we conclude that the function definition is finitely terminating.

2.1.2.2. *Unambiguity.* A term rewriting definition of a function is *unambiguous* if the TR-evaluation of any ground term yields the same term, irrespective of the reduction sequence used. Thus, if t_1 and t_2 are any two distinct terms obtained by reducing the same ground term s, then there must exist rewrite sequences from t_1 as well as t_2 yielding the same term t. Using the Knuth–Bendix procedure, it is possible to determine whether any terminating TRS is *ground-confluent*. We propose a sufficient, simpler method of checking unambiguity for function definitions with the free constructor assumption (cf. the *nonoverlapping* property [HoOd82]).

If the arguments of lhs's of definition rules are irreducible, a sufficient condition to ensure unambiguous TR-evaluation is the requirement that any term $f(\bar{t})$ match with the lhs of at most one rule defining the function f, i.e., for any term $f(\bar{t})$ there is a unique rule lhs $\to$ rhs such that $\exists\, \sigma \cdot [f(\bar{t}) = (\text{lhs})\sigma]$. For non-unifiability of lhs's of definition rules with constructor

term arguments to be a sufficient criterion for unambiguity, constructors must be "free": every object must be represented by a unique ground constructor term, and there must be no nontrivial equivalences between ground constructor terms. Hence unambiguity is guaranteed if the lhs's of no two definition rules are unifiable, and every proper subterm of the lhs in each definition rule is an irreducible (free) constructor term.

Hence, if the proper subterms of the lhs of a definition rule are reducible, non-unifiability is no longer a sufficient criterion for unambiguity: more complex criteria are required.

For example, let the definition of function f contain the rules $f(s) \to r$ and $f(g(t)) \to t'$, where the term $g(t)$ itself reduces to s using the rules defining g. Although $f(s)$ and $f(g(t))$ are not unifiable, we cannot conclude unambiguity unless we are able to show that r and t' have a common reduct.

Similarly, in integer arithmetic with nonfree constructors 0, *succ*, *pred* related by the rules $\{succ(pred(x)) \to x,\ pred(succ(x)) \to x\}$, if a function is defined by rules $\{f(succ(x)) \to t_1,\ f(0) \to t_2\}$, the lhs's of the two rules are non-unifiable, yet the ground term $f(succ(pred(0)))$ can be reduced to t_1 as well as t_2, depending on which rule is first used for reduction.

2.1.2.3. *Completeness.* A term rewriting definition of a function f is *complete*, if every term $f(\bar{s})$, where $\bar{s}$ is a tuple of ground constructor terms, can be reduced using some definition rule. In the formulation of [JoKo85], the term $f(\bar{x})$ with variable arguments must be *quasi-reducible*. Completeness may be verified by using any of the algorithms given in [Thie83, Koun85]. We outline another similar method using an inference mechanism, analogous to structural induction on the arguments of the function. Completeness is proved if we are able to infer a tuple containing only distinct variables from the tuples of arguments in the lhs's of rules defining the function.

We take pairs of rules and repeatedly infer the union of their domains of definition by applying the inference rule to proper subterms of the lhs's of the defining rules. The inference rules depend on the data structure: for example, for lists, there is just one rule: NIL, CONS$(x, y)/z$.

When the function has several arguments, the inference system needs to be applied to each element of the argument-tuple. For some unifying substitution σ, if $s_i\sigma,\ s_i'\sigma/t_i$ and $\forall\, j \neq i.\ [s_j\sigma = s_j'\sigma]$, then

$$\frac{\langle s_1, s_2, \ldots, s_i, \ldots, s_n\rangle,\ \langle s_1', s_2', \ldots, s_i', \ldots, s_n'\rangle}{\langle s_1\sigma, s_2\sigma, \ldots, t_i, \ldots, s_n\sigma\rangle}.$$

EXAMPLE. We attempt to prove the completeness of the OR function as defined below, using the inference rule **true**, **false**/z for booleans.

$$\mathrm{OR}(y, \textbf{true}) \to \textbf{true},$$

$$\mathrm{OR}(\textbf{false}, \textbf{false}) \to \textbf{false},$$

$$\mathrm{OR}(\textbf{true}, x) \to \textbf{true}.$$

From the two-tuples $\langle y, \textbf{true} \rangle$ and $\langle \textbf{false}, \textbf{false} \rangle$, using the substitution $\sigma = (y \leftarrow \textbf{false})$, we infer $\langle \textbf{false}, z \rangle$. Using this two-tuple and $\langle \textbf{true}, x \rangle$ from the third rule, the completeness of the set of two-tuples of arguments to OR is then concluded. The inference tree is shown below:

$$\dfrac{\dfrac{\langle y, \textbf{true} \rangle, \langle \textbf{false}, \textbf{false} \rangle}{\langle \textbf{false}, z \rangle, \langle \textbf{true}, x \rangle}}{\langle u, v \rangle}$$

2.1.2.4. *Fully Definedness.* We now condense the preceding discussion by describing when a function is *Fully Defined* by rewrite rules, guaranteeing that the function definition satisfies the properties of termination, unambiguity and completeness.

DEFINITION. A function f is *Fully Defined* by the set of rewrite rules $\{f(\bar{s}_1) \to t_1, f(\bar{s}_2) \to t_2, \ldots, f(\bar{s}_n) \to t_n\}$, where each $\bar{s}_i$ comprises of constructor terms, if

(a) all constructors are free, and there are no rewrite rules in which the leading symbol of the lhs is a constructor;

(b) whenever $i \neq j$, $f(\bar{s}_i)$ and $f(\bar{s}_j)$ are not unifiable;

(c) $\forall\, i.\ [Vars(t_i) \subseteq Vars(f(\bar{s}_i))]$;

(d) $\bar{x}$, a tuple of distinct variables, can be inferred from $\{\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_n\}$; and

(e) the definition rules constitute a pseudo-hierarchy under the multiset subterm ordering, and lower level non-primitive functions used are Fully Defined.

EXAMPLE. A function of two arguments is defined below to check whether the first argument is a list which is longer than the second.

(1) $Longer(\mathrm{NIL}, z) \to \textbf{false}$,

(2) $Longer(\mathrm{CONS}(x, y), \mathrm{NIL}) \to \textbf{true}$,

(3) $Longer(\mathrm{CONS}(x, y), \mathrm{CONS}(u, v)) \to Longer(y, v)$.

First, every ground constructor term is its own normal form since there are no nontrivial equivalences between terms built with CONS and NIL.

Second, the lhs's of no two rules unify. The argument tuple of the first

rule, $\langle \text{NIL}, z \rangle$, is disjoint from the other two since its first argument NIL is not unifiable with CONS(x, y) which is the first argument of the other rules. The argument tuple of the second rule is disjoint from the third since NIL, the second argument, is not unifiable with the corresponding argument CONS(u, v) of the third rule.

Third, in each rule, variables of the rhs are all contained in the lhs. The definition is proved to be complete by inferring a two-tuple of variables from the arguments of the lhs's of the rewrite rules. From the arguments of *Longer* in the rules (2) and (3), we infer

$$\frac{\langle \text{CONS}(x, y), \text{CONS}(u, v) \rangle, \langle \text{CONS}(x, y), \text{NIL} \rangle}{\langle \text{CONS}(x, y), z \rangle}.$$

We then apply the inference rule to the arguments from rule (1) and the newly inferred term-tuple,

$$\frac{\langle \text{NIL}, z \rangle, \langle \text{CONS}(x, y), z \rangle}{\langle z', z \rangle}.$$

Last, the definition rules constitute a pseudo-hierarchy since the only occurrence of *Longer* in the rhs of a rule occurs in rule (3) on arguments that are proper subterms of the corresponding arguments of the lhs. Therefore the rhs is "smaller" than the lhs in the well-founded subterm ordering, and the function definition satisfies the termination property.

Hence the function *Longer* is Fully Defined in this formalism.

2.2. *Conditional Definitions*

In this section, we present a formalism of conditional term rewriting and its operational semantics. We illustrate how this formalism can be used to define functions. We then address the issues of termination, unambiguity and completeness for conditional function definitions, and combine these ideas to formulate the property of "Fully Definedness" of conditional definitions.

2.2.1. *Tue-Reduction*

DEFINITION. A *Conditional Term Rewriting System* (CTRS) is a set of conditional rewrite rules. Each conditional rewrite rule consists of a *condition* and a pair of terms (lhs, rhs) and is written [*condition*::lhs $\rightarrow$ rhs].

