Theoretical Computer Science 75 (1990) 111-138
North-Holland

A RATIONALE FOR CONDITIONAL EQUATIONAL PROGRAMMING*

Nachum DERSHOWITZ
Department of Computer Science, University of Illinois, Urbana, IL 61801, U.S.A.

Mitsuhiro OKADA
Department of Computer Science, Concordia University, Montreal, Quebec H3G 1M8, Canada

Abstract. Conditional equations provide a paradigm of computation that combines the clean syntax and semantics of LISP-like functional programming with Prolog-like logic programming in a uniform manner. For functional programming, equations are used as rules for left-to-right *rewriting*; for logic programming, the same rules are used for *conditional narrowing*. Together, rewriting and narrowing provide increased expressive power. We discuss some aspects of the theory of conditional rewriting, and the reasons underlying certain choices in designing a language based on them. The most important correctness property a conditional rewriting program may possess is *ground confluence*; this ensures that at most one value can be computed from any given (variable-free) input term. We give criteria for confluence. Reasonable conditions for ensuring the completeness of narrowing as an operational mechanism for solving goals are provided; these results are then extended to handle rewriting with existentially quantified conditions and built-in predicates. Some termination issues are also considered, including the case of rewriting with higher-order terms.

1. Introduction

In recent years, various suggestions for combining features of functional programming and logic (relational) programming have been made; see the collection in [7] and the survey in [2]. The simplest provide a convenient interface between resolution-based goal reduction and rewrite-based term evaluation, by using rewrite rules to normalize terms (i.e. rewriting them to terms that cannot be rewritten further) before unification is attempted. For example, the *append* function can be defined as usual:

$$append(x, y) = \text{if } null(x) \text{ then } y \text{ else } cons(car(x), append(cdr(x), y))$$

* A previous version of this paper appeared as: Conditional equational programming and the theory of conditional term rewriting, in: *Proc. Internat. Conf. on Fifth Generation Computer Systems*, Tokyo, Japan (1988) 337-346.

This research was supported in part by the National Science Foundation (United States) under Grant DCR 85-13417, the Natural Science and Engineering Research Council (Canada) under Grants OGP36663, ICR92599, Fonds pour la Formation de Chercheurs et l'Aide à la Recherche (Quebec) under Grants EQ41840, NC0025, Social Sciences and Humanities Research Council (Canada) under Grant 410-88-884, and the Committee on Aid to Research Activities (Concordia Univ.) under Grant N42. Part of this work was done when the second author was visiting the Laboratoire de Recherche en Informatique (L.R.I.), Université de Paris-Sud, with support from Ministère de l'Enseignment National (France).

0304-3975/90/$03.50 © 1990—Elsevier Science Publishers B.V. (North-Holland)

and Quicksort can be written with Horn clauses as follows:

$$eq(x, x)$$

$$sort(nil, nil)$$

$$partition(y, x, y_1, y_2) \wedge sort(y_1, z_1) \wedge sort(y_2, z_2)$$
$$\wedge\, eq(z, append(z_1, cons(x, z_2))) \supset sort(cons(x, y), z).$$

(with a suitable program for *partition*). Such languages do not use function definitions to instantiate free variables during goal reduction, and, consequently, are *incomplete*, in the sense that a solution to a goal will not necessarily be found whenever one provably exists. For example, the above program would find no solutions to the goal

$$sort(cons(2, cons(1, nil)), append(x, y)).$$

An early example of such a language is QLOG [34].

An alternative approach is to treat function definitions, like *append*, as a pair of implications

$$null(x) \supset append(x, y) = y$$

$$\neg null(x) \supset append(x, y) = cons(car(x), append(cdr(x), y)).$$

The necessary properties of *car*, *cdr*, and *null* could also be given as clauses:

$$null(nil)$$

$$\neg null(cons(x, y))$$

$$car(cons(x, y)) = x$$

$$cdr(cons(x, y)) = y.$$

Paramodulation (unifying one side of an equation with a nonvariable subterm of a clause and replacing with the other side) would then be used (with resolution) to solve goals. Uniform [30] is an early extension of Prolog incorporating such an equality rule. Though completeness is achievable in such a manner, the resultant language requires non-linear forward reasoning and lacks the sense of direction that distinguishes computing from theorem proving.

Another alternative is to use function definitions as one-way rewriting rules:

$$append(x, y) \to \text{if } null(x) \text{ then } y \text{ else } cons(car(x), append(cdr(x), y)).$$

Rewrite rules are used to replace equals-by-equals, but only in the left-to-right direction. That is, a rule $l \to r$ may be applied to a term t if a subterm s of t matches (by "one-sided" unification) the left-hand side l with some substitution σ of terms for the variables in l. The rule is applied by replacing the subterm $s\sigma = l\sigma$ in t with the right-hand side $r\sigma$. Two additional rules for simplification are always needed:

$$\text{if } true \text{ then } r \text{ else } s \to r$$

$$\text{if } false \text{ then } r \text{ else } s \to s.$$

In the above example, rules for *car*, *cdr*, and *null* are also needed. Terms are rewritten until no rule applies; when (and if) that situation occurs, the resultant irreducible term, called a *normal form*, is considered the *value* of the initial term.

Often, the two cases defined by a condition can be better expressed as mutually exclusive left-hand-side patterns. For example, the following set of rules suffices for *append*:

$$append(nil, y) \rightarrow y$$

$$append(cons(x, y), z) \rightarrow cons(x, append(y, z))$$

$$null(nil) \rightarrow true$$

$$null(cons(x, y)) \rightarrow false$$

$$car(cons(x, y)) \rightarrow x$$

$$cdr(cons(x, y)) \rightarrow y.$$

To use rewrite rules for logic programming, i.e. to find values for variables that satisfy an equational goal like $append(x, y) = x$, a "linear" restriction of paramodulation, analogous to the SLD-strategy for Horn-clause logic, can be used. *Narrowing* [52] is like rewriting, except that unification is used in place of pattern matching: a rule $l \rightarrow r$ may be applied to a term t if a *nonvariable* subterm s of t unifies with the left-hand side l with some substitution of terms for the variables in l. (Variables in l and t are treated as disjoint.) The result is $t\sigma$ with $s\sigma$ replaced by $r\sigma$, where σ is the most general unifier of l and s. A programming language with narrowing-like operational semantics was first suggested in [9]. Other languages using narrowing or related mechanisms as a way of incorporating functions with goal reduction include: SEC [18]; EQLOG [21]; TABLOG [36]; QUTE [51]; FGL+LV [35]; and EQL [26]. Like paramodulation, narrowing can be simulated in Prolog by decomposing terms [8, 55].

For completeness of narrowing, *ground confluence* of the system of oriented equations is required. Ground confluence implies that a variable-free term can have at most one normal form. With ground confluence, any irreducible solution to a goal can be found by narrowing. *Orthogonal* (or *regular*) rewriting systems obey the following syntactic conditions:

(a) no variable appears more than once on any left-hand side,

(b) only variables that appear on the left appear on the right,

(c) no left-hand side is unifiable with a nonvariable (not necessarily proper) subterm of another left-hand side, and

(d) no left-hand side unifies with (a renamed) nonvariable proper subterm of itself.

Orthogonal systems are always confluent [40].

The above system for *append*, for instance, meets all these requirements. Semantic methods for establishing ground confluence are given in [47, 56].

The main problem with using conditional terms is that the resultant rules are usually nonterminating. In other words, uninhibited rewriting need not halt with

an irreducible term. For example, with the "if-then-else" definition of *append*, an infinite sequence of rewrites is possible:

$$\begin{aligned} append(nil, nil) &\rightarrow \text{if } null(nil) \text{ then } nil \text{ else } cons(car(nil), append(cdr(nil), nil)) \\ &\rightarrow \text{if } true \text{ then } nil \text{ else } cons(car(nil), append(cdr(nil), nil)) \\ &\rightarrow \text{if } true \text{ then } nil \text{ else } cons(car(nil), \text{if } null(cdr(nil)) \text{ then } nil \\ &\qquad \text{else } cons(car(cdr(nil)), append(cdr(cdr(nil)), nil))) \\ &\rightarrow \cdots. \end{aligned}$$

Thus, to guarantee that normal forms of terms and irreducible solutions to goals will be found whenever they exist, requires a lazy, outermost evaluation strategy, in which conditions are evaluated first and the "if-then-else" is simplified. See, for example, [49]. Thus, eager evaluation of arguments (as in LISP, is not permissible. Furthermore, rather strict syntactic conditions (i.e. orthogonality) are necessary for completeness.

These considerations suggest the use of *conditional* rewrite rules as a means of expressing function definitions. Each definition

$$f(\bar{x}) = \text{if } p[\bar{x}] \text{ then } r[\bar{x}] \text{ else } s[\bar{x}]$$

(square brackets are used to indicate that the variables $\bar{x}$ may appear anywhere in the indicated term) translates into two conditional rules:

$$p[\bar{x}] = true \mid f(\bar{x}) \rightarrow r[\bar{x}]$$

$$p[\bar{x}] = false \mid f(\bar{x}) \rightarrow s[\bar{x}].$$

Using conditional equations allows one to program with rules that can never lead to infinite sequences of rewrites. Such systems are called *terminating*. Proposed languages along these lines include RITE [14, 27], SLOG [19], and EQLOG [22].

In this paper, we concentrate on narrowing-based programming languages that require the programmer to use only terminating rules. Termination does not limit expressibility, since any potentially nonterminating function definition (such as the definition of an interpreter) can be rephrased to include a bound, as illustrated in [14]. Instead of a definition

$$f(\bar{x}) = t[f(\bar{s})],$$

a definition with one extra argument,

$$f'(\bar{x}, succ(n)) = t[f'(\bar{s}, n)],$$

is used, and instead of computing $f(\bar{v})$, the goal $f'(\bar{v}, n) = z$ can be solved.

