Conditional Rewriting

Nachum Dershowitz and David A. Plaisted

February 1985

1. INTRODUCTION

In this paper, we indicate how conditional (directed) equations provide a paradigm of computation that combines the clean syntax and semantics of both the logic programming and applicative/functional programming paradigms in one, uniform manner. Programming styles such as this impact verification in two ways: by reducing the disparity between specification and programming languages and by provided a potentially convenient language in which to specify, if not implement, verifier components.

Equations can be used to compute by repeatedly substituting equal terms in a given formula, until the simplest form possible is obtained. Such a computation scheme is similar to that of "applicative", or "functional" programming languages, such as LISP and its "pattern-directed" derivatives. Programs can also be written as a set of equations between formulas or equivalences between statements in logic, and then executed by applying a special-purpose theorem prover that derives consequences from the given formulas until the desired output values are obtained. This latter form of computation is similar to the "logic-programming" paradigm, as exemplified by PROLOG.

A *conditional equation* is a statement of the form

$$p \supset l{=}r,$$

meaning that the term l is equal to the term r when the condition p holds. In general, there may be variables x, y, etc. in p, l, and/or r, in which case the conditional equation is meant to hold for *all* terms x, y, etc. If any term containing l is "more complicated" than that term with r in place of l, then the conditional equation may be *directed*. In that case, we write the equation as a *conditional rewrite rule:*

$$l \quad :\!- \quad p \quad \rightarrow \quad r.$$

Either (or both) of the rule parts $:\!-p$ and $\rightarrow r$ may be omitted, in which case it is taken to be **true**. (No left-hand side should be the term **true**.) A rule with only a left-hand side l is called an *assertion;* one with no condition p is a *rewrite rule;* one with no right-hand side r is a *logic rule*.

A *rewrite system* is a collection of such conditional, directed equations, and serves as a (nondeterministic) program for computing a set of output values that satisfy a given goal. The two uses of rewrite systems, for straightforward computation by simplification and for computation by generating consequences, may be combined in a single program. Therein lies the power of the proposed language. Related work includes [Barbutti,*etal.*-85, Chester-80, Robinson-Sibert-82, Komorowski-82, Goguen-Meseguer-84, Malachi-Manna-Waldinger-84, Fribourg-84, Reddy-85].

In the next section we consider functional programming using simplification and rewriting, and in the section that follows we consider logic programming using unification and narrowing.

*This research was supported in part by the National Science Foundation under Grant MCS-83-07755.

2. FUNCTIONAL PROGRAMMING

A *(conditional) rewrite system* R is a finite set of rewrite rules, each of the form

$$l[\bar{x}] \quad :\!- \quad p[\bar{x}] \quad \rightarrow \quad r[\bar{x}],$$

where l and r are terms and p is a predicate. The ***right-hand side*** r and the ***condition*** p can only contain variables $\bar{x}$ that appear in the ***left-hand side*** l. (This restriction will be eased in the next section.) Such a rule may be applied to a term t if a subterm s of t matches the left-hand side l with some substitution $\bar{\sigma}$ of terms for the variables $\bar{x}$ appearing in l and if the corresponding condition $p[\bar{\sigma}]$ is true. The rule is applied by replacing the subterm $s[\bar{\sigma}]{=}l[\bar{\sigma}]$ in t with the corresponding right-hand side $r[\bar{\sigma}]$ of the rule, after the same substitution of terms for variables has been made. The choice of which rule to apply where is made nondeterministically from amongst all possibilities. We write $t{\Rightarrow}t'$ to indicate that a term t' is ***derivable*** from the term t by a single application of some rule in R; by $t{\Rightarrow^*}t'$ (read t ***reduces*** to t') we mean $t\Rightarrow \cdots \Rightarrow t'$ via zero or more rule applications. When we said above that $p[\bar{\sigma}]$ must be true for the rule to be applied, we meant $p[\bar{\sigma}]{\Rightarrow^*}\mathbf{true}$, i.e. that $p[\bar{\sigma}]$ reduces to the constant **true**. There is no backtracking over reductions. (There is backtracking over deductions, as we will see in the next section.)