The operational semantics of CTRS has earlier been described in several

different ways. We now define one approach, *Reduction T-unifying equalities*, (where T is the given CTRS), abbreviated as *Tue-Reduction.*

The condition may contain literals with variables not present in the lhs. In earlier work [DePl85], such variables have been assumed to be existentially quantified, and an attempt is made to find a substitution for these variables that *satisfies* the literals in the *condition.* We wish to ensure the specification of "complete" definition systems, in which some rule can always be used to reduce a term headed by the function being defined. If some rule is applicable when a particular literal is satisfiable, we also need some rule to be applicable in the complementary case, when such a literal is unsatisfiable. Hence, we allow *conditions* in rules to have literals that must be proved *unsatisfiable.*

In our formulation, the *condition* in each rule will consist of the two parts **sat**[*pos*] and **unsat**[*neg*], where *pos* and *neg* are sets of literals, some of which may be equality predicates, that must respectively be proved satisfiable and unsatisfiable. If $\bar{x} = [Vars(pos) - Vars(\text{lhs})]$ and $\bar{y} = [Vars(neg) - Vars(\text{lhs}) - Vars(pos)]$, where $pos = \{P_1,..., P_m\}$ and $neg = \{N_1,..., N_n\}$, then the intended logical meaning of the rule

$$\textbf{sat}[P_1,..., P_m]\ \textbf{unsat}[N_1,..., N_n] :: \text{lhs} \to \text{rhs} \quad \text{is}$$

$$(\exists\, \bar{x} \cdot [(P_1 \wedge \cdots \wedge P_m) \wedge \forall\, \bar{y}.(\neg N_1 \wedge \cdots \wedge \neg N_n)]) \supset (\text{lhs} = \text{rhs}).$$

This makes it easy to specify the function for conditions complementary to the above rule, using rules like

$$\textbf{unsat}[P_1,..., P_m] :: \text{lhs} \to \text{rhs}_1$$

and

$$\textbf{sat}[P_1,..., P_m, N_1,..., N_n] :: \text{lhs} \to \text{rhs}_2.$$

For convenience, a positive literal whose variables are all contained in the lhs will be kept in the **sat** part of the condition, while a negative literal ($\neg P$) whose variables are all contained in the lhs will be represented by keeping the corresponding atom(P) in the **unsat** part. When empty, the **sat** or **unsat** part of a rule may be omitted.

If a term is to be Tue-Reduced, we must first find a subterm that matches with the lhs of some rule of the CTRS. The condition of the rule is then tested, after replacing non-equality literals by equivalent equality predicates.

An attempt is made to satisfy the equality predicates occurring in **sat**[*pos*] by jointly conditionally T-unifying the two sides of each equality, using a modified version (cf. [Kapl84, Huss85]) of the T-Unification

algorithm described in [Hull80], where T is the given set of conditional rewrite rules, and only the non-lhs variables in the **sat** part of the rule are substituted for. If this attempt succeeds and a joint unifier is found for the equalities in the **sat** part of the rule, the equalities in the **unsat**(*neg*) part of the condition must then be proved unsatisfiable for *every* substitution of the variables of the given term or the rule. This is done by attempting to jointly T-unify the sides of each equality, substituting for even the variables of the term given to be reduced (as illustrated in Example 3).

Tue-Reduction occurs if there is no such T-unifier, for some satisfying substitution of the **sat** part of the condition. As a result, the appropriately instantiated rhs of the rule replaces the subterm in the given term that had originally matched with the lhs of the rule. A more precise description of Tue-Reduction follows:

(1) All variables of rules are renamed to avoid name conflicts with the given term. For uniformity, each predicate A occurring in the condition of each rule is written $[A \stackrel{?}{=} \textbf{true}]$, while each negative literal $\neg A$ in the condition is written $[A \stackrel{?}{=} \textbf{false}]$.

(2) A rule $\textbf{sat}[P_1,..., P_m]\ \textbf{unsat}[N_1,..., N_m]$::lhs $\rightarrow$ rhs is chosen such that lhs matches with some subterm t of the given term, i.e., $t = (\text{lhs})\sigma_0$, for some substitution σ_0.

(i) If no such rule is found, we conclude that the given term is irreducible using the set of rules being considered.

(ii) If found, the matching substitution σ_0 is applied to the rest of the rule.

(3) Now the equality literals in the **sat** part of the rule are jointly T-Unified, so that $\forall\, i$, $[s_i\sigma \rightarrow {}^*n_i,\ t_i\sigma \rightarrow {}^*n_i]$, where each literal $(P_i)\sigma_0$ of the **sat** part is the equality predicate $(s_i \stackrel{?}{=} t_i)$, and n_i are irreducible, for some new most general T-unifying substitution σ for the variables $[\bigcup_i Vars(P_i)]$ in the **sat** part of the rule. Note that σ is not an instance of a substitution obtained in a previous execution of this step.

(i) If no such T-unifier σ is found, we conclude that this rule cannot be used to Tue-Reduce the given term, and attempt Tue-Reduction using some other rule, beginning from step (2).

(ii) If found, the T-Unifier σ obtained above is applied to the rest of the rule.

(4) Equality literals in the **unsat** part of the rule are now jointly T-Unified, so that $\forall\, i$, $[s_i\rho \rightarrow {}^*n_i,\ t_i\rho \rightarrow {}^*n_i]$, where each literal $(N_i)\,\sigma_0\sigma$ of the **unsat** part is the equality predicate $(s_i \stackrel{?}{=} t_i)$, and n_i are irreducible, for some substitution ρ for the variables in $Vars(t) \cup [\bigcup_i Vars(N_i\sigma_0\sigma) - \bigcup_j Vars(P_j\sigma_0)]$.

(i) If no such T-Unifier ρ is found, we conclude that Tue-Reduction can occur: in the given term, the subterm t may be replaced by $[(\text{rhs})\,\sigma_0\sigma]$.

(ii) If any T-unifier ρ is found, we conclude that for substitution σ for variables in the **sat** part, this rule cannot be used to Tue-Reduce the given term: we rebegin from step (3) and try to find another suitable σ.

We illustrate the above description using three examples. The first shows how equality predicates are handled in our approach, the second demonstrates the conditional narrowing of literals, while the third illustrates the Tue-reduction of a term containing variables.

EXAMPLE 1. The term $f(\text{CONS}(a, \text{CONS}(b, \text{CONS}(b, w))))$ is to be Tue-Reduced using the conditional rule

$$\textbf{sat}[z \stackrel{?}{=} \text{CONS}(x, \text{CONS}(v, y))]\ \textbf{unsat}[x \stackrel{?}{=} v] :: f(z) \rightarrow \text{CONS}(x, f(\text{CONS}(v, y))).$$

The matching substitution obtained with the lhs of the rule is

$$\sigma = [z \leftarrow \text{CONS}(a, \text{CONS}(b, \text{CONS}(b, w)))].$$

This substitution transforms the condition of the rule to

$$\textbf{sat}[\text{CONS}(a, \text{CONS}(b, \text{CONS}(b, w))) \stackrel{?}{=} \text{CONS}(x, \text{CONS}(v, y))]\ \textbf{unsat}[x \stackrel{?}{=} v].$$

The first equality is satisfied using the unifier $\rho_1 = [x \leftarrow a,\ v \leftarrow b,\ y \leftarrow \text{CONS}(b, w)]$ and the condition then changes to $\textbf{sat}[\ ]\ \textbf{unsat}[a \stackrel{?}{=} b]$. Non-unifiability and non-narrowability of the constants a and b imply the unsatisfiability of $(a \stackrel{?}{=} b)$. Hence the Tue-Reduction of the given term succeeds and the resulting term is $\text{CONS}(a, f(\text{CONS}(b, \text{CONS}(b, w))))$.

We may now attempt to apply the same rule again to this term, matching the variable z to $\text{CONS}(b, \text{CONS}(b, w))$. The substitution $\sigma_1 = [x \leftarrow b,\ v \leftarrow b,\ y \leftarrow w]$ satisfies the equality $[\text{CONS}(b, \text{CONS}(b, w)) \stackrel{?}{=} \text{CONS}(x, \text{CONS}(v, y))]$, and instantiates the literal in the **unsat** part to $[b \stackrel{?}{=} b]$. Since $(b \stackrel{?}{=} b)$ is trivially satisfiable, and every other substitution satisfying $[\text{CONS}(b, \text{CONS}(b, w)) \stackrel{?}{=} \text{CONS}(x, \text{CONS}(v, y))]$ is an instance of σ_1, the term $\text{CONS}(a, f(\text{CONS}(b, \text{CONS}(b, w))))$ cannot be reduced using this rule.

EXAMPLE 2. There are two constructors (constants) a, b, and two lower level predicates P, Q defined by the conditional rules

(1) **sat**$[x \stackrel{?}{=} a, y \stackrel{?}{=} a] :: P(x, y) \rightarrow$ **true**,

(2) **sat**$[x \stackrel{?}{=} b] :: P(x, y) \rightarrow$ **false**,

(3) **sat**$[y \stackrel{?}{=} b] :: P(x, y) \rightarrow$ **false**,

{i.e., P is **true** only when both arguments are a},

(4) **sat**$[x \stackrel{?}{=} b, y \stackrel{?}{=} b] :: Q(x, y) \rightarrow$ **true**,

(5) **sat**$[x \stackrel{?}{=} a] :: Q(x, y) \rightarrow$ **false**,

(6) **sat**$[y \stackrel{?}{=} a] :: Q(x, y) \rightarrow$ **false**,

{i.e., Q is **true** only when both arguments are b},

(7) **sat**$[P(y, z)]$ **unsat**$[Q(x, v)] :: f(x, z) \rightarrow a$.