Section 2 is on completeness of conditional rewriting and Section 3 is on decidability and uniqueness of normal forms. Section 4 addresses the completeness of conditional narrowing, and Section 5 considers what happens when conditions contain existentially quantified variables. Section 6 extends previous results to the

case in which conditions involve built-in predicates. Then, Section 7 considers rewriting with higher-order lambda terms and existential variables. We conclude with a brief discussion.

2. Conditional rewriting

In functional programming, one is usually interested in finding a "normal form" t that is "equal" to a given (variable-free) input term s. To be a normal form, t must satisfy some criterion, usually that no "rule" or "function definition" applies to it. Equality of s and t must be provable in some logical system. In this section, we explore the adequacy of rewriting as a means of computing normal forms. A preliminary version of this section appeared in [43].

We use standard notations [23]: $s = t$ stands for the usual sense of equality in logical systems; $s \rightarrow t$ stands for one rewrite step in a given rewriting system; $\leftarrow$ is the inverse of the rewrite relation $\rightarrow$; $\leftrightarrow$ is the symmetric closure of $\rightarrow$; $\rightarrow^+$ is its transitive closure; $\rightarrow^*$ is its reflexive-transitive closure; $s \leftrightarrow^* t$ is its symmetric-reflexive-transitive closure; $^*\!\leftarrow$ is the reflexive-transitive closure of $\leftarrow$; and $s\downarrow t$ means $s \rightarrow^* u \;^*\!\leftarrow t$, for some u. A term s is in *normal form* if there is no t such that $s \rightarrow t$; we write $s \rightarrow^! t$ if $s \rightarrow^* t$ for normal form t. We will assume some familiarity with the main notions in rewriting, viz. termination, confluence, and critical pair; see [24] or [10].

By a *conditional equational system*, we mean a set of Horn clauses of the form

$$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \quad \supset \quad l = r.$$

A *natural conditional rewriting system* R has rules of the form

$$s_1 \overset{*}{\leftrightarrow} t_1 \wedge \cdots \wedge s_n \overset{*}{\leftrightarrow} t_n \mid l \rightarrow r.$$

When such a rule applies, an instance $l\sigma$ of l in a term s is replaced by $r\sigma$, yielding a term t. We write $R \vdash s \rightarrow t$, or just $s \rightarrow t$. The rule applies, however, only if there exists a proof $s_i \leftrightarrow^* t_i$ for each of the instantiated conditions, where these proofs may use any number of rewrites in *either* direction. If $n = 0$, the rule is *unconditional.*

Note that a natural rewriting system is not very different from the underlying equational system. Every rule (of the above form) corresponds to the conditional equation (shown above), obtained by replacing $\rightarrow$ with $=$, $\leftrightarrow^*$ with $=$, and $\mid$ with $\supset$. Let R^{eqn} denote the *underlying* equational system obtained from a natural system R^{nat} in this way.

Theorem 2.1. *For any natural conditional rewriting system R^{nat} and underlying conditional equational system R^{eqn},*

$$R^{\text{nat}} \vdash p \overset{*}{\leftrightarrow} q \;\; \textit{iff} \;\; R^{\text{eqn}} \vdash p = q$$

for all terms p and q.

Proof. If $p = q$ is provable in the logical system R^{eqn}, there is a proof in SPU (*Selected Positive Unit*) form, i.e. a proof in which each condition $s_i\sigma = t_i\sigma$ is proved before a substitution instance

$$s_1\sigma = t_1\sigma \wedge \cdots \wedge s_n\sigma = t_n\sigma \supset l\sigma \rightarrow r\sigma$$

of a conditional rule is used to replace equals. Thus, R^{nat} can prove $p \leftrightarrow^* q$ by imitating the proof in R^{eqn}, step by step. In fact, the rewriting proof has exactly the same proof structure as the SPU one, where a right-to-left use of a rule in the proof of $p \leftrightarrow^* q$ in R^{nat} corresponds to use of the symmetric axiom in the proof of $p = q$ in R^{eqn}. The SPU-proof strategy is complete with respect to first-order equational validity and provability, from which it follows that conditional replacement of equals for natural rewriting systems also is complete. Conversely, any proof in R^{nat} can readily be interpreted as a proof in R^{eqn}. □

For this reason we can identify (SPU-)proofs in an equational system with the corresponding proofs in the corresponding natural system. A proof $p \leftrightarrow^* q$ in a natural rewriting system is therefore called an *equational proof.*

The question is, under what conditions can one be assured that any normal form provably equal to a given input term can be found by rewriting? In other words, when is it the case that $s \leftrightarrow^* t$, for an unrewritable term t, implies $s \rightarrow^* t$?

From classical rewriting theory, we know that the *confluence* property ($p \downarrow q$ whenever $p \; {}^*\!\!\leftarrow u \rightarrow^* q$ for some u), is equivalent to the *Church-Rosser* property (any equational proof $p \leftrightarrow^* q$ can be replaced by a rewrite proof $p \downarrow q$). Thus, for any confluent system and normal form t, $s \leftrightarrow^* t$ implies $s \rightarrow^* t$, since $t \rightarrow^* v$ implies $t = v$. So, rewriting with confluent systems can be used to find normal forms.

One problem with natural systems is that the conditions for applying a rule involve arbitrary proofs of equality, so we have gained little from the notion of rewriting. To remedy this defect, we consider a more restrictive definition of conditional rewriting: A *standard (join) conditional rewriting system* is a set of rules of the form

$$s_1 \downarrow t_1 \wedge \cdots \wedge s_n \downarrow t_n \mid l \rightarrow r,$$

meaning that an instance $l\sigma$ of l rewrites to $r\sigma$ only if each $s_i\sigma$ can be reduced (by zero or more rewrites) to the same term as the corresponding $t_i\sigma$. For example, with the standard system

$$null(nil) \rightarrow true$$

$$null(cons(x, y)) \rightarrow false$$

$$car(cons(x, y)) \rightarrow x$$

$$cdr(cons(x, y)) \rightarrow y$$

$$null(x) \downarrow true \mid append(x, y) \rightarrow y$$

$$null(x) \downarrow false \mid append(x, y) \rightarrow cons(car(x), append(cdr(x), y))$$

we have $append(cons(a, cons(b, nil)), cons(c, nil)) \rightarrow^* cons(a, cons(b, cons(c, nil)))$.

For a standard system R^{std}, let R^{eqn} denote the underlying equational system and R^{nat} denote the corresponding natural system (replacing each $\downarrow$ with $\leftrightarrow^*$). An inductive argument provides the following.

Theorem 2.2. *For any confluent standard conditional rewriting system* R^{std},

$$R^{\text{std}} \vdash p\downarrow q \ \textit{ iff } \ R^{\text{eqn}} \vdash p = q.$$

It is easy to see that if R^{std} is confluent, then so is the natural system R^{nat}, but the converse does not hold, in general, since the enabling conditions in a natural proof need not have proofs that are transformable into downarrow ones.

A (standard or natural) conditional system is *decreasing* (cf. [13, 28, 32]) if there exists a well-founded extension $>$ of the rewrite relation $\rightarrow$ which satisfies two additional properties:

(a) $>$ contains the proper subterm relation $\rhd$ (i.e. if s is a proper subterm of t then $t > s$) and

(b) for each rule $s_1\downarrow t_1 \wedge \cdots \wedge s_n\downarrow t_n \mid l \rightarrow r$, $l\sigma > s_i\sigma$, $t_i\sigma$ for all substitutions σ and indices i $(1 \leq i \leq n)$.

We will say that a proof $s \leftrightarrow^* t$ is *fully normal* if it is a normal proof $s\downarrow t$, and there are fully normal subproofs of the conditions $s_i\sigma \leftrightarrow^* t_i\sigma$ used in the "surface" proof $s\downarrow t$. With these notions, we have the following.

Theorem 2.3. *If a natural conditional rewriting system* R^{nat} *is decreasing and confluent, then*

$$R^{\text{nat}} \vdash p\downarrow q \ \textit{ iff } \ R^{\text{std}} \vdash p\downarrow q.$$

Proof. Any proof in the standard system is also a proof in the natural system, so one direction is trivial. For the other direction, we show that any proof in the natural system can be transformed into a fully normal proof, which holds in the standard system.

The proof proceeds by replacing each proof level with a rewrite ($\downarrow$) proof. More precisely, first we normalize a surface proof $s \leftrightarrow^* t$ to a normal form $s\downarrow t$ in the given natural system. This is possible by the confluence of the natural system. Then we consider the immediate conditions $c_1 \leftrightarrow^* d_1, \ldots, c_n \leftrightarrow^* d_n$ used for the proof $s\downarrow t$, and normalize the surface proof of each of these to $c_i\downarrow d_i$, exactly as in proof simplification for unconditional systems [1]. This normalization process is repeated until all proofs and subproofs are rewrite proofs.

To see that the successive subproof normalizations stop after a finite number of steps, one can use multiset induction [10a], with elements compared in the well-founded ordering $>$ showing decreasingness. Normalizing the surface proof of $s = t$ above has complexity $\{s, t\}$, while normalization of the conditions $c_i =^* d_i$ have complexity $\{c_i, d_i\}$, which is smaller on account of the decreasingness condition.

Since the multiset ordering is well-founded, normalization must terminate with a fully normal proof of $s = t$. □

One can weaken the condition of rule applicability even further. A *normal conditional rewrite system* is a rewrite system whose rules have the following form:

$$s_1 \rightarrow^! t_1 \wedge \cdots \wedge s_n \rightarrow^! t_n \mid l \rightarrow r,$$

meaning, for each condition, that t_i is a normal form derivable from s_i.