For example, the following is an unconditional rewrite system for appending two lists:

List Append		
$Append(x{\cdot}X,Y)$	$\rightarrow$	$x{\cdot}Append(X,Y)$
$Append(\mathbf{nil},Y)$	$\rightarrow$	Y
$Append(Y,\mathbf{nil})$	$\rightarrow$	Y

where **nil** is the empty list and $\cdot$ (*cons*) adds a single element to the head of a list. (Throughout this paper we use the following conventions: bold-face for standard built-in functions, lower-case for individual variables and functions that return atoms, capitalization for list functions and variables.)

Any functional program definition of the form

$$f(\bar{x}) \quad ::= \quad \mathbf{if}\ p(\bar{x})\ \mathbf{then}\ r(\bar{x})\ \mathbf{else}\ s(\bar{x})$$

can be translated into

$$\begin{aligned} f(\bar{x}) &\rightarrow f'(p(\bar{x}),\bar{x}) \\ f'(\mathbf{true},\bar{x}) &\rightarrow r(\bar{x}) \\ f'(\mathbf{false},\bar{x}) &\rightarrow s(\bar{x}), \end{aligned}$$

where f' is a new function symbol. This has the effect of ensuring that the condition is evaluated only once before either branch is explored. If, for some value of $\bar{x}$, the condition $p(\bar{x})$ evaluates to **true**, the term $f(\bar{x})$ is eventually replaced by $r(\bar{x})$; if $p(\bar{x})$ evaluates to **false**, then $\mathbf{not}(p(\bar{x}))$ evaluates to **true** and $f(\bar{x})$ is replaced by $s(\bar{x})$.

An alternative rewrite program, using conditional rules and no new symbols, would be

$$\begin{aligned} f(\bar{x}) \quad &\text{:-} \quad p(\bar{x}) \quad \rightarrow \quad r(\bar{x}) \\ f(\bar{x}) \quad &\text{:-} \quad \mathbf{not}(p(\bar{x})) \quad \rightarrow \quad s(\bar{x}), \end{aligned}$$

where **not** is negation. A straightforward example is the following program for computing the union of two sets of numbers (say) represented as lists (without repetition). That is, given two lists X and Y, it returns a list $Union(X,Y)$, containing those elements that appear in at least one of the input lists. The program is

List Union

$Union(\mathbf{nil},Y)$			$\rightarrow$	Y
$Union(X,\mathbf{nil})$			$\rightarrow$	X
$Union(x \cdot X,Y)$	:-	$member(x,Y)$	$\rightarrow$	$Union(X,Y)$
$Union(x \cdot X,Y)$	:-	$\mathbf{not}(member(x,Y))$	$\rightarrow$	$x \cdot (Union(X,Y))$
$member(x,\mathbf{nil})$			$\rightarrow$	**false**
$member(x,x \cdot Y)$				
$member(x,y \cdot Y)$	:-	$\mathbf{not}(x=y)$	$\rightarrow$	$member(x,Y)$
$i = j$	:-	$\mathbf{number?}(i,j)$	$\rightarrow$	$\mathbf{eq}(i,j)$
$\alpha = \alpha$				
$x \cdot X = y \cdot Y$			$\rightarrow$	$x=y \ \& \ X=Y$
$\mathbf{true} \ \& \ p$			$\rightarrow$	p
$p \ \& \ \mathbf{true}$			$\rightarrow$	p

where *Union* is the function being defined, *member* is an auxiliary predicate testing for membership of an element in a list, **eq** is a built-in predicate that tests for equality of numbers, and **number?** is a built-in predicate that returns **true** if all its argument are numbers (and **false** otherwise). Note that we must have the **false** case for *member*, since there is no "negation by failure".