We attempt to Tue-Reduce the term $f(a, a)$ using the last rule: on applying the matching substitution $[x \leftarrow a, \ z \leftarrow a]$, the condition becomes **sat**$[P(y, a)]$ **unsat**$[Q(a, v)]$. $P(y, a)$ is satisfiable since it can be conditionally narrowed to **true** using rule (1), with the substitution $[y \leftarrow a]$. $Q(a, v)$ is unsatisfiable: it Tue-Reduces to **false** using rule (5), and cannot be narrowed to **true**. Hence $f(a, a)$ Tue-reduces to a using rule (7).

Similar Tue-reduction on the term $f(x', a)$, however, is not possible using rule (7). The reason is that $Q(x', v)$ is not unsatisfiable: x' is treated as a variable and not a constant. $Q(x', v)$ can be narrowed to **true** using rule (4), but cannot be reduced to **false**.

EXAMPLE 3. We define a function *Filter* which reports an error when given as argument any list containing a non-empty list as the first element.

(1) **sat**$[x \stackrel{?}{=} \text{CONS}(\text{CONS}(u, v), z)] :: \mathit{Filter}(x) \rightarrow$ **err**,

(2) **unsat**$[x \stackrel{?}{=} \text{CONS}(\text{CONS}(u, v), z)] :: \mathit{Filter}(x) \rightarrow x$.

The nonground term *Filter*(CONS(w, NIL)) is itself not reducible using either rule, although narrowable. For example, to reduce this term using either rule, substituting CONS(w, NIL) for the variable x, we need to prove that $[\text{CONS}(w, \text{NIL}) \stackrel{?}{=} \text{CONS}(\text{CONS}(u, v), z)]$ is either satisfiable or unsatisfiable.

(i) To prove satisfiability, w must be treated as a constant: it is not unifiable with CONS(u, v), hence **sat**$[\text{CONS}(w, \text{NIL}) \stackrel{?}{=} \text{CONS}(\text{CONS}(u, v), z)]$ is unprovable and rule (1) is inapplicable.

(ii) For checking unsatisfiability, w must be treated like any other variable, and we have to disprove the unifiability of CONS(w, NIL) and CONS(CONS(u, v), z). However, unification is indeed possible using the substitution $[w \leftarrow \text{CONS}(u, v), z \leftarrow \text{NIL}]$, hence we fail to prove **unsat**$[x \stackrel{?}{=} \text{CONS}(\text{CONS}(u, v), z)]$ and the given term cannot be reduced using rule (2).

2.2.2. *Conditional Function Definitions*

The definition of a function f in a CTRS contains a set of conditional rules $\{C_i :: f(\bar{t}_i) \to s_i\}$. The *Tue-Evaluation* of a term $f(\bar{t})$ is its repeated Tue-Reduction using the rules in the definition of f until the resultant term is a constructor term or is irreducible.

EXAMPLE. The following CTRS defines a function *Longer* which verifies whether its first argument is a list longer than the second.

(1) **unsat**$[x \stackrel{?}{=} \text{CONS}(u, v)] :: \mathit{Longer}(x, y) \to$ **false**,

(2) **sat**$[x \stackrel{?}{=} \text{CONS}(u, v)]$ **unsat**$[y \stackrel{?}{=} \text{CONS}(w, z)] :: \mathit{Longer}(x, y) \to$ **true**

(3) **sat**$[[x \stackrel{?}{=} \text{CONS}(u, x')], [y \stackrel{?}{=} \text{CONS}(v, y'))] :: \mathit{Longer}(x, y) \to \mathit{Longer}(x', y')$.

We now illustrate the Tue-Evaluation of the ground term *Longer*(CONS(a, CONS(b, NIL)), CONS(c, NIL)) using this function definition. With $[x \leftarrow \text{CONS}(a, \text{CONS}(b, \text{NIL})), y \leftarrow \text{CONS}(c, \text{NIL})]$ as the matching substitution, the given term matches with the lhs in each of rules (1), (2), and (3). However, $[\text{CONS}(a, \text{CONS}(b, \text{NIL})) \stackrel{?}{=} \text{CONS}(u, v)]$ and $[\text{CONS}(c, \text{NIL}) \stackrel{?}{=} \text{CONS}(w, z)]$, the literals in the **unsat** parts of the conditions of rules (1) and (2), are satisfiable. Hence rules (1) and (2) cannot be used for Tue-Reduction.

Evaluating the condition in rule (3), satisfying the first equality $[\text{CONS}(a, \text{CONS}(b, \text{NIL})) \stackrel{?}{=} \text{CONS}(u, x')]$ with the unifying substitution $\rho_1 = [u \leftarrow a, x' \leftarrow \text{CONS}(b, \text{NIL})]$, and satisfying the second equality $[\text{CONS}(c, \text{NIL}) \stackrel{?}{=} \text{CONS}(v, y')]$ with $\rho_2 = [v \leftarrow c, y' \leftarrow \text{NIL}]$, we conclude that rule (3) can be used to Tue-Reduce the given term to *Longer*(CONS(b, NIL), NIL).

This term may again be Tue-Reduced: rule (2) is now applicable with the substitution $[x \leftarrow \text{CONS}(b, \text{NIL}), y \leftarrow \text{NIL}]$, since the literal $\text{CONS}(b, \text{NIL}) \stackrel{?}{=} \text{CONS}(u, v)$ is satisfiable and $\text{NIL} \stackrel{?}{=} \text{CONS}(w, z)$ is unsatisfiable. The rhs of rule (2) is the irreducible term **true**, and hence we conclude that the given term *Longer*(CONS(a, CONS(b, NIL)), CONS(c, NIL)) Tue-Evaluates to **true**.

2.2.2.1. *Termination.* To avoid infinite chains of reductions in the evaluation of the condition in a rule, [Remy83] uses hierarchical specifications in which the subterms of the condition of every rule are all defined by rewrite rules at a lower level. However, this is restrictive since it does not allow recursion in the conditions of definition rules. [Kapl84] defines *Fair* CTRS in which all the subterms in the condition and the rhs of every rule have to be smaller than the lhs in some well-founded ordering.

We attempt to give clearer guidelines to a programmer to help define functions using terminating rewrite rules. We modify the above ideas for the specific situations when the CTRS is a function definition, with rules that have lhs's comprising the defined function invoked on constructor term arguments. As we have done for unconditional systems, we require that conditional function definitions be specified as *pseudo-hierarchies*. We allow several layers of rules, each layer containing mutually recursive function definitions.

DEFINITION. A conditional function definition is a *pseudo-hierarchy* if every subterm of the *condition* and the rhs of each rule is either

(a) a term with leading symbol denoting a function hierarchically lower than the function being defined, or

(b) an invocation of a function at the same level (as the function being defined) on an argument-tuple which is $\ll$ (smaller than) the argument-tuple of the lhs of the rule, in some well-founded ordering $\ll$ (e.g., the multiset subterm ordering given in Sect. 2.1.2.1).

PROPOSITION. *A pseudo-hierarchical conditional function definition terminates for every invocation of the defined function on constructor terms.*

2.2.2.2. *Unambiguity.* A conditional function definition is *unambiguous* if Tue-evaluation of any ground term yields a unique result, irrespective of the reduction sequence used. In general, no semidecision procedure can be outlined to check the ground confluence of arbitrary CTRS, but in some restricted cases an extension of the Knuth–Bendix procedure may be used [ReZh85]. We propose a sufficient, simpler method to check unambiguity of conditional definitions, extending the technique outlined earlier (Sect. 2.1.2.2) for checking unambiguity of unconditional term rewriting definitions.

As in Section 2.1.2.2, we assume that in defining a function f:

(i) if f is the leading symbol of the lhs of a rule, all proper subterms of the lhs are constructor terms; and

(ii) distinct constructor terms are not equivalent and cannot be reduced to one another.

We may allow a term to unify with the lhs's of different rules, if it can be ascertained that the *conditions* of these rules do not simultaneously hold. This is ensured if, under the substitution unifying the lhs's of two rules, the satisfaction of some literal in the **sat** part of one rule implies that some literal in the **unsat** part of the other rule must also be satisfied. In other words, the function definition is unambiguous if for every pair of rules

whose lhs's unify using a substitution θ, we have $(p\theta \supset (q\theta)\sigma)$ for some substitution σ, where p is in the **sat** part of one rule and q is in the **unsat** part of the other rule.

Tue-evaluation of any term using a given rule must lead to a unique result. Hence, unambiguity requires that the terms which may be substituted for variables $y_i \in [Vars(\text{rhs}) - Vars(\text{lhs})]$ must be unique. This is ensured if every y_i occurs in an equality predicate $[y_i \stackrel{?}{=} g(\bar{x})]$ in the **sat** part of the rule where $\bar{x} \subseteq Vars(\text{lhs})$ and g is any function, effectively rendering y_i redundant since its occurrences in the rhs may as well be replaced by $g(\bar{x})$.

2.2.2.3. *Completeness.* A conditional function definition of f is *complete* if the term $f(\bar{s})$ can be Tue-reduced using some definition rule, for every tuple of ground constructor terms $\bar{s}$. Analogous to the inference rules for functions defined using unconditional TRS (given earlier in Sect. 2.1.2.3), the basic technique is the use of structural induction over terms with constructors and variables. In addition, we infer that a term is reducible under the disjunction of conditions of different rules, locating identical literals in the **sat** and **unsat** parts of different rules.