The following two theorems show that normal systems are not too restrictive. The first states that any standard system can be simulated by a normal system. For any given standard system R^{std}, let R^{ext} be the normal system obtained by replacing rules of the form

$$s_1 \downarrow t_1 \wedge \cdots \wedge s_n \downarrow t_n \mid l \rightarrow r$$

with rules of the form

$$eq(s_1, t_1) \rightarrow^! true \wedge \cdots \wedge eq(s_n, t_n) \rightarrow^! true \mid l \rightarrow r$$

where $eq(\cdot, \cdot)$ is a *new* binary function symbol and *true* is a *new* constant (0-ary function) symbol. Additionally, R^{ext} contains the rule

$$eq(x, x) \rightarrow true.$$

We have the following.

Theorem 2.4 (Simulation). *For any standard rewrite system R^{std} and terms s, t,*

$$R^{\text{std}} \vdash s \downarrow t \text{ implies } R^{\text{ext}} \vdash s \downarrow t.$$

The proof of this theorem (and others to follow) is by induction on the "depth" of a proof, by which we mean the maximum depth of recursion in the evaluation of conditions. More precisely, *depth* is defined as follows:

(1) the depth of a proof of $s \rightarrow t$ is 0 if $s \rightarrow t$ is the result of an application of an unconditional rule;

(2) the depth of a proof of $s \rightarrow t$ is one more than the maximum depth of subproofs for conditions $u_1 \downarrow v_1, \ldots, u_n \downarrow v_n$ if $s \rightarrow t$ is the result of an application of a substitution instance of the form $u_1 \downarrow v_1 \wedge \cdots \wedge u_n \downarrow v_n \mid l \rightarrow r$ of a rule;

(3) the depth of a proof $s \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_m \rightarrow v \leftarrow t_n \leftarrow t_1 \leftarrow t$ is the maximum depth of subproofs for $s \rightarrow s_1$, $s_1 \rightarrow s_2, \ldots, s_m \rightarrow v$, $t_n \rightarrow v, \ldots, t_1 \rightarrow t_2$, $t \rightarrow t_1$.

Proof. Consider the depth of a proof P of $s \downarrow t$ in R^{std}. If the depth is zero, that means $s \downarrow t$ is provable in R^{std} without any use of conditions. Then, obviously, the same proof P is a proof of $s \downarrow t$ in R^{ext}.

Assume, then, that P has depth $m+1$. By the induction hypothesis each subproof of P whose depth is at most m satisfies the property of the theorem. Suppose that P uses a substitution instance of a rule of the form

$$s_1\downarrow t_1 \wedge \cdots \wedge s_n\downarrow t_n \mid l \rightarrow r$$

to replace l by r, and that

$$P_1, \ldots, P_n$$

are subproofs in R^{std} for $s_1\downarrow t_1, \ldots, s_n\downarrow t_n$, respectively. By the induction hypothesis, there are proofs

$$P'_1, \ldots, P'_n$$

in R^{ext} for $s_1\downarrow t_1, \ldots, s_n\downarrow t_n$, respectively. Hence, there are proofs

$$P''_1, \ldots, P''_n$$

in R^{ext} for $eq(s_1, t_1) \rightarrow^! true, \ldots, eq(s_n, t_n) \rightarrow^! true$, respectively. It follows that there is a substitution instance of the corresponding rule (in R^{ext}) of the form,

$$eq(s_1, t_1) \rightarrow^! true \wedge \cdots \wedge eq(s_n, t_n) \rightarrow^! true \mid l \rightarrow r,$$

and in R^{ext} the same rewrite $l \rightarrow r$ may be used. Hence, there is a proof P^{ext} of $s\downarrow t$ in R^{ext}. □

In general, for two deduction systems R_1 and R_2 in languages (i.e. signatures) L_1 and L_2, respectively, where L_1 is a subset of L_2, R_2 is called a *conservative extension* of R_1, if for every formula ϕ which is expressed in L_1,

$$R_2 \vdash \phi \text{ implies } R_1 \vdash \phi.$$

We have the following converse of the previous theorem.

Theorem 2.5 (Conservation). *For any standard rewrite system R^{std} and terms s and t (having no occurrence of "eq" and of "true"), if $R^{ext} \vdash s\downarrow t$ then $R^{std} \vdash s\downarrow t$, i.e. R^{ext} is a conservative extension of R^{std}.*

Together, the two theorems tell us that for all s and t in the language of R^{std}

$$R^{ext} \vdash s\downarrow t \text{ iff } R^{std} \vdash s\downarrow t.$$

Without loss of generality the language of R^{std} may be assumed not to include *eq* or *true*. If it does, then, one could choose alternative symbols for R^{ext}.

Proof. The proof is by induction on the depth of a given proof P of $s\downarrow t$ in R^{ext}. If the depth is 0, then the same proof holds in R^{std}, because the additional rule $eq(x, x) \rightarrow true$ is not used.

The inductive step is similar to that in the previous proof, and uses the fact that if $eq(s, t) \rightarrow^! true$ is provable in R^{ext} with depth l, then $s\downarrow t$ is also provable in R^{ext} with the same depth l (for s, t not containing eq or $true$. It follows that a subproof P_i' of P with depth less than l for a condition $eq(s_i, t_i) \rightarrow^! true$, admits a proof P_i'' for $s_i\downarrow t_i$ in R^{ext} with the same depth. By the induction hypothesis, we have $R^{std} \vdash s_i\downarrow t_i$; hence, the same rewrite step (of P) can be used for $R^{std} \vdash s\downarrow t$. □

These two theorems state that the expressive power of the class of normal conditional systems is the same as that of the class of standard conditional systems. But, as we will see in the next section, some confluence results for normal systems also require *left-linearity*, i.e. each variable occurring in a left-hand side l occurs only once in l. In general, however, one cannot re-express a left-linear non-normal system as a left-linear normal system, since we need the non-left-linear rule $eq(x, x) \rightarrow true$.

For a comparison of different formulations of conditional rewriting, see [13].

3. Conditional convergence

A convergent (conditional or unconditional) rewriting system is one with both confluence and termination properties. Convergent systems give unique normal forms for any given term. *Ground confluence* means that $s\downarrow t$ for some v whenever $s \,{}^*\!\!\leftarrow u \rightarrow^* t$ for a variable-free term u. A *ground convergent* system is one that is both terminating and ground confluent. Ground convergent rewriting may be used as the evaluation mechanism for first-order functional programs and lends itself easily to parallel evaluation schemes.

For terminating unconditional systems, the Critical Pair Lemma [33] provides an effective test for confluence. Also, for such systems (assuming a finite number of rules), the joinability ($\downarrow$) relation is decidable. Thus, validity is decidable for convergent unconditional systems. Unfortunately, with conditional systems, we are faced with two new phenomena:

(a) joinability is not necessarily decidable, even for finite terminating conditional systems [32];

(b) contrary to what had been surmised [32], the critical pair test does not guarantee confluence for terminating standard systems [12].

To overcome these difficulties, additional constraints on conditional systems are required.

Theorem 3.1. *For decreasing standard systems, the basic notions are decidable, i.e. the rewrite relation* ($\rightarrow$), *derivability relation* ($\rightarrow^*$), *joinability relation* ($\downarrow$), *and normal form attribute are all recursive.*

This can be proved by simultaneous induction with respect to the ordering $>$ that makes it decreasing. For example, to prove that the relation $s \rightarrow t$ is decidable,

we use the decidability of the related conditions of the form $u\downarrow v$, where $u, v < s$ by the decreasingness constraint.

One can readily confirm that decreasing systems are strictly more general than "simplifying systems" [32] or "reductive systems" [28] but enjoy the same nice properties (see [13]). In fact, decreasing systems exactly capture the finiteness of recursive evaluation of terms, in the following technical sense. For given conditional system R, let $\rightsquigarrow$ be the relation defined by $s \rightsquigarrow p$ if there are a rule $s_1\downarrow t_1 \wedge \cdots \wedge s_n\downarrow t_n | l \rightarrow r$ in R and substitution σ such that $l\sigma$ is a subterm of s and p is one of the $s_i\sigma$ or $t_i\sigma$. The relation $\rightarrow \cup \rightsquigarrow$ corresponds to one step of computation, and its transitive closure $(\rightarrow \cup \rightsquigarrow)^+$ represents an arbitrary computation branch.

Theorem 3.2. *For any decreasing natural conditional rewriting system R, the relation $(\rightarrow \cup \rightsquigarrow)^+$ is well-founded if and only if R is decreasing.*

Proof. The "if" direction follows directly from the definition of "decreasing". For the "only if" direction, we can show that if there are no infinite computations (i.e. if no infinite sequences of $\rightarrow$ and $\rightsquigarrow$ steps are possible), then the ordering $(\rightarrow \cup \rightsquigarrow \cup \rhd)^+$ satisfies the conditions for "decreasingness", where $\rhd$ is the subterm ordering. To see that the latter ordering is well-founded, note that were there an infinite chain of $\rightarrow$, $\rightsquigarrow$, and $\rhd$, then there would also be an infinite chain of just $\rightarrow$ and $\rightsquigarrow$, since $s \rightarrow t \rhd v$ for some t whenever $s \rhd u \rightarrow v$ for some u, and $s \rightsquigarrow v$ whenever $s \rhd u \rightsquigarrow v$ for some u. □

It follows from results in the previous section, that in order for a standard rewriting system to be complete with respect to provability in the underlying equational system (or, equivalently, with respect to validity in the sense of first-order logic with identity), we need either to directly establish its confluence, or else to show that the corresponding natural system is decreasing and confluent. We turn now to consider conditions under which a terminating conditional system is confluent whenever its critical pairs are joinable.