3. LOGIC PROGRAMMING

Rewrite systems may be used as "logic programs" [Kowalski-74], in addition to their straightforward use for computation by rewriting, illustrated in the previous section. The programming paradigm described below allows for the advantageous combination of both computing modes. The result is a PROLOG-like programming language, the main differences being that rewrite rules are conditional equivalences, rather than implications in Horn-clause form, and that what is called "narrowing" is used in place of resolution.

The following, for example, is a program $div(a,b,q,r)$ to compute the quotient q and remainder r of nonnegative integer a and positive integer b:

Integer Division

$div(u,v,q+1,r)$	:-	$u \geq v$	$\rightarrow$	$div(u-v,v,q,r)$
$div(u,v,0,u)$	:-	$v > u$		
$div(u,v,0,r)$	:-	$u > v$	$\rightarrow$	**false**
$i > j$	:-	$\mathbf{number?}(i,j)$	$\rightarrow$	$\mathbf{greater}(i,j)$
$i \geq j$	:-	$\mathbf{number?}(i,j)$	$\rightarrow$	$\mathbf{not}(\mathbf{less}(j,i))$
$i - j$	:-	$\mathbf{number?}(i,j)$	$\rightarrow$	$\mathbf{diff}(i,j)$

The first rule is the recursive case; the second is the base case; the third covers false cases.

The *resolution procedure*, developed by J. A. Robinson [Robinson-63] in the early 1960's, derives consequences of logical formulas written in "clausal" form, and (a specialized "linear-input" version) is used to execute PROLOG programs. The *completion procedure*, developed by D. E. Knuth and P. Bendix [Knuth-Bendix-70] in the late 1960's, was introduced as a means of deriving canonical term-rewriting systems to serve as decision procedures for given unconditional equational theories. More recently, it has been applied to other aspects of equational reasoning (see, for example, [Dershowitz-82b]), and in [Dershowitz-85, Dershowitz-Josephson-84] it has been applied to logic programming, as well. Completion is, in a sense, an extension of resolution in that it allows unification at subterms. *Narrowing* [Slagle-74] is a "linear" restriction on completion, analogous to "linear resolution". To execute a rewrite program like the one shown above, we adapt the narrowing process to rules with conditions; we call this adaptation *conditional narrowing*. Since, in general, there may be many ways to achieve a subgoal, alternative narrowing computations must be attempted, either in parallel (until one succeeds) or sequentially (by backtracking upon failure).

A *rewrite program* is a set of rewrite rules of the form

$$l[\bar{x}] \quad \text{:-} \quad p[\bar{x},\bar{y}] \quad \rightarrow \quad r[\bar{x},\bar{y}],$$

where the condition *may* contain variables $\bar{y}$ not in the left-hand side. To compute with a logic program, a *goal rule* is added to a rewrite system. Goal rules are of the form

$$g[\bar{x},\bar{z}] \quad \rightarrow \quad answer(\bar{z}),$$

where g is the *calling term* containing input values (i.e. irreducible ground terms) $\bar{x}$ and output variables $\bar{z}$, and *answer* is the function symbol that will store the result. At each point in the computation the current subgoal is of the general form

$$h \quad \text{:-} \quad q_1, \ \cdots, q_n \quad \rightarrow \quad answer(\bar{s}),$$

meaning that the answer is $\bar{s}$ if the subgoals q_1, ..., q_n, and h are achieved (in that order). Given such a subgoal, and a rule

$$l \ :- \ p \ \rightarrow \ r$$

whose left-hand side l can be unified with a (nonvariable) subterm of q_1 via most general unifier σ, i.e. $q_1\sigma = t[l\sigma]$ for some context t, the subgoal is *narrowed* to

$$h\sigma \ :- \ p\sigma, \ t[r\sigma], \ q_2\sigma, \ \ldots, \ q_n\sigma \ \rightarrow \ answer(\bar{s}\sigma).$$