First, we obtain the *Guarded tuple starter* (**sat**[*pos*] **unsat**[*neg*]$::\bar{s}$) from each conditional rule **sat**[*pos*] **unsat**[*neg*]$::f(\bar{s}) \to t$. Multiple occurrences of the same variable in the lhs of any rule are replaced by new variables, and equality predicates (equating the new variables) are introduced in the **sat** part of the guarded tuple starter obtained from the rule. This enables us to check that a different rule has a complementary condition: for example, from the (unconditional) non-linear rule

$$f(\text{CONS}(x, \text{CONS}(x, y))) \to f(\text{CONS}(x, y))$$

we obtain the guarded tuple starter (**sat**$[x \stackrel{?}{=} z]::\text{CONS}(x, \text{CONS}(z, y))$) which is complementary to the starter obtained from the rule

$$\textbf{unsat}[x \stackrel{?}{=} z]::f(\text{CONS}(x, \text{CONS}(z, y))) \to \text{CONS}(x, f(\text{CONS}(z, y))).$$

Similarly, every occurrence of a non-variable subterm in the lhs may be replaced by a new variable and a corresponding equality predicate introduced in the **sat** part of the corresponding starter. This would be needed to infer completeness from complementary rules like

$$f(t) \to r_1 \qquad \text{(giving the starter } \textbf{sat}[x \stackrel{?}{=} t]::x\text{); and}$$
$$\textbf{unsat}[x \stackrel{?}{=} t]::f(x) \to r_2.$$

From these starters, the inference system then derives new *Guarded tuples*, each of which is a 3-tuple denoted (**sat**[*pos*] **unsat**[*neg*]$::$*termtuple*) (cf.

contextual terms [ReZh85]). The function definition is complete if we succeed in using the inference rules for the relevant data types to derive the guarded tuple (**sat**[ ] **unsat**[ ]::$\bar{x}$), i.e., a tuple of variables with empty conditions, or something trivially equivalent. For example, note that (**sat**[$y \stackrel{?}{=} z$] **unsat**[$w \stackrel{?}{=} w$]::x) may be considered equivalent to (**sat**[ ] **unsat**[ ]::x). Inferences can be made for subterms as well as instantiations of any of the components of a guarded tuple.

For example, using the inference system for list structures, we may make the inferences

$$\frac{\left[\begin{array}{l}(\textbf{sat}[P]\ \textbf{unsat}[N]::\mathrm{CONS}(u, \mathrm{NIL})), \\ \quad (\textbf{sat}[P]\ \textbf{unsat}[N]::\mathrm{CONS}(u, \mathrm{CONS}(x, y)))\end{array}\right]}{(\textbf{sat}[P]\ \textbf{unsat}[N]::\mathrm{CONS}(u, z))},$$

$$\frac{\left[\begin{array}{l}\textbf{sat}[(t \stackrel{?}{=} \mathrm{NIL}), P]\ \textbf{unsat}[N]::\bar{s}), \\ \quad (\textbf{sat}[(t \stackrel{?}{=} \mathrm{CONS}(y, z)), P]\ \textbf{unsat}[N]::\bar{s})\end{array}\right]}{(\textbf{sat}[(t \stackrel{?}{=} x, P]\ \textbf{unsat}[N]::\bar{s})}.$$

If the same term P occurs in the **sat** part of one rule and the **unsat** part of another (otherwise identical) rule, then the guarded tuples of these rules may be merged, eliminating P. This gives us another inference scheme, which take identical terms from the **sat** and **unsat** parts of otherwise identical guarded tuples and cancel them:

$$\frac{(\textbf{sat}[P, Q]\ \textbf{unsat}[N]::s), (\textbf{sat}[Q]\ \textbf{unsat}[P, N]::s)}{(\textbf{sat}[Q]\ \textbf{unsat}[N]::s)}.$$

Like other inferences (cf. Sect. 2.1.2.3), this inference scheme also extends to subterms and instances of guarded tuples.

In a strict sense, a function definition may be completely defined although we may not reach this conclusion using this inference technique: for example, we are unable to infer that $f(x)$ is always Tue-reducible using the definition below:

$$g(x) \rightarrow \textbf{false}$$

$$\textbf{unsat}[g(x)]::f(x) \rightarrow \mathrm{rhs}(x)$$

However, we do enable the task of checking completeness of specification using *simple* syntactic criteria, without attempting to perform actual reduction of terms using function definitions. The specification method related to our inference technique is also *modular*: other functions may be independently defined and altered, without changing the completeness property

of the function in consideration (whose definition has been verified to be complete using the inference mechanism).

PROPOSITION. *If the conditional definition of a function f is complete for every possible definition of other fully defined functions (used in defining f), then, using the inference mechanism outlined above, the guarded tuple* **sat**[ ] **unsat**[ ]::$\bar{x}$ *can be inferred from the guarded tuple starters obtained from the rules defining f.*

2.2.2.4. *Fully Definedness.* We now summarize the above discussion and list a set of veriable criteria to ensure that a conditional definition is terminating, unambiguous, and complete.

DEFINITION. Let the set of all conditional rewrite rules whose lhs is headed by f be

$$\{\mathbf{sat}[P_1]\ \mathbf{unsat}[N_1]::f(\bar{s}_1)\to t_1,\ldots,\ \mathbf{sat}[P_n]\ \mathbf{unsat}[N_n]::\ f(\bar{s}_n)\to t_n\},$$

where $\bar{s}_i$ are tuples of constructor terms. The function f is *Fully Defined* in the conditional term rewriting formalism if

(a) all constructors are free, and there are no rewrite rules in which the leading symbol of the lhs is a constructor;

(b) if the lhs's of two rules are unifiable using substitution σ, then there exists a literal P in the **sat** part of one rule, a literal Q in the **unsat** part of the other rule, and a substitution ρ such that $(P\sigma \supset (Q\sigma)\rho)$;

(c) every variable $y \in [Vars(\text{rhs}) - Vars(\text{lhs})]$ occurs in the *condition* of the rule in an equality predicate of the form $y \stackrel{?}{=} g(\bar{x})$ where $\bar{x} \subseteq Vars(\text{lhs})$;

(d) a tuple of distinct variables (equivalently, the guarded tuple **sat**[ ] **unsat**[ ]::$\bar{z}$) can be inferred from the guarded tuple starters obtained from the definition rules; and

(e) the definition rules constitute a pseudo-hierarchy, in which lower level functions used are Fully Defined.

EXAMPLE. We define a unary function *Del* on lists which returns a list obtained by deleting consecutive repetitions of members of the argument of *Del*.

(1) $Del(\text{NIL}) \to \text{NIL}$,

(2) $Del(\text{CONS}(x, \text{NIL})) \to \text{CONS}(x, \text{NIL})$,

(3) $Del(\text{CONS}(x, \text{CONS}(x, y))) \to Del(\text{CONS}(x, y))$,

(4) $\mathbf{unsat}[x \stackrel{?}{=} v]::Del(\text{CONS}(x, \text{CONS}(v, y))) \to \text{CONS}(x, Del(\text{CONS}(v, y)))$.

First, all constructor terms are irreducible.

Second, the only rules with unifiable lhs's are (3) and (4), and the application of their unifier $[v \leftarrow x]$ transforms the condition of rule (4) to the trivially false $\mathbf{unsat}[x \stackrel{?}{=} x]$. Hence (4) is inapplicable whenever (3) is applicable.

Third, in each rule, the rhs does not contain any variables other than those of the lhs.

Completeness: We transform rule (3) into a conditional rule with linear lhs, from which we obtain the guarded starter $\mathbf{sat}[x \stackrel{?}{=} z]::\mathrm{CONS}(x, \mathrm{CONS}(z, y))$. Using the guarded starters obtained from rules (3) and (4) with variables appropriately renamed, we infer:

$$\frac{\mathbf{sat}[x \stackrel{?}{=} z]::\mathrm{CONS}(x, \mathrm{CONS}(z, y)),\ \mathbf{unsat}[x \stackrel{?}{=} z]::\mathrm{CONS}(x, \mathrm{CONS}(z, y))}{\mathrm{CONS}(x, \mathrm{CONS}(z, y))}.$$

From this inferred term and the starter obtained from the second rule, we infer:

$$\frac{\mathrm{CONS}(x, \mathrm{NIL}),\ \mathrm{CONS}(x, \mathrm{CONS}(z, y))}{\mathrm{CONS}(x, w)}$$

Using the starter obtained from the first rule, we then infer:

$$\frac{\mathrm{NIL},\ \mathrm{CONS}(x, w)}{z}.$$

The term thus obtained is a variable, unaccompanied by any condition. Hence, we conclude that the definition is complete.