If $c | l \rightarrow r$ and $p | s \rightarrow t$ are rules in a conditional system R and l unifies via most general unifier μ with a nonvariable subterm of s, then the conditional equation $c\mu \wedge p\mu \supset s\mu[r\mu] = t\mu$ is a *critical pair* of R, where $s\mu[r\mu]$ is $s\mu$ with its subterm $l\mu$ replaced by $r\mu$. It can be verified that the critical pair test does hold for terminating natural systems, i.e. if $s\sigma\downarrow t\sigma$ for every critical pair $c \wedge p \supset s = t$ and substitution σ such that $c\mu\sigma$ and $p\mu\sigma$ hold, then the system is confluent.

Theorem 3.3 (Dershowitz and Plaisted [15]). *A terminating natural conditional rewriting system is confluent if every critical pair is joinable.*

But standard systems require an additional constraint: no left-hand side may unify with a proper nonvariable subterm of any left-hand side. Such systems will be called *overlay* systems.

Theorem 3.4 (Dershowitz et al. [12]). *A terminating overlay standard conditional rewriting system is confluent if every critical pair is joinable.*

This theorem is a corollary of the following substitution lemma.

Lemma 3.5 (Substitution Lemma). *In any terminating overlay standard conditional rewriting system, for any terms s, t, r (containing any number of occurrences of s) and normal form N, if* $r[s, s, \ldots s] \rightarrow^! N$ *and* $s \xrightarrow{*} t$ *hold, then* $r[s, t, \ldots t] \rightarrow^! N$ *also holds.*

To prove the lemma, we define the degree of a term s, $\deg(s)$, so that for any terms s and t, $\deg(s) < \deg(t)$ if s is a subterm of t or $t \rightarrow s$. The proof depends on a triple induction on $\langle a, b, c\rangle$, where a is $\deg(s)$, b is the depth of the given proof of $r[s, s, s] \rightarrow^! N$, and c is $\deg(r[s, s, \ldots s])$. See [12] for more details of the proof.

We should remark that interpreting Horn clauses as conditional rewrite rules (with right-hand side *true*) leads to an overlay system, because predicate symbols are never nested in the head of a clause. Furthermore, all critical pairs are joinable, since all right-hand sides are the same. This theorem also applies to pattern-directed functional languages in which defined functions may not be nested on left-hand sides.

Recall (from Section 2) that when one side of each condition is an irreducible term (like *true*), the system is said to be *normal*. Bergstra and Klop [3] extended the confluence result for orthogonal unconditional systems [39] to normal conditional systems. They showed that any left-linear (possibly nonterminating) normal system with no critical pairs is confluent. Since our interest here is solely in terminating systems, we can relax the "no critical pair" part. A critical pair $c\mu \wedge p\mu \supset s\mu[r\mu] = t\mu$, obtained from rules $c \mid l \rightarrow r$ and $p \mid s \rightarrow t$, is said to be *shallow joinable* if there exist a term v, a derivation $t\mu \rightarrow^* v$ with depth less than or equal to the depth of the rewrite $l\mu \rightarrow r\mu$, and a derivation $s\mu[r\mu] \rightarrow^* v$ with depth less than or equal to that of $s\mu \rightarrow t\mu$.

Theorem 3.6 (Dershowitz et al. [12]). *A left-linear terminating normal conditional rewriting system is confluent, if every critical pair is shallow joinable.*

This theorem is a corollary of the following.

Lemma 3.7 (Diamond Lemma). *Let R be a terminating left-linear, conditional rewrite normal system all critical pairs of which are shallow-joinable. Then, if* $u \rightarrow^* s$ *with depth at most m and* $u \rightarrow^* t$ *with depth at most n there exists a term v such that* $s \rightarrow^*_n v$ *with depth at most n and* $t \rightarrow^*_m v$ *with depth at most m.*

Proof. The proof is by induction on the pair $(m + n, u)$ with respect to the (lexicographic combination of the) natural ordering of natural numbers and the well-founded relation $\rightarrow$ on terms (cf. [12]). □

For the last result of this section, we have the following theorem (cf. [32]).

Theorem 3.8. *A decreasing standard conditional rewriting system is confluent if every critical pair is joinable.*

The proof uses a substitution lemma similar to Lemma 3.5. This substitution lemma for decreasing systems is proved by induction on the underlying well-founded ordering. In this proof, one need only consider normal systems (by Theorem 2.5).

By virtue of Theorems 2.3 and 3.3, one can also establish confluence of decreasing standard systems R^{std} by showing that the corresponding natural system R^{nat} is decreasing and all its critical pairs are joinable.

4. Conditional narrowing

The previous two sections concerned the use of conditional rewrite systems to reduce terms to their value and to answer universal queries of the form $\forall \bar{x}\, s[\bar{x}] = t[\bar{x}]$. In this and the following section, we discuss the application of rewriting techniques to solving existential queries of the form $\exists \bar{x}\; s[\bar{x}] = t[\bar{x}]$. This corresponds to the logic-programming capability of resolution-based languages like Prolog.

Narrowing has been proposed as an extension for solving goals in rewriting-based languages. *Conditional narrowing* may be defined as follows. Let $s{\downarrow}t$ be a goal. If s and t are unifiable, then the goal is said to "narrow to true" via their most general unifier. Alternatively, if there is a conditional rule $c \mid l \rightarrow r$ such that l unifies with a nonvariable subterm of s (or t) via most general unifier μ (the variables in the rule are renamed so that they are disjoint from those in s), then all the conditions in $c\mu$ are narrowed in tandem until they are solved, say via substitution ρ. Then we say that the top-level goal narrows to $s\mu\rho{\downarrow}t\mu\rho$ via the composed substitution $\mu\rho$. Thus, narrowing is a "linear" process: rules are overlapped only on goals, not on other rules.

For example, given the standard conditional system for *append* in Section 2, the goal $append(x, y){\downarrow}x$ narrows using the last rule if $null(x){\downarrow}false$ narrows to true. By using (the renamed rule) $null(cons(u, v)) \rightarrow false$, we can solve the condition, narrowing the original goal to $cons(car(cons(u, v)), append(cdr(cons(u, v)), y)){\downarrow}cons(u, v)$. Rewriting is a special case of narrowing; it reduces the above goal to $cons(u, append(v, y)){\downarrow}cons(u, v)$. This, in turn, is narrowable by the first rule for *append* if $null(v){\downarrow}true$ narrows to true. Solving, by letting v be nil, we narrow to a new goal $cons(u, y){\downarrow}cons(u, v)$. Since the two terms are now unifiable (letting $y = v$), narrowing has produced the solution $x = cons(u, v) = cons(u, nil)$ and $y = v = nil$.

In the unconditional case, it has been shown that narrowing is complete for any (ground) convergent system [25]. By *complete*, we mean that if there exists a substitution σ such that $s\sigma \leftrightarrow^* t\sigma$, then $s{\downarrow}t$ narrows to true. Similarly, the variant of narrowing in which terms are reduced to normal form before narrowing is

complete [17]. For conditional systems the analogous result is that (under the same assumptions) any equationally satisfiable goal can be solved by conditional narrowing.

Theorem 4.1 (Dershowitz and Plaisted [15]). *Narrowing is complete for ground convergent standard conditional rewriting systems (with no extra variables in conditions).*

The restriction that all variables occurring in conditions also appear on the left will be lifted in the next section.

For ground convergent systems, all goals may be reduced to normal form before any narrowing step. Simplification, that is, reduction via terminating rules, is a very powerful feature, particularly when defined function symbols are allowed to be arbitrarily nested in left-hand sides. Assuming ground confluence and termination, any strategy can be used for simplification. Furthermore, negation can be partially handled by incorporating negative information in the form of rewrite rules, which are then used to simplify subgoals to *false*. Combined with eager simplification, this approach has the advantage of allowing unsatisfiable goals to be pruned, thereby avoiding some potentially infinite narrowing paths (see [14]). Normalizing before narrowing is not necessary, however, and other language proposals employ different strategies. Some superfluous paths (that cannot lead to solutions) can be avoided by making a distinction between constructor symbols and defined ones (assuming that terms built entirely from constructors are irreducible). Two terms headed by different constructors can never be equal; when headed by the same constructor, they are equal if and only if their respective arguments are equal. See, for example [14, 19, 31, 49]. Other restrictions and variations of narrowing which preserve completeness are included in [25, 38, 16].

Even if a program is ground convergent, alternative narrowing derivations must be explored if completeness is to be assured. Thus, narrowing-based languages that deterministically choose one possible narrowing over others cannot guarantee that solutions will be found. Preprocessing and structure-sharing techniques for rewriting and narrowing are explored in [27].

5. Extra variables

Traditional rewriting theory (e.g. [23]) usually has a constraint on occurrences of variables, namely that every variable occurring on a right-hand side of a rule also occurs on the corresponding left-hand side. A natural extension of this constraint for conditional rules is that every variable occurring either in a condition or on the right-hand side also occurs on the left. But if conditional rules are to generalize Horn-clause programming, such a constraint is unacceptable, since even very simple relations, such as transitivity, require extra variables in conditions.

Accordingly, we can redefine rewriting in the extra-variable case as follows: $u[l\sigma] \to u[r\sigma\tau]$ for a rule $c \mid l \to r$ if there exists a substitution τ for the new variables such that $c\sigma\tau$ holds. As an example, let us replace the rules for *append* with

$$append(nil, y) \to y$$

$$x \downarrow cons(u, v) \wedge append(v, y) \downarrow z \mid append(x, y) \to cons(u, z),$$

where u, v, and z are extra variables. The last rule is applicable if there *exist* substitutions for the extra variables that make the conditions hold. Now we have $append(cons(a, nil), nil) \to cons(a, nil)$, since $u = a$, $v = nil$, and $z = nil$ is a solution to the conditions. Operationally, narrowing may be used to solve conditions with extra variables; the definition of narrowing is unchanged.