Furthermore, at each such step, all possible simplifications (as in the previous section) are applied throughout the subgoal, including

$$\mathbf{true}, x \ \rightarrow \ x.$$

That gives a new subgoal

$$h' \ :- \ q_1', \ q_2', \ \ldots, \ q_m' \ \rightarrow \ answer(\bar{s'}).$$

where h', q_1', etc. are irreducible. Only when all the conditions become **true**, and the subgoal is of the (unconditional) form

$$h' \ \rightarrow \ answer(\bar{s'}),$$

are unifications attempted within h'. Computation ends when a ***solution rule***

$$\mathbf{true} \ \rightarrow \ answer(\bar{t})$$

is generated, giving an answer $\bar{t}$ such that

$$g[\bar{x}, \bar{t}]$$

holds.

Note that conditions, when separated by commas, are executed from left-to-right, and must all be true before the left-hand side is replaced by the right-hand side. Conditions separated by the symbol $\&$, on the other hand, may be executed in any order. Such commas are just "syntactic sugar" in that a rule l:-$p,q\rightarrow r$ can always be replaced by two rules, l:-$p\rightarrow if(q,r)$ and $if(x,y)$:-$x\rightarrow y$, having only one condition each.

To compute the quotient and remainder of two numbers a and b with the above program, the rule

$$div(a,b,q,r) \ \rightarrow \ answer(q,r)$$

is added, meaning that q are r are the answer if and only if they are the quotient and remainder, respectively, of a and b. The interpreter then generates a rule

$$\mathbf{true} \ \rightarrow \ answer(c,d),$$

containing the answer values c and d for q and r, respectively. For example, to compute the quotient and remainder of 7 and 3, the rule

$$div(7,3,q,r) \ \rightarrow \ answer(q,r)$$

is added. Narrowing generates

$$div(7-3,3,z,r) \ \rightarrow \ answer(z+1,r),$$

by applying the first program rule, which simplifies to

$$div(4,3,z,r) \ \rightarrow \ answer(z+1,r),$$

applying the last rule for built-in substraction. Using the first rule again gives

$$div(1,3,z,r) \ \rightarrow \ answer(z+1+1,r).$$

Now the second rule yields the answer

$$\mathbf{true} \ \rightarrow \ answer(0+1+1,1).$$

Note that the above program can test whether ***or not*** two numbers have the given quotient and remainder. It is not, however, in a form that would allow computing the first argument, say, from the other three, unless the built-in **number?**(i,j) generates all instances of i and j that are numbers.

Any PROLOG statement may be directly translated into rewrite rules: the statement

$$A \ :- \ B,C,$$

corresponds to the identical rule. A rule of the form

$$A \ :- \ B \ \rightarrow \ C$$

is stronger than the above Horn clause and means that $B \supset (A \equiv C)$. A rule

$$A \ \rightarrow \ B \,\&\, C$$

is even stronger; it has A true *if and only if* B and C both hold. PROLOG goals

$$:- \ B,C$$

correspond to goal rules

$$B, \ C \ \rightarrow \ answer(\bar{x}),$$

where $\bar{x}$ are the variables in B and C.

The two paradigms of computation, viz. simplification and narrowing, can be combined in a single rewrite program. Every narrowing step is followed by as much simplification as possible. Simplification steps employ pattern matching, while narrowing involves unification. Simplifications are irrevocable; narrowing steps can be backtracked over.

For example, the functional *Union* program can be used in a logic program to find a list Z and element a such that

$$\{1\} \cup Z = \{a,1\} \cup \{2,3\}.$$

That goal can be expressed as

$$Union(1\cdot\mathbf{nil},Z) = Union(a\cdot 1\cdot\mathbf{nil},2\cdot 3\cdot\mathbf{nil}) \,\&\, \neg(a=1) \ \rightarrow \ answer(a,Z),$$

where the condition $\neg(a=1)$ is needed to ensure that the list $a\cdot 1\cdot\mathbf{nil}$ is a proper encoding of a set. The computation could then proceed as follows:

$$\mathit{Union}(1\cdot\mathbf{nil},Z) = \mathit{Union}(a\cdot 1\cdot\mathbf{nil},2\cdot 3\cdot\mathbf{nil})\ \&\ \neg(a{=}1) \quad\rightarrow\quad \mathit{answer}(a,Z)$$
$$Z = 2\cdot 3\cdot\mathbf{nil}\ \&\ \neg(a{=}1) \quad{:}{-}\quad \mathit{member}(a,2\cdot 3\cdot\mathbf{nil}) \quad\rightarrow\quad \mathit{answer}(a,Z)$$
$$Z = 2\cdot 3\cdot\mathbf{nil} \quad\rightarrow\quad \mathit{answer}(2,Z)$$
$$\mathbf{true} \quad\rightarrow\quad \mathit{answer}(2,2\cdot 3\cdot\mathbf{nil})$$

yielding one answer out of many. (Here, and in the sequel, we show computation sequences consisting of one narrowing step followed by full simplification.)

The *Union* program can also be extended with rules like

$$\begin{array}{lcl} \mathit{member}(x,\mathit{Union}(X,Y)) & {:}{-} & \mathit{member}(x,X) \\ \mathit{member}(x,\mathit{Union}(X,Y)) & {:}{-} & \mathit{member}(x,Y) \\ \neg\mathit{member}(x,\mathit{Union}(X,Y)) & \rightarrow & \neg\mathit{member}(x,X)\&\neg\mathit{member}(x,Y) \end{array}$$

to find, say, an X such that x either is or is not a member of $\mathit{Union}(X,Y)$, given x and Y.

As an example of the utility of applying rules at more than one level of a goal, consider the following situation:

List Generator				
$\mathit{listp}(\mathbf{nil})$				
$\mathit{listp}(x\cdot Y)$			$\rightarrow$	$\mathit{listp}(Y)$
$\mathit{size}(\mathbf{nil})$			$\rightarrow$	0
$\mathit{size}(x\cdot Y)$			$\rightarrow$	$\mathit{size}(Y)+1$
$i < j$	:-	$\mathbf{number?}(i,j)$	$\rightarrow$	$\mathbf{less}(i,j)$
$i+1 < j$	:-	$\mathbf{number?}(j)$	$\rightarrow$	$i < \mathbf{sub1}(j)$
$i < 0$			$\rightarrow$	$\mathbf{false}$
$p\&\mathbf{false}$			$\rightarrow$	$\mathbf{false}$

A subgoal $\mathit{listp}(Z)$ can, by repeating the second rule, generate arbitrarily large lists $Z{=}x_1\cdot x_2\cdot\cdots\cdot x_n\cdot\mathbf{nil}$. But when combined with a test for size, as in

$$\mathit{listp}(Z)\ \&\ \mathit{size}(Z) < 10,$$

it will *reduce* to **false** after ten iterations. At that point the second subgoal becomes **false**, thereby pruning a potentially infinite computation path.

The conditional part can be used for generalized assignment (subsuming **setq** in LISP and **is** in PROLOG), in the following manner:

Insertion Sort				
$\mathit{Sort}(\mathbf{nil})$			$\rightarrow$	$\mathbf{nil}$
$\mathit{Sort}(x\cdot\mathbf{nil})$			$\rightarrow$	$x\cdot\mathbf{nil}$
$\mathit{Sort}(x\cdot Y)$	:-	$Z \doteq \mathit{Sort}(Y)$	$\rightarrow$	$\mathit{Insert}(x,Z)$
$x\cdot Y \doteq x\cdot Y$				
$\mathit{Insert}(x,\mathbf{nil})$			$\rightarrow$	$x\cdot\mathbf{nil}$
$\mathit{Insert}(x,y\cdot Z)$	:-	$\mathbf{not}(\mathbf{greater}(x,y))$	$\rightarrow$	$x\cdot y\cdot Z$
$\mathit{Insert}(x,y\cdot Z)$	:-	$\mathbf{not}(\mathbf{less}(x,y))$	$\rightarrow$	$y\cdot\mathit{Insert}(x,Z)$