Termination: The only defined function invoked in the rules is *Del* itself. The conditions in the rules do not involve any recursion. In rule (3), the argument of *Del* in the rhs is $\mathrm{CONS}(x, y)$ which is a subterm of the argument $\mathrm{CONS}(x, \mathrm{CONS}(x, y))$ in the lhs. Similarly, in rule (4), the argument $\mathrm{CONS}(v, y)$ in the rhs is a subterm of the argument $\mathrm{CONS}(x, \mathrm{CONS}(v, y))$ in the lhs. Thus, the only recursive invocations of *Del* in the rhs terms have arguments which are strictly smaller than the arguments in the lhs, under the subterm ordering, which is well founded. Hence this is a terminating conditional definition of the function *Del*.

All five criteria being satisfied, we conclude that function *Del* is Fully Defined by the above set of rules.

2.3. *Applicative Function Definitions*

An *Applicative Function Definition* (cf. [McCa60, Hend80]), consists of a function-name, a list of variables ($\bar{x}$), and a *conditional expression* which contains no variables other than those in $\bar{x}$. A *conditional expression*,

headed by a special function COND, is a list of pairs $[c_i \rightarrow r_i]$ of conditions and (resultant) terms. Thus, each definition is of the form

$$[f(x_1,..., x_m)\ \mathrm{COND}([c_1 \rightarrow r_1][c_2 \rightarrow r_2]\cdots[c_n \rightarrow r_n])].$$

The Applicative definition of a function f may be used for the *Ap-evaluation* of a term $f(\bar{t})$ as follows, using the normal order semantics. The variables that follow the function-name in the Applicative definition are bound to the terms that are the top-level arguments of the given term, and occurrences of each such variable in the definition are replaced by the terms thus bound to them. The conditions c_i in the pairs $[c_i \rightarrow r_i]$ in the conditional expression are sequentially *Ap*-evaluated until some c_j *Ap*-evaluates to **true**. The *Ap*-evaluation of the given term $f(\bar{t})$ is the same as the *Ap*-evaluation of the corresponding resultant term r_j (the latter part of the pair). If none of the conditions c_i in the conditional expression *Ap*-evaluates to **true**, or if the leading symbol of a term does not have an Applicative definition, (e.g., a constructor term), we may assume that *Ap*-evaluation leaves the term unaltered.

Termination of applicative definitions is ensured by the termination of the *Ap*-evaluation of c_i as well as r_i for every $[c_i \rightarrow r_i]$ in the conditional expression. As in the case of conditional definitions, pseudo-hierarchical restrictions may be imposed to guarantee termination: subterms of the conditional expression are allowed to invoke only (a) functions being defined at the same level of specification, invoked on 'smaller' arguments; and (b) other functions defined at lower levels.

The former contraint is difficult to check: it is nontrivial to decide when we may consider some arguments to be smaller than others. In the Applicative definition $[f(x_1,..., x_m)\ \mathrm{COND}([c_1 \rightarrow r_1][c_2 \rightarrow r_2]\cdots [c_n \rightarrow r_n])]$, if either c_i or r_i contains a subterm $f(s_1,..., s_m)$, then we require that $[s_1,..., s_m] \ll [x_1,..., x_m]$, in some ordering $\ll$ among tuples, where the unknown variables are treated as constants. Following the common programming practice of defining functions using recursive invocations on extractor terms, we allow for nonsubterm orderings: we assume that for any term t and for any function g which is either an extractor or a structure-tester, we have $[g(t) \ll t]$.

Occurrence of equality and inequality literals in conditions can help refine the above criterion. If c_i contains an equality $[x_j = t_j]$ then we need to verify that $[s_1,..., s_m] \ll [x_1,..., t_j,..., x_m]$. If c_i contains an inequality $x_j \neq t_j$, where t_j is a constructor term, we may assume that every condition c_k that follows c_i (i.e., whenever $k > i$) includes the equality $x_j = t_j$.

Sequential evaluation of the conditions ensures that each condition c_k is tested only if the preceding conditions $c_1,..., c_{k-1}$ have not evaluated to **true**. This guarantees *unambiguity* of *Ap*-evaluation.

Completeness may be ensured by having **true** as the condition in the last

pair of the conditional expression. When *Ap*-evaluating a given term, if none of the previous conditions *Ap*-evaluates to **true**, then the last condition always succeeds, and *Ap*-evaluation of the corresponding (last) resultant term ensues.

EXAMPLE.

$$[Del(z)\ \mathrm{COND}([(z = \mathrm{NIL}) \rightarrow \mathrm{NIL}]$$
$$[(\mathrm{CDR}(z) = \mathrm{NIL}) \rightarrow z]$$
$$[(\mathrm{CAR}(z) = \mathrm{CAR}(\mathrm{CDR}(z))) \rightarrow Del(\mathrm{CDR}(z))]$$
$$[\mathbf{true} \rightarrow \mathrm{CONS}(\mathrm{CAR}(z), Del(\mathrm{CDR}(z)))])].$$

The function defined above eliminates duplicate adjacent occurrences of elements of a list: *Ap*-evaluation of the term

$$Del(\mathrm{CONS}(\mathrm{NIL}, \mathrm{CONS}(\mathrm{CONS}(\mathrm{NIL}, \mathrm{NIL}),$$
$$\mathrm{CONS}(\mathrm{CONS}(\mathrm{NIL}, \mathrm{NIL}), \mathrm{NIL}))))$$

yields

$$\mathrm{CONS}(\mathrm{NIL}, Del(\mathrm{CONS}(\mathrm{CONS}(\mathrm{NIL}, \mathrm{NIL}),$$
$$\mathrm{CONS}(\mathrm{CONS}(\mathrm{NIL}, \mathrm{NIL}), \mathrm{NIL})))),$$

using the last part of the conditional expression. The *Ap*-evaluation of this term yields $\mathrm{CONS}(\mathrm{NIL}, Del(\mathrm{CONS}(\mathrm{CONS}(\mathrm{NIL}, \mathrm{NIL}), \mathrm{NIL})))$, which finally yields $\mathrm{CONS}(\mathrm{NIL}, \mathrm{CONS}(\mathrm{CONS}(\mathrm{NIL}, \mathrm{NIL}), \mathrm{NIL}))$, a list in which no two consecutive elements are the same.

The function *Del* has been *Fully Defined* above. Termination is assured because recursive invocations of *Del* take $\mathrm{CDR}(z)$ as argument, which is smaller than z in the ordering we assume for lists, since CDR is an extractor function. Unambiguity is guaranteed for all Applicative function definitions. Completeness is given by the predicate **true** in the last part of the conditional expression.

3. TRANSFORMATIONS BETWEEN FORMALISMS

In this section, we outline the methods that may be adopted to translate function definitions from one formalism to another. We also indicate restrictions in the definitions required to ensure translatability.

3.1. *Unconditional to Conditional Rewrite Rule Formalism*

Translation from an unconditional term rewriting definition to a con-

ditional term rewriting definition is straightforward. Each rule of the definition is prefixed by the condition **sat**[ ]**unsat**[ ].

EXAMPLE. The rule $(f(s) \rightarrow t)$ translates to (**sat**[ ]**unsat**[ ]$::f(s) \rightarrow t$).

3.2. *Unconditional Rules to Allicative Formalism*

A first attempt to accomplish the translation of a function definition from the unconditional term rewriting formalism to the Applicative formalism involves the introduction of equalities between variables and arguments of the *lhs*'s of rules into the conditional expression. For example, the set of rules $\{f(s_1) \rightarrow t_1, f(s_2) \rightarrow t_2, ..., f(s_n) \rightarrow t_n\}$ might be translated to $[f(x)\ \mathrm{COND}([(x = s_1) \rightarrow t_1], ..., [(x = s_n) \rightarrow t_n])]$.

However, the result may not adhere to the applicative style since the terms s_i may contain variables other than those that immediately follow the function-name. This must be avoided in Applicative definitions. We now describe a more elaborate algorithm for the translation of rewrite rules in which all proper subterms of the lhs are constructor terms. Each definition rule corresponds to one of the $[c_i \rightarrow r_i]$ pairs in the conditional expression, constructed as follows.

First, new variables are introduced as the arguments of the function defined. Each occurrence of a variable in the lhs of the rule is replaced by an extractor term over the new variables, depending on its position in the lhs of the given rewrite rule. We obtain r_i from the rhs of the rule, replacing variables by the extractor terms obtained above. When proper subterms of the lhs are constructor terms, the use of structure-tester functions in the applicative definition achieves a purpose analogous to matching a given term agains the lhs of the rewrite rule.

EXAMPLE. We illustrate the translation of the term rewriting definition for *Longer* that was given in section 2.1.2.4.

Term Rewriting Definition:

(1) *Longer* (NIL, z) $\rightarrow$ **false**,

(2) *Longer* (CONS(x, y), NIL) $\rightarrow$ **true**,

(3) *Longer* (CONS(x, y), CONS(u, v)) $\rightarrow$ *Longer*(y, v).

Applicative Definition:

[*Longer*(x, y) COND([ISNULL(x) $\rightarrow$ **false**]

[(ISCONS(x) $\wedge$ ISNULL(y)) $\rightarrow$ **true**]

[(ISCONS(x) $\wedge$ ISCONS(y)) $\rightarrow$ *Longer*(CDR(x), CDR(y))])].