Even with extra variables, Theorems 3.4 and 3.6 hold as stated. Allowing extra variables, however, does introduce a problem: ground confluence no longer guarantees the completeness of the narrowing mechanism [3b]. The following theorem allows for extra variables, at the expense of a stronger confluence condition, called "level-confluence". A standard system R is (*ground*) level-confluent if there exists a term v such that whenever $s \,{}^*\!\leftarrow u \to^* t$ with a maximum depth of n, there is a rewrite proof $s \downarrow t$ of depth no greater than n.

Theorem 5.1 (Bosco et al. [3b]). *Narrowing is complete for terminating ground level-confluent standard conditional rewriting systems.*

The proof of this theorem (as well as the previous theorem) is based on the following.

Lemma 5.2. *Let R be a standard conditional rewrite system (possibly having extra variables). If $s\sigma \to^* t$, σ is an irreducible substitution, and all the instances of rewrite rules use in the proof (not only the surface proof but also subproofs for conditions) are irreducible, then there exist a term u and substitutions η and τ such that s narrows to u via η, $u\tau = t$, and $\eta\tau = \sigma$.*

By "irreducible substitution", we mean that the substitution maps all variables to irreducible terms; by "irreducible instance", we mean that the rule is applied to a subterm having irreducible terms matching left-hand side variables and any extra variables appearing in conditions.

The lemma is proved by double induction on the depth and length of the derivation $s\sigma \to^* t$. If the derivation is empty (has zero steps), then $s\sigma = t$, and the result is obvious. Suppose then that $s\sigma$ is first reduced using some rule $p|l \to r$ in R. Since σ is irreducible, it must be that s has a nonvariable subterm u such that $u\sigma$ is an instance $l\theta$ of l, i.e. $s\sigma[u\sigma] = s\sigma[l\theta] \to s\sigma[r\theta] \to^* t$ and $p\theta$ narrows to true where θ is a substitution for variables in l and/or p. Let μ be the most general unifier of u and l. Then, for some irreducible substitution τ, we have $\sigma = \mu\tau$ when μ is restricted to the variables in s, and $\theta = \mu\tau$ when μ is restricted to variables in l.

Since $p\theta = p\mu\tau$, by induction $p\mu$ narrows to true via some ρ such that $\tau = \rho\eta$ for some irreducible η. By the definition of a single narrowing step, $s[u]$ narrows to $s\mu\rho[r\mu\rho]$ via $\mu\rho$. Since $s\sigma[r\theta] = s\mu\rho\eta[r\mu\rho\eta] \rightarrow^* t$, by induction $s\mu\rho[r\mu\rho]$ narrows to u via some ϕ, $\eta = \phi\psi$, and $t = u\psi$ for some ψ. Hence, we have $s[u]$ narrows to u via $\mu\rho\phi$, $\sigma = \mu\rho\phi\psi$, and $t = u\psi$, as desired.

Unfortunately, level-confluence of critical pairs does not ensure level-confluence of the system, as can be seen from the following counterexample:

$$h(f(a)) \rightarrow c$$

$$d{\downarrow}d \mid h(x) \rightarrow j(x)$$

$$d{\downarrow}d \mid c \rightarrow j(f(a))$$

$$d{\downarrow}d \mid a \rightarrow b$$

$$c \rightarrow d$$

$$j(g(b)) \rightarrow d$$

$$d{\downarrow}h(f(x)) \mid f(x) \rightarrow g(x).$$

This normal system is terminating and every critical pair is level-joinable, but despite the fact that $f(b) \leftarrow f(a) \rightarrow g(a) \rightarrow g(b)$, narrowing cannot solve the goal $f(b){\downarrow}g(x)$. However, since shallow-joinable critical pairs are level-joinable, we can apply Theorem 3.6, thereby ensuring completeness of narrowing for terminating left-linear shallow-joinable normal systems. The following, similar counterexample demonstrates the need for left-linearity:

$$k(a, a){\downarrow}c \mid h(f(a)) \rightarrow p(a)$$

$$k(a, a){\downarrow}c \mid h(x) \rightarrow j(x)$$

$$p(b) \rightarrow j(f(a))$$

$$k(a, a){\downarrow}c \mid a \rightarrow b$$

$$k(x, a){\downarrow}c \mid p(x) \rightarrow q(x)$$

$$q(b) \rightarrow j(g(a))$$

$$h(f(x)){\downarrow}c \mid f(x) \rightarrow g(x)$$

$$j(g(b)) \rightarrow c$$

$$k(x, x) \rightarrow c.$$

An alternative approach to new variables can be based on the notion of decreasing systems. There is of course no way one can insist that left-hand sides be greater than *all* instances of a condition containing a new variable. Instead, we revise our definition of rewriting in the extra-variable case, and define $u[l\sigma] \rightarrow u[r\sigma\tau]$ for a rule $c \mid l \rightarrow r$ only if there exists an *irreducible* substitution τ for the new variables such that $c\sigma\tau$ holds. Then we can say that a system R is decreasing if there exists

a well-founded ordering containing the (new) rewrite relation $\rightarrow$ and the proper subterm relation $\triangleright$, and for which $l\sigma$ is greater than both terms of each condition in $c\sigma\tau$, for any irreducible substitution τ.

For example, the above *append* system is decreasing in this sense. Note that the joinability of the conditions must take the form $x \rightarrow^* cons(u, v) \wedge append(v, y) \rightarrow^* z$, if the new variables are irreducible.

Employing the above lemma again, the following theorem can be shown.

Theorem 5.3. *Narrowing is complete for decreasing ground-confluent standard conditional rewriting systems.*

Related ideas appear in [3a].

6. Built-in predicates and functions

In this section, we consider programs utilizing built-in predicates and functions to evaluate the value for values of given terms. We first consider built-in predicates. A rule may have the form

$$P(t[\bar{x}], s[\bar{x}]) \wedge Q(u[\bar{x}]) \mid l[\bar{x}] \rightarrow r[\bar{x}],$$

where P and Q are built-in predicates. For example, we can define *append* as before, using built-in *null* and *nonnull* predicates:

$$null(x) \mid append(x, y) \rightarrow y$$

$$nonnull(x) \mid append(x, y) \rightarrow cons(car(x), append(cdr(x), y)).$$

We assume that a built-in predicate evaluates the Boolean value of a term by some mechanism that is independent of rewriting (e.g. by hardware, or using a theorem-prover). This is like the "hierarchical" conditional case of [56]. We allow the usual equational conditions, as before, as well as a mixture of built-in predicates and equations such as

$$P(t[\bar{x}], s[\bar{x}]) \wedge Q(u[\bar{x}]) \wedge v[\bar{x}]\downarrow w[\bar{x}] \mid l[\bar{x}] \rightarrow r[\bar{x}].$$

Here we assume that built-in predicates evaluate truth values only for terms in normal form. (In practical applications, they may evaluate only for *ground* normal terms.) More precisely, a rewriting system is usually based on a many-sorted theory, in which case a built-in predicate evaluates the truth value for the values (normal forms) of given types. For example, if a predicate $P(x_1, \ldots, x_m)$ has variables of types $A_1, A_2, \ldots, A_m$, then for any values (normal forms) $n_1, n_2, \ldots, n_m$ of types $A_1, A_2, \ldots, A_m$, $P(n_1, \ldots, n_m)$ returns Boolean value *true* or *false*. In an actual implementation, the normal form of a given type in the language of the rewriting system needs to be interpreted as a value of the corresponding built-in type. For example, for an arithmetical predicate $P(x)$ of type Natural and for a ground term t of type Natural, the rewriting system first executes $P(t)$ to reach a normal form,

say $P(s(s(s(0))))$, and interprets it as $P(3)$, then the built-in execution evaluates $P(3)$. If $P(3)$ evaluates to *true*, the condition is satisfied.

In this new setting, the results of earlier sections can be extended. The definition of a decreasing system should be modified as follows. A system (with built-ins) is *decreasing* if there exists a well-founded extension $>$ of the rewrite relation $\rightarrow$ which satisfies

(a) $>$ contains the proper subterm relation $\triangleright$ and

(b) for each rule $s_1{\downarrow}t_1 \wedge \cdots \wedge s_n{\downarrow}t_n \wedge P_1(u_1) \wedge \cdots \wedge P_m(u_m) \mid l \rightarrow r$, $l\sigma > s_i\sigma, t_i\sigma, u_j\sigma$ for all substitutions σ and indices i $(1 \leqslant i \leqslant n)$ and j $(1 \leqslant j \leqslant m)$. With this definition, the earlier theorems on decreasing systems still hold.

Theorem 6.1. *For a decreasing standard conditional rewriting system with built-in predicates*:

(a) *the basic notions are decidable, i.e. the rewrite relation* $(\rightarrow)$, *derivability relation* $(\rightarrow^*)$, *joinability relation* $(\downarrow)$, *and normal form attribute are all recursive* (*cf. Theorem* 3.1);

(b) *the system is confluent if every critical pair is joinable* (*cf. Theorem* 3.8);

(c) *when confluent, it is equivalent to the corresponding natural system* (*cf. Theorem* 2.3).

Proof. The earlier induction arguments on the underlying well-founded ordering $<$ still hold. We use a substitution lemma similar to Lemma 3.5 for (b). Here, in the induction step we need to show, for example, that if $P(u[s])$ is *true* and $s \rightarrow^* t$ then $P(u[t])$ is also *true*. This statement can be reduced to the statement that if $u[s]{\downarrow}N$ and $s \rightarrow^* t$ then $u[t]{\downarrow}N$, where N is a normal form of $u[s]$. Hence, one can ignore the occurrence of built-in predicates in the induction argument. For (c) we regard a proof of $P(u[t])$ as a normal proof if a proof of $u[t] \rightarrow^* N$ is normal, where N is a normal term. □

The same reduction argument as in the proof of the substitution lemma for (b) above leads to a modified proof of the substitution lemma corresponding to Lemma 3.5 for an overlay system, which implies the following.