The purpose of the condition $Z{\doteq}\mathit{Sort}(Y)$ is to assign the *sorted* list to Z. Only when $\mathit{Sort}(Y)$ is partially evaluated to a list of the form $x\cdot Y$ can the rule for $\doteq$ be applied; the term $\mathit{Sort}(Y)$ itself cannot be assigned to Z, as would be the case were $=$ used. (In general, to accomplish the "read-only" function of ? in the language Concurrent Prolog [Shapiro-83], we would want to have built-in assignment rules of this form for each "constructor" function.)

REFERENCES

[Barbutti,*etal.*-85] Barbutti, R., Bellia, M., Levi, G., and Nartelli, M. "LEAF: A language which integrates logic, equations and functions". In: *Functional and Logic Programming*, D. deGroot and G. Lindstrom, eds. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Chester-80] Chester, D. "HCPRVR: An interpreter for logic programs". *Proceedings of the First Annual National Conference on Artificial Intelligence*, Stanford, CA (1980).

[Dershowitz-82b] Dershowitz, N. "Applications of the Knuth-Bendix completion procedure". *Proceedings of the Seminaire d'Informatique Theorique*, Paris, France (December 1982).

[Dershowitz-85] Dershowitz, N. "Computing with rewrite systems". *Information and Control* (1985) (to appear).

[Dershowitz-Josephson-84] Dershowitz, N., and Josephson, N. A. "Logic programming by completion". *Proceedings of the Second International Conference on Logic Programming*, Uppsala, Sweden (July 1984), pp. 313-320.

[Fribourg-84] Fribourg, L. "Oriented equational classes as a programming language". *Proceedings of the Eleventh EATCS Colloquium on Automata, Languages and Programming*, Antwerp, Belgium (July 1984).

[Goguen-Meseguer-84] Goguen, J. A., and Meseguer, J. "Equality, types, modules and generics for logic programming". *Logic Programming* (1984).

[Knuth-Bendix-70] Knuth, D. E., and Bendix, P. B. "Simple word problems in universal algebras". *Computational Problems in Abstract Algebra* (1970), pp. 263-297.

[Komorowski-82] Komorowski, H. J. "QLOG—The programming environment for PROLOG in

LISP". In: *Logic Programming*, Keith L. Clark and Sten- Åke Tärnlund, eds. Academic Press, 1982, pp. 315-322.

[Kowalski-74] Kowalski, R. A. "Predicate logic as programming language". *Proceedings of the IFIP Congress*, Amsterdam, The Netherlands (1974), pp. 569-574.

[Malachi-Manna-Waldinger-84] Malachi, Y., Manna, Z., and Waldinger, R. J. "TABLOG: The deductive tableau programming language". *Proceedings of the ACM Lisp and Functional Programming Conference* (1984), pp. 323-330.

[Reddy-85] Reddy, U. "Narrowing as the operational semantics of functional languages", Unpublished report, Department of Computer Science, University of Utah, Salt Lake City, UT, 1985 (submitted).

[Robinson-63] Robinson, J. "Theorem proving on the computer". *J. of the Association for Computing Machinery* (1963), pp. 163-174.

[Robinson-Sibert-82] Robinson, J. A., and Sibert, E. E. "LOGLISP: an alternative to PROLOG". In: *Machine Intelligence 10*, P. Hayes, D. Michie and Y-H. Pao, eds. Ellis Horwood, 1982, pp. 399-419.

[Shapiro-83] Shapiro, E. Y. "A subset of Concurrent Prolog and its interpreter", Institute for New Generation Computer Technology, Tokyo, February 1983.

[Slagle-74] Slagle, J. R. "Automated theorem-proving for theories with simplifiers, commutativity, and associativity". *J. of the Association for Computing Machinery*, Vol. 21, No. 4 (1974), pp. 622-642.