If a rule contains multiple occurrences of a variable, the corresponding condition c_i in the Applicative definition must also contain a conjunction of equality predicates equating the extractor terms corresponding to each occurrence.

EXAMPLE. We translate the rewrite rule $(halve(\mathrm{CONS}(x, x)) \rightarrow x)$ into the Applicative (partial) function definition

$$[halve(z)\ \mathrm{COND}([(\mathrm{CAR}(z) = \mathrm{CDR}(z)) \rightarrow \mathrm{CAR}(z)])].$$

3.3. *Conditional to Unconditional Rewrite Rule Formalism*

When the condition of a conditional rule contains variables not occurring in the lhs, their quantification cannot in general be handled by unconditional rewrite systems. Translation from conditional to unconditional rewriting definition rules is possible only in restricted cases, which we now consider.

Restriction 3.3.1.

The first restriction is that the conditions of rules contain only a **sat** part which contains only structure-tester terms and equality predicates on primitive terms. Each such equation is "solved," i.e., each equality is transformed into the form $z_i \stackrel{?}{=} t_i$ where z_i is a variable and t_i is a primitive term not containing occurrences of z_i. This transformation occurs by

(i) inverting every extractor function, introducing new variables as required (e.g., $[\mathrm{CAR}(x) \stackrel{?}{=} \mathrm{CDR}(y)]$ becomes $[x \stackrel{?}{=} \mathrm{CONS}(z, w),\ y \stackrel{?}{=} \mathrm{CONS}(u, z)]$);

(ii) replacing structure-testers and variables by corresponding constructor terms (e.g., $[\mathrm{ISCONS}(x)]$ becomes $[x \stackrel{?}{=} \mathrm{CONS}(y, z)]$); and

(iii) splitting equalities between unifiable constructor terms into subterm equalities (e.g., $[\mathrm{CONS}(u, v) \stackrel{?}{=} \mathrm{CONS}(s, \mathrm{CONS}(t, w))]$ becomes $[u \stackrel{?}{=} s,\ v \stackrel{?}{=} \mathrm{CONS}(t, w)]$).

The equality predicates can then be eliminated: the translated unconditional rule is then obtained by replacing (for each equality predicate $z_i \stackrel{?}{=} t_i$) every occurrence of z_i in the rest of the rule by the corresponding term t_i. In the above process, if any equality between non-unifiable constructor terms is obtained (e.g., $[\mathrm{CONS}(x, y) \stackrel{?}{=} \mathrm{NIL}]$, or $[x \stackrel{?}{=} \mathrm{CONS}(x, y)]$), the rule may be discarded.

EXAMPLE. The conditional rule $\mathbf{sat}[x \stackrel{?}{=} \mathrm{CAR}(y),\ z \stackrel{?}{=} \mathrm{CONS}(x, y)] :: f(z) \rightarrow f(y)$ may be first transformed to

sat$[y \stackrel{?}{=} \mathrm{CONS}(x, v),\ z \stackrel{?}{=} \mathrm{CONS}(x, y)] :: f(z) \to f(y)$ and then to **sat**$[z \stackrel{?}{=} \mathrm{CONS}(x, \mathrm{CONS}(x, v))] :: f(z) \to f(\mathrm{CONS}(x, v))$. This rule may then be translated to the unconditional rule $f(\mathrm{CONS}(x, \mathrm{CONS}(x, v))) \to f(\mathrm{CONS}(x, v))$.

Restriction 3.3.2.

In the other restricted case of conditional definition rules considered, the condition contains only variables contained in the lhs. Then, the elimination of a condition from a rewrite rule is possible by introducing dummy function symbols (cf. [DePl85]). For example, we may translate the conditional rule **sat**$[P(\bar{x})]$**unsat**$[Q(\bar{x})] :: f(\bar{x}) \to r(\bar{x})$ into the following TRS:

(1) $f(\bar{x}) \to f'(P(\bar{x}), Q(\bar{x}), \bar{x})$,

(2) $f'(\mathbf{true}, \mathbf{false}, \bar{x}) \to r(\bar{x})$.

This has the effect of forcing condition evaluation before the rhs of the corresponding conditional rule is evaluated. It is also necessary to provide **else** clauses, of the form

$$f'(\mathbf{false}, \mathbf{false}, \bar{x}) \to s_1(\bar{x}),$$

$$f'(\mathbf{false}, \mathbf{true}, \bar{x}) \to s_2(\bar{x}),$$

$$f'(\mathbf{true}, \mathbf{true}, \bar{x}) \to s_3(\bar{x}),$$

which should be generated by translating other conditional rules if the original conditional definition is complete. When the condition $P(\bar{t})$ does not hold, (or when $Q(\bar{t})$ holds), the conditional rule *cannot* be used for reduction, while the translated rule (1) can be used to reduce $f(\bar{t})$ to $f'(P(\bar{t}), \bar{t})$, a term originally absent from the language.

EXAMPLE. Two of the rules of the conditional function definition *Del* in section 2.2.2.4 are translated together to unconditional rules:

Conditional Rules:

(i) $Del(\mathrm{CONS}(x, \mathrm{CONS}(x, y))) \to Del(\mathrm{CONS}(x, y))$,

(ii) **unsat**$[x \stackrel{?}{=} v] :: Del(\mathrm{CONS}(x, \mathrm{CONS}(v, y))) \to \mathrm{CONS}(x, Del(\mathrm{CONS}(v, y)))$.

Unconditional Rules:

(1) $Del(\mathrm{CONS}(x, \mathrm{CONS}(z, y))) \to f'(eq(x, z), x, z, y)$,

(2) $f'(\mathbf{true}, x, z, y) \to Del(\mathrm{CONS}(z, y))$,

(3) $f'(\mathbf{false}, x, z, y) \to \mathrm{CONS}(x, Del(\mathrm{CONS}(z, y)))$.

Note. Rules (1), (2), (3) are together equivalent to

$$Del(\text{CONS}(x, \text{CONS}(z, y))) \rightarrow \textbf{if } eq(x, z) \textbf{ then } Del(\text{CONS}(z, y)) \textbf{ else } \text{CONS}(x, Del(\text{CONS}(z, y)))$$

The definition for the underlying *eq* predicate is:

(4) $eq(\text{NIL}, \text{NIL}) \rightarrow \textbf{true}$,

(5) $eq(\text{CONS}(x, y), \text{NIL}) \rightarrow \textbf{false}$,

(6) $eq(\text{NIL}, \text{CONS}(x, y)) \rightarrow \textbf{false}$,

(7) $eq(\text{CONS}(x, y), \text{CONS}(u, v)) \rightarrow \textbf{and}([eq(x, u)], [eq(y, v)])$.

3.4. *Conditional Rules to Applicative Formalism*

In the general case in which the conditions in the conditional rules contain new variables not occurring in the lhs, no straightforward translation to the applicative formalism is possible, since satisfiability and unsatisfiability of arbitrary literals cannot be handled in the latter. Again, we consider restricted cases.

Restriction 3.4.1.

When all the variables used in a conditional rule have occurrences in the lhs, translation from a conditional definition to an applicative definition may be carried out in a manner very similar to the translation from unconditional rules described in Section 3.2. The **unsat** part may be eliminated by negating all the literals in it and moving them to the **sat** part of the condition. New variables are introduced as arguments of the function, occurrences of other variables are replaced by extractor terms, and equality predicates are obtained by equating the extractor terms corresponding to multiple occurrences of the same variable. Each condition in the conditional expression of the applicative definition is obtained by taking the conjunction of these equalities and the literals in the condition of the given rewrite rule.

Restriction 3.4.2.

Another restricted case is when variables not in the lhs occur only in equality literals of the forms $[c_0(\bar{y}) \stackrel{?}{=} g(\bar{x})]$ or $[c_1(\bar{x}) \stackrel{?}{=} c_2(\bar{x})]$ in the **sat** part of a rule, where c_i are primitive terms, $\bar{y} \cap Vars(\text{lhs}) = \varnothing$, and $\bar{y} \cap \bar{x} = \varnothing$. Each such equation is "solved," variables $\bar{y}$ are eliminated, and the conditional rules translated into unconditional definition rules in which

proper subterms of the lhs are constructor terms, as illustrated earlier in Restriction 3.3.1 (Sect. 3.3). Using the technique outlined in Section 3.2, these unconditional rules are translated into an applicative definition.