Theorem 6.2. *A terminating overlay standard conditional rewriting system with built in predicates is confluent if every critical pair is joinable.*

We define the depth of a proof $P(u[t])$ to be the depth of the subproof of $u[t] \rightarrow^* N$, where N is a normal form. In other words, once a term $u[t]$ reaches a normal form N, we regard the evaluation process of $P(N)$ as a zero-depth proof. We can now use the notion of shallow joinability based on this modified definition of depth. Using a reduction argument, we can re-prove the Diamond Lemma (Lemma 2.2), which implies the following.

Theorem 6.3. *A left-linear terminating normal conditional rewriting system with built-in predicates is confluent if every critical pair is shallow joinable.*

Note that built-in predicates typically evaluate only ground terms. In such a situation, the above convergence criteria are actually criteria for ground convergence only.

Our earlier results on conditional narrowing can be extended to the case in which built-in predicates occur, but completeness requires that built-ins also *solve* for variables (not normally the case). In particular, ground confluence for standard systems ensures completeness of conditional narrowing. The argument for completeness of conditional narrowing with extra variables holds true even with built-in predicates and extra variables, but the definition of level confluence must be modified to use the above re-definition of proof-depth. The definition of a decreasing system with extra-variable conditions (from the previous section) also applies to built-in predicates.

A condition expressed by a built-in predicate, say $P(t[x])$ can also be expressed by using the corresponding built-in function, say G in the form of $G(t[x]) = true$. Here, G is the characteristic function of P such that $G(s) = true$ iff $P(s)$ for any normal term (or ground normal term) of the underlying type. This use of a built-in function can naturally be extended to a function of any type, i.e. a function, say G of type $P_1 \times P_2 \times \cdots \times P_n \supset Q$, instead of type Boolean. Hence we assume that the value of $G(m_1, \ldots, m_n)$ for given (ground) normal terms $m_1, \ldots, m_n$ of sorts $P_1, \ldots, P_n$ is a (ground, respectively) normal term of type Q. With this extension, all the arguments in this section still hold.

One may, of course, use composition of built-in functions to express conditions. For example, one can use an expression $F(G(t), H(s, u))$, where $F(G(*), H(*, *))$ is a composition of built-in functions and t, s and u are terms of the rewriting system. But we do not allow built-in functions as arguments of non-built-ins.

7. Conditional rewriting with higher-order terms

The expressiveness of functional languages owes much to the ease with which higher-order functions can be defined and used. Rewriting systems, on the other hand, are better suited to computing with first-order terms. This suggests that one might obtain even more expressive languages by combining a λ-calculus with (term) rewriting systems. It is by no means clear, a priori, that such a combination would have the necessary properties for computation. Klop [32a] has shown that, although the untyped λ-calculus has the Church–Rosser (confluence) property, the combination of untyped λ-calculus with a convergent term-rewriting system is not necessarily Church–Rosser. As is well known, the typed λ-calculus is convergent (terminating and Church–Rosser). Breazu-Tannen [5] noticed that the combination of the (convergent) typed λ-calculus and a convergent term-rewriting system is Church–Rosser,

i.e. the Church–Rosser property is preserved (cf. also [4]). The preservation of the termination property was proved in [44, 40, 4]. These two preservation results imply the preservation of the convergence property. Breazu-Tannen and Gallier [4] use polymorphically typed λ-calculus for those preservation results, as does [41]. The purpose of this section is to provide a basic framework for combined systems of λ-calculus and conditional term rewriting. We first consider a very simple version of a combined system with only unconditional rewriting and give a simple proof technique for termination. We show how to reduce the termination problem for a wide class of typed λ-calculi, and for more general rewrite systems, including the conditional case, to our simple framework.

It is easily seen that introduction of higher order terms provides more expressive power in rewriting rules. For example, consider the conditional or unconditional first-order rules for *append* given earlier, along with the following algebraic rules:

$$append(l, nil) \to l$$

$$append(append(l, m), n) \to append(l, append(m, n)),$$

and two additional rules with higher-order terms:

$$mapcan(X, nil) \to nil$$

$$mapcan(X, cons(x, l)) \to append(X(x), mapcan(X, l)).$$

Here, *nil* is a constant of base type *List*; *l*, *m* and *n* are variables of base type *List*; *append* is of type $List \times List \supset List$; x is a variable of base type, say *Integer*; *cons* is a function of constant type $Integer \times List \supset List$; X is a higher-order variable of type $Integer \supset List$; and *mapcan* is a higher-order constant of type $[Integer \supset List] \times List \supset List$.

Although the main result of this section holds for many versions of the typed λ-calculus, we fix our attention on the simply-typed λ-calculus, which we combine with a many-sorted rewrite theory. Each function symbol of arity n in the signature of the theory has a base type $P_1 \times P_2 \times \cdots \times P_n \supset Q$, where $P_1, P_2, \ldots, P_n$, and Q are base types. Given a set $\{P_1, P_2, \ldots, P_n\}$ of base types, we define the general notion of (higher order) type inductively:

(1) If P is a base type then P is a type.

(2) If A and B are types then $A \supset B$ is a type.

We will write $A_1 \supset (A_2 \supset (\cdots (A_n \supset B) \cdots))$ in the abbreviated form $A_1 \times A_2 \times \cdots \times A_n \supset B$.

Typed λ-terms are defined as follows. The language has countably many (free and bound) variables for each type. It may also contain constants.

(1) If x is a variable of type A then it is a λ-term of type A.

(2) If t is a λ-term of type B and x is a variable of type A then $\lambda x.t$ is a λ-term of type $A \supset B$.

(3) If t is a λ-term of type $A \supset B$ and s is a λ-term of type A, then $t(s)$ is a λ-term of type B.
(Juxtaposition is a binary operation.)

The pure typed λ-calculus is the reduction system consisting of the following rule (called *β-reduction*):

$$(\lambda x.t[x])(s) \rightarrow t[s].$$

where each occurrence of the bound variable x is replaced by s, a λ-term of the same type as x.

We consider combined systems, consisting of an arbitrary convergent term rewriting system and the typed λ-calculus as defined above. (Note that the *append* example does not fit this paradigm, since the rewrite rules contain higher-order variables, not just symbols of base types.) The following theorem is the principal result of this section.

Theorem 7.1. *If a term rewriting system R is terminating, then the combined system of R and the types λ-calculus is also terminating.*

Preservation of convergence follows as a corollary of this and local confluence (Sublemma 7.9); cf. [5].

To prove termination, we modify the strong normalization technique of [20, 37, 48, 54]. We assume a fixed version of the computability (reducibility) predicate used in the usual Tait–Girard type of strong normalization proof. For our argument to work, we make the following assumption:

Condition 7.2. If t is of a base type P, then $R_P(t)$ (meaning "t belongs to the reducibility set of type P") iff t is strongly normalizable (meaning "no infinite reduction sequences from t are possible").

The usual definitions of computability predicates for the various typed calculi satisfy this condition. This point is that under this common condition, one can factor out the computability predicate from the strong-normalizability argument, and just establish the Principal Case given below.

We give one example of a computability predicate (we will use "R-reduction" as a synonym for a rewrite step $\rightarrow_R$ in R). For each type A, we can define the *computability* predicate R_A in a way similar to [37, 48]:

(1) When t is of the form $\lambda x.s[x]$, for x of type A and s of type $A \supset B$, $R_{A \supset B}(t)$ if for all u of type A, $R_A(u)$ implies $R_B(s[u])$.

(2) When t is not of the form $\lambda x.s$, then $R_A(t)$ if t is a normal form (i.e. irreducible) with respect to both β-reduction and R-reduction, or if $R_A(u)$ for every term u obtainable from t by one β- or R-reduction step.

Lemma 7.3. *If $R_A(t)$ then t is strongly normalizable.*

Lemma 7.4. *For any term s of type A, if $R_A(s)$ and t reduces to s in one or more steps of β- or R-reduction, then $R_A(t)$.*

Both of these lemmas are proved by induction on the construction of the predicate R, as in [37, 48].

Lemma 7.5. *If $R_{A_1}(u_1), \ldots, R_{A_n}(u_n)$, then $R_B(t[u_1, \ldots, u_n])$ for any $t[x, \ldots, x]$ of type B, where $x, \ldots, x$ are all the free variables occurring in t and have types $A_1, \ldots, A_n$ respectively.*

Proof. The proof is by induction on the length of t. Since the other cases are similar to those in [37], we describe only the case in which the outermost operator is a function symbol, say f, in the signature of the rewrite system R.

In this case, t and any reduced form of t have a base type, say P. In particular, the symbol λ will not appear as the outermost symbol after any reductions for t. Hence, by definition of R, to show $R_P(t)$, it suffices to show the strong normalizability of t. To check for strong normalizability of the combined system, we need only establish the following one case; all other cases are treated exactly in the same way as the traditional argument for strong normalizability of the λ-calculus, by computability predicates.

Principal Case 7.6. *If $t = f(s_1, \ldots, s_n)$ is of a base type and $s_1, \ldots, s_n$ are of base type, then t is strongly normalizable.*

For any term t, we define the *estimated cap* (ec) of t. Assume that t is of the form $s[u_1, \ldots, u_n]$, where s is the maximum subterm of t such that s contains the top operator (the root) and is a term of the original language of R. Then s is the *cap* of t. If pure λ-terms except for variables of the base type are attached to the cap, then all such λ-terms of each base type are regarded as occurrences of the same *new* variable symbol of that base type when the cap is defined. Variables of base types attached to the cap remain (cf. [44, 40]).

For example, if $s[z(\lambda y \lambda x.y), x'(\lambda y.y), w(v), u]$ where s is in the signature of R, z and w have result type C, x' has result type B, and u is of type C, then the cap is $s[x, y, x, u]$, where x and u are variables of type C and y is of type B.

The estimated cap (ec) s of term t is the cap for the β-normal form of t (written $\beta\mathrm{NF}(t)$), which is the irreducible form of t with respect to β-reduction; the *β-normal* form is always determined uniquely. Hence the estimated cap is uniquely defined for any term t.

We prove this case by transfinite induction on $(ec(t), sub(t))$, where $sub(t)$ is the multiset $\{s_1, \ldots, s_n\}$. The pair $(ec(t), sub(t))$ is ordered lexicographically: the first component in the well-founded R-reduction relation, and the second in the multiset extension of the combined reduction relation. By our induction hypothesis,

$R_{A_1}(u_1), \ldots, R_{A_n}(u_n)$ imply $R_{P_1}(s_1), \ldots, R_{P_n}(s_n)$ for suitable base types $P_1, \ldots, P_n$. By Lemma 3.5, $s_1, \ldots, s_n$ are strongly normalizable. Hence, $sub(t)$ is well-founded.

We consider three subcases as follows:

(1) $\hat{t}$ is obtained from t by one-step β-reduction. Then obviously $sub(\hat{t}) < sub(t)$ and $ec(\hat{t}) = ec(t)$.

(2) $\hat{t}$ is obtained from t by one-step R-reduction, and the R-reduction applies only within $ec(t)$. Then $ec(\hat{t}) < ec(t)$.

(3) $\hat{t}$ is obtained from t by one-step R-reduction, and the R-reduction does not reduce any part of $ec(t)$. Obviously, $sub(\hat{t}) < sub(t)$. On the other hand, as we will see from Sublemmas 7.8 and 7.9, $ec(t) = ec(t)$.

Sublemma 7.7. *If λ-terms $s_1, \ldots, s_n$ are of base types and if $s_1, \ldots, s_n$ and $t[x, \ldots, x]$ are β-normal forms, then $t[s_1, \ldots, s_n]$ is also a β-normal form.*

This is obvious from the definition of β-normal form.

Sublemma 7.8. *If t a β-normal form, and if s is obtained from t by an R-reduction that does not reduce any part of the cap of t, then $ec(t) = ec(s)$.*

Even if some R-term part collapses by the R-reduction, there is no further possibility of β-reduction, by Sublemma 7.7 and there is no further possibility of collapsing any λ-term part. Hence, $ec(t) \equiv ec(s)$.

Sublemma 7.9 (Breazu-Tannen [5]). *If $t \rightarrow_R \hat{t}$, via some rule in R, then the β-normal form of t can be R-reduced to the β-normal form of $\hat{t}$ by (possibly repeated use of) the same R-rule.*

In particular, if the reduction $t \rightarrow_R \hat{t}$ is in case (3) above, then $\beta\mathrm{NF}(t) \rightarrow_R^* \beta\mathrm{NF}(t)$ is obtained by the reductions only from the part which does not reduce the cap. That is, the diagram in Fig. 1 commutes. $\square$

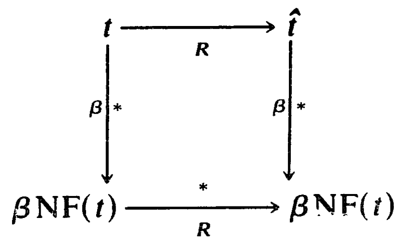

Fig. 1.

The proof of strong normalization can be extended to the combined system of the polymorphically typed λ-calculus and an arbitrary convergent rewrite system, using Girard's notation of the candidates of the computability predicates. For

example, one can follow the strong normalizability proof in [37]. Again, the essential difference between that and our result is in the last of the three subcases just mentioned. Otherwise, the proofs are essentially the same [44, 41, 4]. In particular, we can reduce our problem to the form of the Principal Case 7.6, and prove it by transfinite induction exactly the same way as before. This means that even if we go to the polymorphically typed system, we can reduce the whole problem to the Principal Case, which is independent of the details of the strong normalization argument (by candidates of reducibility).

The above proof of termination of the combined system of typed λ-calculus with a terminating term rewriting system does not require finiteness of the number of rewriting rules. At the same time, any conditional term rewriting system can be viewed as a set of (possibly infinitely many) unconditional rewrite rules by choosing those substitution instances of a rule for which the condition parts are satisfied. This unconditional system has the same rewrite relation as the conditional one. Therefore, the above termination results hold for any combined system of typed λ-calculus with a terminating conditional term rewriting system. Preservation of convergence follows from this and the fact that the confluence property is preserved [5].

Corollary 7.10. *If a conditional rewriting system R is terminating, then the combined system of R and the typed λ-calculus is also terminating.*

Now we consider the inclusion of the following rule (called *η-reduction*):

$$\lambda x.(t(x)) \to t,$$

where x does not appear in t as a free variable. The analogue of Theorem 7.1 for termination (strong normalization) continues to hold and the proof is essentially the same as before. Especially, all the lemmas and sublemmas above hold, with β-normal form replaced by β-η-normal form in the sublemmas. The only hitch is that Sublemma 7.7 does not hold as is for β-η-normal form, but one parallel η-reduction returns the terms to normal form. With this modification, the rest of the proof can be carried through (cf. [4]).

The above termination and convergence results also hold for higher order rewrite systems, if the rewrite rules are restricted to the form of higher-order primitive recursive definition based on given constructor terms [44, 29]. Here, the primitive recursion takes the following form: Let F be a new higher-order constant. For each constructor symbol f_j of the same base type, there is a rule

$$F(f_j(y_1,\ldots,y_n),\bar{x}) \;\to\; s[F(y_1,\bar{x}),\ldots,F(y_n,\bar{x}),y_1,\ldots,y_n,\bar{x}],$$

and for each constant c_k of that type, there is a rule

$$F(c_k,\bar{x}) \;\to\; t[\bar{x}],$$

where s and t are (a higher-order term) composed of predefined constants and any of the variables and expressions in the square brackets. We also allow explicit

definitions of the form:

$$F(\bar{x}) \to t[\bar{x}],$$

where t is as above (in particular, F does not appear in t). This is a natural extension of Gödel's System T to arbitrary inductive data structures. The rules for *mapcan* at the beginning of this section are a simple example.

Theorem 7.11. *If a first-order conditional rewrite system is terminating, then the system extended by higher-order constructor-based primitive recursive rules and the typed λ-calculus is also terminating.*

The usual proof (by computability predicates) for termination of System T (based on the typed λ-calculus) can be extended to include these primitive-recursive higher-order functionals. Then, the combination of this system with a terminating set of first-order rules is terminating, as before.

In the above, if we relax the constraint on the form of higher-order rewriting, then this theorem no longer holds, i.e., there is a terminating higher-order rewrite system which, when combined with β-reduction, is nonterminating: Consider the higher-order rule: $F(X(x), x) \to F(X(x), X(x))$, where F is a higher-order functional constant and X is a functional variable. If one substitutes $\lambda x.x$ for X, one gets an infinite reduction sequence with this rule and β-reduction:

$$F((\lambda x.x)(x), x) \to F((\lambda x.x)(x), (\lambda x.x)(x)) \to F((\lambda x.x)(x), x) \to \cdots.$$

In the previous two sections, we considered extensions of expressive power in conditions, allowing existential quantifiers and built-in predicates and functions. As the last topic of this section, we consider another extension to expressivity by allowing mixed terms with λ-expressions in conditions (not in the rule). We consider a combined system consisting of a set of such rewrite rules with β-reduction. If one can transform the set of conditional rewrite rules to an equivalent terminating set of (maybe infinitely many) first-order rewrite rules (not containing λ-expressions), then one can apply Theorem 7.1 for strong normalization of the combined system. In particular, if the free variables are all of base types, one can substitute all normal forms of base types (which do not contain λ-expressions) for them.

8. Conclusion

There is much to be gained from a theory that supports several high-level paradigms of programming. In this paper, we have presented new results which make conditional term rewriting an obvious candidate for such a theory. Our results help demonstrate that, although the theory of conditional term rewriting systems is more subtle than that of unconditional systems, it is no less useful. Conditional systems are considerably more expressive and natural than unconditional systems and for this reason alone they deserve study. Finally, by proving the convergence

of combined higher-order systems, we demonstrate that term rewriting can be incorporated within existing functional frameworks without the loss of important correctness properties.

References

[1] L. Bachmair, N. Dershowitz and J. Hsiang, Orderings for equational proofs, in: *Proc. IEEE Symp. on Logic in Computer Science*, Cambridge, MA (1986) 346-357.

[2] M. Bellia and G. Levi, The relation between logic and functional languages: a survey, *J. Logic Programming* 3(3) (1986) 217-236.

[3] J.A. Bergstra and J.W. Klop, Conditional rewrite rules: confluency and termination, *J. Comput. System Sci.* **32** (1986) 323-362.

[3a] H. Bertling and H. Ganzinger, Completion-time optimization of rewrite-time goal solving, in: *Proc. 3rd Internat. Conf. on Rewriting Techniques and Applications*, Chapel Hill, NC, Lecture Notes in Computer Science **355** (Springer, Berlin, 1989) 45-58.

[3b] P.G. Bosco, E. Giovannetti and C. Moiso, Refined strategies for semantic unification, in: *Proc. Internat. Joint Conf. on Theory and Practice of Software Development*, Pisa, Italy, Lecture Notes in Computer Science **250** (Springer, Berlin, 1987) 276-290.

[4] V. Breazu-Tannen and J. Gallier, Polymorphic rewriting conserves algebraic strong normalization and confluence, in: *Proc. ICALP, 1989*, Lecture Notes in Computer Science **372** (Springer, Berlin, 1989).

[5] V. Breazu-Tannen, Combining algebra and higher-order types, in: *Proc. 3rd IEEE Symp. on Logic in Computer Science*, Edinburgh (1988) 82-90.