EXAMPLE. Given below is the conditional definition of a function that flattens a given list, so that no element of the resulting list is a CONS term. Other than NIL, constants $a, b, c,...$, could be in the argument list of the defined function.

$$\textbf{unsat}[z \stackrel{?}{=} \text{CONS}(u, v)] :: \mathit{Flat}(z) \rightarrow z,$$

$$\textbf{unsat}[x \stackrel{?}{=} \text{CONS}(u, v)] :: \mathit{Flat}(\text{CONS}(x, y)) \rightarrow \text{CONS}(x, \mathit{Flat}(y)),$$

$$\textbf{sat}[x \stackrel{?}{=} \text{CONS}(u, v)] :: \mathit{Flat}(\text{CONS}(x, y)) \rightarrow \mathit{Flat}(\text{CONS}(u, \text{CONS}(v, y))).$$

This conditional definition may be translated to the following Applicative definition:

$$[\mathit{Flat}(x)\ \text{COND}([\textbf{not}\ \text{ISCONS}(x) \rightarrow x]$$
$$[\text{ISCONS}(x) \wedge \textbf{not}\ \text{ISCONS}(\text{CAR}(x))$$
$$\rightarrow \text{CONS}(\text{CAR}(x), \mathit{Flat}(\text{CDR}(x)))]$$
$$[\text{ISCONS}(x) \wedge \text{ISCONS}(\text{CAR}(x))$$
$$\rightarrow \mathit{Flat}(\text{CONS}(\text{CAR}(\text{CAR}(x)), \text{CONS}(\text{CDR}(\text{CAR}(x)), \text{CDR}(x))))])].$$

3.5. *Applicative to Conditional Rewrite Rule Formalism*

Transformation of functions from an applicative definition to a conditional definition is straightforward, as illustrated below: each rule corresponds to one of the $[C_k \rightarrow t_k]$ pairs of the conditional expression. The condition in each (kth) rule also contains negations of the conditions in each preceding pair $[C_{k-i} \rightarrow t_{k-i}]$ of the conditional expression. However, such a translation does not in general lead to a Fully Defined CTRS: the well-founded term ordering required for termination of the CTRS cannot be obtained when (as in most Applicative definitions), extractor terms are to be deemed "smaller" than their subterms (e.g., $\text{CAR}(x) \lll x$).

EXAMPLE. The applicative definition $[f(\bar{x})\ \text{COND}([C_1 \rightarrow t_1],..., [C_n \rightarrow t_n])]$ may be translated to the conditional rewrite rules

$$\textbf{sat}[C_1] :: f(\bar{x}) \to t_1$$
$$\textbf{sat}[C_2]\ \textbf{unsat}[C_1] :: f(\bar{x}) \to t_2$$
$$\vdots$$
$$\textbf{sat}[C_n]\ \textbf{unsat}[C_1, C_2, \ldots, C_{n-1}] :: f(\bar{x}) \to t_n.$$

3.6. *Applicative to Unconditional Rewrite Rule Formalism*

The transformation of functions from the applicative definition to an unconditional rewrite rule definition can be done in two phases: first, to the conditional rule formalism, and second from the conditional to the unconditional formalism. Both of these phases have been described in earlier sections (3.5 and 3.3). Since the only variables of an applicative definition are those that occur as "arguments" to the function being defined, both phases of translation are possible. However, considerable control structure is embedded in the resulting TRS, disallowing the nondeterminism normally inherent in term rewriting reductions.

4. Comparison of the Formalisms

When new variables not occurring in the lhs are allowed in the condition of a conditional definition rule, the CTRS formalism is more expressive than the other formalisms, and spans a larger subset of first order logic formulas than the others. We focus the following discussion on the more comparable case, wherein no new variables are allowed in the condition of a rule. We examine how the three formalisms described above differ in the amount of control information embedded in their operational semantics.

It may appear to be possible to incorporate conditional rewriting into unconditional TRS by using a 3-ary *if_then_else* (or a 2-ary *if_then*) function with the rules:

$$\textit{if\_then\_else}(\textbf{true}, x, y) \to x,$$
$$\textit{if\_then\_else}(\textbf{false}, x, y) \to y.$$

However, these rules may lead to nonterminating rewrite sequences since the nondeterminism in the choice of the subterm to be rewritten allows undesirable and unnecessary rewriting.

Example. Consider the TRS with the rule

$$\textit{Fact}(y) \to \textit{if\_then\_else}((y \leqslant 0), 1, y \times \textit{Fact}(y - 1)).$$

When this rule is invoked by attempting to rewrite $Fact(1)$, one of the reduction sequences obtainable is

$$\begin{aligned} Fact\,(1) &\to \mathit{if\_then\_else}((1 \leqslant 0), 1, 1 \times Fact\,(1-1)) \\ &\to \mathit{if\_then\_else}((1 \leqslant 0), 1, 1 \times \mathit{if\_then\_else}(((1-1) \leqslant 0), 1, (1-1) \\ &\quad \times Fact((1-1)-1)) \\ &\to \\ &\quad \vdots \end{aligned}$$

To obtain the desired result, it is necessary to impose the restriction that the *if* part be evaluated first, before rewriting the *then* and *else* parts of the term. Such a restriction is foreign to the range of non-determinism allowed by TRS, and conforms more to the operational semantics of CTRS. Thus, although there exists an unconditional rewrite sequence achieving the same effect as every conditional rewrite sequence, this TRS allows several other rewrite sequences (disallowed by the corresponding CTRS), some of which may be nonterminating.

Using the method proposed in [DePl85] (discussed in Sect. 3.3), translation from conditional rules to unconditional TRS can be carried out whenever constructor terms denote distinct objects. However, this translation also involves considerably restricting the order in which rules are to be applied, into a compound *if_then_else* structure with more control embedded than even that of the corresponding CTRS rules, and hence resembling an applicative definition rather than a TRS. Translation avoiding the deterministic structure causes the problems of either non-termination (as in the *Fact* example above) or reduction to an irreducible term with a new function-symbol.

A greater amount of control information is embedded in an applicative definition than in a CTRS definition: applicative definitions imply a strict sequencing in the evaluation of conditions. As discussed in Section 3.5, any applicative definition may be translated into a conditional definition in which each rule is of the form $\mathbf{sat}[C_i]\mathbf{unsat}[C_1, C_2, \ldots, C_{i-1}] :: f(\bar{x}) \to t_i$. The operational semantics of Tue-Reduction allows any of the subterms C_j (where $j \leqslant i$) to be evaluated first. For some k, the Tue-Reduction of C_k may not terminate, whereas Tue-Reducing a different subterm of the condition may obviate the need to Tue-Reduce C_k. An effective and terminating evaluation may hence be possible only if a control strategy enforces some order of evaluation among conditions.

EXAMPLE. We consider the translation to a CTRS of the following applicative definition of a function f generating the sequence 1, 2, 5, 10, 21,..., for integer arguments 0, 1, 2, 3, 4,..., respectively:

$$[f(z)\ \mathrm{COND}([(z \leqslant 0) \rightarrow 1]$$
$$[odd(f(z-1)) \rightarrow 2{*}f(z-1)]$$
$$[\mathbf{true} \rightarrow 1 + 2{*}f(z-1)])]$$

where *odd*, $\leqslant$, $+$, $-$, $*$ are functions assumed to be defined at a lower level. The corresponding conditional definition rules obtained by translation are:

(1) $\mathbf{sat}[z \leqslant 0] :: f(z) \rightarrow 1$,

(2) $\mathbf{sat}[odd(f(z-1))]\ \mathbf{unsat}[z \leqslant 0] :: f(z) \rightarrow 2{*}f(z-1)$,

(3) $\mathbf{unsat}[(z \leqslant 0),\ odd(f(z-1))] :: f(z) \rightarrow 1 + 2{*}f(z-1)$.

In this conditional definition, the attempt to Tue-Reduce $f(-1)$ will succeed only if rule (1) is used for reduction. Attempting to reduce $f(-1)$ using conditional rule (2) entails checking the condition $\mathbf{sat}[odd(f(-2))]\ \mathbf{unsat}[-1 \leqslant 0]$ which means $f(-2)$ will first have to be evaluated, which in turn means $f(-3)$ has to be evaluated, and so on in a non-terminating sequence.

We thus identify the degree of non-determinism in execution strategy as the factor important in distinguishing between definitions for the same function in the TRS, CTRS and applicative formalisms.

5. Conclusions

We have presented three formalisms of function definition in this paper. Issues of unambiguity, completeness and termination of functions upon invocation on constructor terms have been addressed. Syntactic criteria have been drawn up to enable verification of these properties for function definitions. The formalisms of TRS, CTRS, and Applicative systems have been compared, particularly with respect to interconvertibility of definitions among the formalisms investigated.

We have defined an operational semantics for conditional term rewriting systems that is more general than several other formulations. Conditions in the rules may contain variables not in the lhs, and literals (including equalities) that are to be proved satisfiable or unsatisfiable. Variables absent from the lhs (allowed in the condition and the rhs of a rule) serve as intermediate objects during computation and also allow the testing of syntactic structure of normal forms of terms. However, Tue-reducibility as formulated in this paper is inefficient as well as undecidable, and practical considerations necessitate restrictions on the degree of generality allowed. We have given restrictions that allow the definition of functions in a way that satisfies the desirable properties of unambiguity, completeness and ter-

mination. The multilevel pseudo-hierarchy allows for recursive function invocations in the conditions of a rule.

We find that it is easiest to translate unconditional term rewriting definitions to other formalisms. Correspondingly, translation from conditional term rewriting systems to other systems seems hardest. The expressive power of each system appears to be directly related to the degree of nondeterminism allowed by the formalism.

In summary, we have explored function definition mechanisms with varied expressive power, ease of computation, and degrees of nondeterminism, formulating syntactically verifiable restrictions to ensure provability of termination, unambiguity, and completeness. We feel that this work would assist the tasks of program design and verification in declarative languages.