[6] J. Darlington, A.J. Field and H. Pull, The unification of functional and logic languages, in: D. DeGroot and G. Lindstrom, ed., *Logic Programming* (1986) 37-70.

[7] D. DeGroot and G. Lindstrom, eds., *Logic Programming: Functions, Relations, and Equations* (Prentice Hall, Englewood Cliffs, NJ, 1986).

[8] P. Deransart, An operational algebraic semantics of PROLOG programs, Internal report, Institut National de Recherche en Informatique et Automatique, Le Chesnay, France, 1983.

[9] N. Dershowitz, Computing with rewrite systems, Technical Report ATR-83(8478)-1, Information Sciences Research Office, The Aerospace Corp., El Segundo, CA, January 1983; revised version appeared in *Inform. and Control* **64** (1985) 122-157.

[10] N. Dershowitz and J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (North-Holland, Amsterdam, 1990) 243-320.

[10a] N. Dershowitz and Z. Manna, Proving termination with multiset orderings, *Comm. ACM* **22**(8) (1979) 465-476.

[11] N. Dershowitz and M. Okada, Proof-theoretic techniques and the theory of rewriting, in: *Proc. 3rd IEEE Symp. on Logic in Computer Science*, Edinburgh, Scotland (1988) 104-111.

[12] N. Dershowitz, M. Okada and G. Sivakumar, Confluence of conditional rewrite systems, in: *Proc. 1st Internat. Workshop on Conditional Rewriting*, Orsay, France, Lecture Notes in Computer Science **308** (Springer, Berlin, 1988) 31-44.

[13] N. Dershowitz, M. Okada and G. Sivakumar, Canonical conditional rewrite systems, in: *Proc. 9th Conf. on Automated Deduction*, Argonne, IL, Lecture Notes in Computer Science **310** (Springer, Berlin, 1988) 538-549.

[14] N. Dershowitz and D.A. Plaisted, Logic programming *cum* applicative programming, in: *Proc. IEEE Symp. on Logic Programming*, Boston, MA (1985) 54-66.

[15] N. Dershowitz and D.A. Plaisted, Equational programming, in: J.E. Hayes, D. Michie and J. Richards, eds., *Machine Intelligence 11: The Logic and Acquisition of Knowledge* (Oxford Press, Oxford, 1988) 21-56.

[16] N. Dershowitz and G. Sivakumar, Goal-directed equation solving, in: *Proc. 7th National Conf. on Artificial Intelligence*, St. Paul, MN (1988) 166-170.

[17] M. Fay, First-order unification in an equational theory, in: *Proc. 4th Workshop on Automated Deduction*, Austin, TX (1979) 161-167.

[18] L. Fribourg, Oriented equational clauses as a programming language, *J. Logic Programming* **1** (1984) 179-210.

[19] L. Fribourg, SLOG: a logic programming language interpreter based on clausal superposition and rewriting, in: *Proc. IEEE Symp. on Logic Programming*, Boston, MA (1985) 172-184.

[20] J.Y. Girard, Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, in: *Proc. 2nd Scandinavian Logic Symp.* (1971) 63-92.

[21] J.A. Goguen and J. Meseguer, Equality, types, modules and (why not?) generics for logic programming, *Logic Programming* **1**(2) (1984) 179-210.

[22] J.A. Goguen and J. Meseguer, EQLOG: Equality, types, and generic modules for logic programming, in: D. DeGroot and G. Lindstrom, eds., *Logic Programming* (1986) 295-363.

[23] G. Huet, Confluent reductions: abstract properties and applications to term rewriting systems, *J. Assoc. Computing Mach.* **27**(4) (1980) 797-821.

[24] G. Huet and D.C. Oppen, Equations and rewrite rules: a survey, in: R. Book, ed., *Formal Language Theory: Perspectives and Open Problems* (Academic Press, New York, 1980) 349-405.

[25] J.-M. Hullot, Canonical forms and unification, in: *Proc. 5th Conf. on Automated Deduction*, Les Arcs, France, Lecture Notes in Computer Science **87** (Springer, Berlin, 1980) 318-334.

[26] B. Jayaraman and F.S.K. Silbermann, Equations, sets, and reduction semantics for functional and logic programming, in: *Proc. ACM Conf. on LISP and Functional Programming*, Cambridge, MA (1986) 320-331.

[27] N.A. Josephson and N. Dershowitz, An implementation of narrowing, *J. Logic Programming* **6**(1, 2) (1989) 57-77.

[28] J.-P. Jouannaud and B. Waldmann, Reductive conditional term rewriting systems, in: *Proc. 3rd IFIP Working Conf. on Formal Description of Programming Concepts*, Ebberup, Denmark (1986).

[29] J.-P. Jouannaud and M. Okada, Rewriting with higher order terms, in preparation.

[30] K.M. Kahn, Uniform—A language based upon unification which unifies much of Lisp, Prolog and Act 1, in: *Proc. 7th Internat. Joint Conf. on Artificial Intelligence*, Vancouver, B.C. (1981) 933-939.

[31] T. Kanamori, Computation by meta-unification with constructors, Report TR-152, Institute for New Generation Computer Technology, Tokyo, Japan, 1985.

[32] S. Kaplan, Simplifying conditional term rewriting systems: unification, termination and confluence, *J. Symbolic Comput.* **4**(3) (1987) 295-334.

[32a] J.W. Klop, *Combinatory Reduction Systems*, Mathematical Centre Tracts **127** (Centre for Mathematics and Computer Science, Amsterdam, 1980).

[33] D.E. Knuth and P.B. Bendix, Simple word problems in universal algebras, in: J. Leech, ed., *Computational Problems in Abstract Algebra* (Pergamon Press, Oxford, 1970) 263-297.

[34] H.J. Komorowski, QLOG—The programming environment for PROLOG in LISP, in: K.L. Clarke and S.-A. Tärnlund, eds., *Logic Programming* (Academic Press, New York, 1983) 315-322.

[35] G. Lindstrom, Functional programming and the logic variable, in: *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, LA (1985) 266-280.

[36] Y. Malachi, Z. Manna and R.J. Waldinger, TABLOG: the deductive tableau programming language, in: *Proc. ACM Symp. on LISP and Functional Programming*, Austin, TX (1984) 323-330.

[37] P. Martin-Löf, Haupsatz for the theory of species, in: *Proc. 2nd Scandinavian Logic Symp.* (1971) 217-233.

[38] A. Martelli, C. Moiso and G.F. Rossi, An algorithm for unification in equational theories, in: *Proc. IEEE Symp. on Logic Programming*, Salt Lake City, UT (1986) 180-186.

[39] M.J. O'Donnell, *Computing in Systems Described by Equations*, Lecture Notes in Computer Science **58** (Springer, Berlin, 1977).

[40] M. Okada, Strong normalizability for the combined system of the pure typed λ-calculus and an arbitrary convergent term rewrite system, in: *Proc. Internat. Symp. on Symbolic and Algebraic Computation* (1989).

[41] M. Okada and P. Grogono, Practical application of conditional term rewriting systems, in: *Proc. IX Conf. of the Chilean Computer Science Society/XV Latin American Conf. on Informatics* (1989).

[42] M. Okada and P. Grogono, New results in term rewriting theory, in: *Proc. of the Internat. Conf. on Symbolic and Logic Computation* (1989).

[43] M. Okada, Note on a proof of the extended Kirby-Paris game for labeled finite trees, *European J. Combin.* **9** (1988) 249-253.

[44] M. Okada, Introduction to proof theory for computer science, Unpublished lecture notes, Laboratoire de Recherche Informatique, Universite de Paris-Sud, Orsay, France, 1988.

[45] M. Okada, Proof-theoretic ordinals and ordering structures in term rewriting theory, IPSJ Report 86-20, Information Processing Society of Japan, 1987.

[46] M. Okada, A logical analysis for the theory of conditional rewriting, in: *Proc. of the First Internat. Workshop on Conditional Rewriting*, Orsay, France, Lecture Notes in Computer Science **308** (Springer, Berlin, 1988) 179–196.

[47] D.A. Plaisted, Semantic confluence tests and completion methods, *Inform. and Control* **65**(2/3) (1985) 182–215.

[48] D. Prawitz, Ideas and results in proof theory, in: *Proc. 2nd Scandinavian Logic Symp.* (1971) 235–307.

[49] U.S. Reddy, Narrowing as the operational semantics of functional languages, in: *Proc. IEEE Symp. Logic Programming*, Boston, MA (1985) 138–151.

[50] J.A. Robinson, Beyond LOGLISP—Combining functional and relational programming in a reduction setting, in: J.E. Hayes, D. Michie and J. Richards, eds., *Machine Intelligence 11: The Logic and Acquisition of Knowledge* (Oxford Press, Oxford, 1988) 1–20.

[51] M. Sato and T. Sakurai, QUTE: a functional language based on unification, in: *Proc. Internat. Conf. on Fifth Generation Computer Systems*, Tokyo, Japan (1984) 157–165.

[52] J.R. Slagle, Automated theorem-proving for theories with simplifiers, commutativity, and associativity, *J. Assoc. Comput. Mach.* **21**(4) (1974) 622–642.

[53] G. Smolka, FRESH: a higher-order language with unification and multiple results, in: D. DeGroot and G. Lindstrom, eds., *Logic Programming* (1986) 469–524.

[54] W.W. Tait, Intensional interpretation of functionals of finite type, *J. Symbolic Logic* **32** (1967) 198–212.

[55] H. Tamaki, Semantics of a logic programming language with a reducibility predicate, in: *Proc. IEEE Symp. on Logic Programming*, Atlantic City, NJ (1984) 259–264.

[56] H. Zhang and J.-L. Rémy, Contextual rewriting, in: *Proc. 1st Intern. Conf. on Rewriting Techniques and Applications*, Dijon, France, Lecture Notes in Computer Science **202** (Springer, Berlin, 1985) 46–62.