ACKNOWLEDGMENTS

This research has been supported by National Science Foundation Grant MCS8401624. We are grateful to Jean Luc Remy (CRIN, Nancy) for several valuable suggestions and fruitful discussions. We thank Deepak Kapur (GE, Schenectady), Jieh Hsiang (SUNY, Stony Brook) and the referees for many useful comments.

RECEIVED January 9, 1985; ACCEPTED December 2, 1985

REFERENCES

[BaRe83] BARROS, A. L., AND REMY, J. L. (1983), Ecologiste: A system to make complete and consistent specifications easier, *in* "Proc. Workshop on Rewrite Rule Lab., Res. Rep.," GE Corporate R&D, Schenectady, New York.

[BeKl82] BERGSTRA, J. A., AND KLOP, J. W. (1982), "Conditional Rewrite Rules: Confluency and Termination," Research Report IW 198/82, Mathematical Centre, Amsterdam.

[BrDJ78] BRAND, D., DARRINGER, J. A., AND JOYNER, W.H. (1978), "Completeness of Conditional Reductions," IBM Research Report, No. RC7404.

[DePl85] DERSHOWITZ, N., AND PLAISTED, D. (1985), Logic programming cum applicative programming, *in* "Proc. 1985 Symp. on Logic Programming," Boston.

[Ders79] DERSHOWITZ, N. (1979), Orderings for term rewriting systems, *in* "Proc. of 20th Symp. on Foundations of Computer Science," pp. 123–131.

[Ders83] DERSHOWITZ, N. (1983), "Applications of the Knuth–Bendix Procedure," Report No. ATR–83(8478)-2, Aerospace Corp., El Segundo, California, May.

[Ders84] DERSHOWITZ, N. (1984), Equations as programming language, *in* "IEEE 1984 Proc. 14th Jerusalem Conf. Info. Tech. (JCIT)."

[Ders85] DERSHOWITZ, N. (1985), Termination of rewriting, *in* "Proc. First Intl. Conf. on Rewriting Techniques and Applications, Dijon, France."

[Fay79] FAY, M. (1979), Fist order unification in an equational theory, *in* "Proc. 4th Workshop on Automated Deduction, Texas."

[GuHo78] GUTTAG, J. V., AND HORNING, J. J. (1978), The algebraic specification of abstract data types, *Acta Inform.* **10**, 27–52.

[GuKM82] GUTTAG, J. V., KAPUR, D., AND MUSSER, D. R. (1982), On proving uniform termination and restricted termination of rewriting systems, *in* "Proc. 9th ICALP, Aarhus, Denmark.

[Hend80] HENDERSON, P. (1980), "Functional Programming," Prentice–Hall, London.

[HoOd82] HOFFMAN, C. M., AND O'DONNELL, M. J. (1982), "Programming with Equations," ACM TOPLAS, Vol. 4, No. 1, pp. 83–112, January.

[Hsia83] HSIANG, J. (1983), "Topics in Automated Theorem Proving and Program Generation," Ph.D. thesis, Univ. of Illinois, Urbana–Champaign.

[Huet80] HUET, G. (1980), Confluent reductions: Abstract properties and applications to term rewriting systems, *J. Assoc. Comput. Mach.* **27**, 797–821.

[HuHu80] HUET, G., AND HULLOT, J. M. (1980), Proofs by induction in equational theories with constructors, *in* "Proc. 21st IEEE Symp. on Foundations of Computer Science," pp. 96–107.

[HuLa78] HUET, G., AND LANKFORD, D. S. (1978), "On the Uniform Halting Problem for Term Rewriting Systems," Report 283, INRIA, Le Chesnay, France.

[Hull80] HULLOT, J. M. (1980), "Canonical Forms and Unification," Tech. Rep. CSL-114, SRI Intl., Calif., April.

[HuOp80] HUET, G., AND OPPEN, D. S. (1980), "Equations and Rewrite Rules: A Survey," Tech. Rep. CSL-111, SRI Int'l, Calif., January.

[Huss85] HUSSMAN, H. (1985), Unification in conditional-equational theories, Vol. 2, *in* "Proc. EUROCAL '85, Linz, Austria," Lecture Notes in Computer Science, Vol. 204, Berlin.

[JoKo85] JOUANNAUD, J. P., AND KOUNALIS, E. (1985), "Proofs by Induction in Equational Theories without Constructors," CRIN, Nancy, France.

[JoLR82] JOUANNAUD, J. P., LESCANNE, P., AND REINIG, F. (1982), Recursive decomposition ordering, *in* "Formal Description of Programming Concepts 2" (D. Borjner, Ed.), North-Holland, Amsterdam.

[KaMu82] KAPUR, D. K., AND MUSSER, D. R. (1982), Rewrite rule theory and abstract data type analysis, *in* "Computer Algebra, EUROSAM 1982," Lecture Notes in Computer Science, Vol. 144, pp. 263–297, (Calmet, Ed.), Springer–Verlag, Berlin, April.

[Kapl83] KAPLAN, S. (1983), "Conditional Rewrite Rules," Rapport de Recherche, No. 150, Universite de Paris-Sud, Orsay, France, December.

[Kapl84] KAPLAN, S. (1984), "Fair Conditional Term Rewriting Systems: Unification, Termination and Confluence," Rapport de Recherche, No. 194, Universite de Paris-Sud, Orsay, France, November.

[KaSr85] KAPUR, D., AND SRIVAS, M. K. (1985), A rewrite rule based approach for synthesizing abstract data types, *in* "Proc. 1985 ICALP Conf.," April.

[Kirc84] KIRCHNER, H. (1984), A general inductive completion algorithm and application to abstract data types, *in* "Proc. 7th CADE," Lecture Notes in Computer Science, Vol. 170, pp. 282–302, Springer–Verlag, Berlin/Heidelberg/New York/Tokyo.

[KnBe70] KNUTH, D., AND BENDIX, P. (1970), Simple word problems in universal algebras, *in* "Computational Problems in Abstract Algebra," (J. Leech, Ed.), pp. 263–297, Pergamon, Elmsford, N. Y.

[Koun85] KOUNALIS, E. (1985), Completeness in date type specification, *in* "Proc. EUROCAL Conf. at Linz, Austria," April.

[Lank79] LANKFORD, D. S. (1979), "Some New Approaches to the Theory and Applications of Conditional Term Rewriting Systems," Research Report MTP-6, Univ. of Louisiana, Ruston.

[Lesc84] LESCANNE, P. (1984), Uniform termination of term rewriting systems—Recursive decomposition ordering with status, *in* "Proc. 9th CAAP Conf., Bordeaux, France,"

[McCa60] MCCARTHY, J. (1960), Recursive functions of symbolic expressions and their computation by machine, Part I, *Comm. ACM* **3** (4), April.

[Muss80] MUSSER, D. R. (1980), On proving inductive properties of abstract data types, *in* "Proc. 7th POPL Conf., Las Vegas,"

[Pada83] PADAWITZ, P. (1983), "Correctness, Completeness, Consistency of Equational Data Type Specifications," TU Berlin Bericht, No. 83-15.

[Plai78a] PLAISTED, D. A. (1978), "Well-Founded Orderings for Proving Termination of Systems of Rewrite Rules," Report R–78–932, Univ. of Illinois, Urbana–Champaign.

[Plai78b] PLAISTED, D. A. (1978), "A Recursively Defined Ordering for Proving Termination of Term Rewriting Systems," Report No. R–78–943, Univ. of Illinois, Urbana–Champaign.

[PlEE82] PLETAT, U., ENGELS, G., AND EHRICH, H. D. (1982), "Operational Semantics of algebraic specifications with Conditional Equations," 7eme C.A.A.P., Lille, France.

[Remy82] REMY, J. L. (1982), "Etude des systemes de reecriture conditionnels et application aux specifications algebriques de types abstraits," Doctoral thesis, INPL, Nancy, France.

[Remy83] REMY, J. L. (1983), Proving conditional identities by equational case reasoning, rewriting and normalization, *in* "Proc. of 1982–83 Research Seminar, LITP, Paris, France; Theorique de Paris, Paris, 1983, Aussi: Rapport CRIN 82–R–085, Nancy, France, 1982.

[ReZh84] REMY, J. L., AND ZHANG, H. (1984), REVE4: A system for validating conditional algebraic specifications of paremeterized abstract data types, *in* "Proc. of 6th ECAI Conf., Pisa, Italy,"

[ReZh85] REMY, J. L., AND ZHANG, H. (1985), REVEUR4: A system to proceed Experiments on Conditional Term Rewriting Systems," CRIN, Nancy, France.

[Sriv82] SRIVAS, M. K. (1982), "Automatic Synthesis of Implementations for Abstract Date Types from Algebraic Specifications," MIT/LCS/TR-276, M.I.T., Boston, June.

[Thie83] THIEL, J. J. (1983), Stop losing sleep over incomplete data type specifications, *in* "Proc. 11th POPL Conf.," Assoc. Comput. Mach., New York.

[Wald85] WALDMANN, B. (1985), "Reducing Conditional Term Rewriting Systems," Research Report, CRIN, Nancy, France.