Higher-Order Conditional Term Rewriting in the L_λ Logic Programming Language

Preliminary Results

Amy Felty

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract

In this paper, we extend the notions of first-order conditional rewrite systems and higher-order rewrite systems to obtain higher-order conditional rewriting. Such rewrite systems can be used to directly express many operations in theorem proving and functional programming. We then illustrate that these rewrite systems can be naturally specified and implemented in a higher-order logic programming language. This paper was presented at the Third International Workshop on Extensions of Logic Programming, February 1992.

1 Introduction

Higher-order rewrite systems extend first-order rewrite systems and provide a mechanism for reasoning about equality in languages that include notions of bound variables [1, 9, 12, 5]. First-order conditional rewrite systems extend first-order rewrite systems, providing more expressive power by allowing conditions to be placed on rewrite rules [2, 8]. Such conditions must be satisfied before a particular rewrite can be applied. In this paper, we extend these two notions to define higher-order conditional rewriting. We extend first-order conditional rewriting to the higher-order case in a manner that can be viewed as analogous to the way that Nipkow [12] and Felty [5] extend first-order rewriting to the higher-order case. We use the simply typed λ-calculus as the language for expressing rules, with a restriction on the occurrences of free variables so that matching of terms with rewrite templates is decidable. Conditions will be expressed in a logic, called E_λ, which extends this restriction on free variables to variables bound by quantification.

We then show how such rewrite systems can be specified and implemented in a higher-order logic programming language whose logical foundation is L_λ [10], a variant of E_λ. This language replaces first-order terms in traditional languages such as Prolog with simply typed λ-terms, and first-order unification with a simple and decidable subcase of higher-order unification, called $\beta_0\eta$-unification. The rules of a higher-order rewrite system can be directly specified

in this language, and unification is directly available for matching terms with rewrite templates. Our extended language also permits queries and the bodies of clauses to be both implications and universally quantified. These operations are essential for applying rewrite and congruence rules to descend through terms in order to apply rewrite rules to subterms. The programs presented here have been tested in the logic programming language λProlog which is more general than the language L_λ used in this paper.

In Section 2 we define L_λ and E_λ, the metalanguages for logic programming and for expressing rewriting, respectively. In Section 3, we define conditional higher-order rewrite systems using this metalanguage. In Section 4 we describe an interpreter for the logic programming language, and in Section 5 we illustrate by example how rewrite systems can be specified in this language. Finally, Section 6 concludes.

2 A Metalanguage for Rewriting

The terms of the metalanguage are the simply typed λ-terms. We present the notation used here and some basic properties. See Hindley and Seldin [7] for a fuller discussion. We assume a fixed set of *primitive types*. The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of function types, denoted by the binary, infix symbol $\rightarrow$. The Greek letter τ is used as a syntactic variable ranging over types. The type constructor $\rightarrow$ associates to the right.

For each type τ, we assume that there are denumerably many constants and variables of that type. Constants and variables do not overlap and if two constants (variables) have different types, they are different constants (variables). To make the type τ of constant a explicit, we often write $a{:}\tau$. We often speak of a fixed *signature* or a finite set of constants and variables, usually denoted Σ. Simply typed λ-terms are built in the usual way using constants, variables, applications, and abstractions. If M is a term and $x_1, \ldots, x_n$ are distinct variables, we often write $\lambda\overline{x_n}.M$ for $\lambda x_1 \ldots \lambda x_n.M$ and $M\overline{x_n}$ for $Mx_1 \ldots x_n$. For a term M of type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $n \geq 0$ and τ_0 is primitive, we say that n is the *arity* of M. In a term of the form $hM_1 \ldots M_n$ where $n \geq 0$ and h is a constant or variable, we say that h is the *head* of this term.

If x is a variable and M is a term of the same type then $[M/x]$ denotes the operation of substituting M for all free occurrences of x, systematically changing bound variables in order to avoid variable capture. We shall assume that the reader is familiar with the usual notions and properties of substitution and α, β, and η conversion for the simply typed λ-calculus. Here, equality between λ-terms is taken to mean $\beta\eta$-convertible. When we write a term, it actually represents an equivalence class of terms.

A term is called a *higher-order pattern* (or simply *pattern*) if every occurrence of a free variable h appears in a subterm of the form $hx_1 \ldots x_n$ where $n \geq 0$ and $x_1, \ldots, x_n$ are distinct bound variables. In Miller [10], it is shown that

unification of patterns, called $\beta_0\eta$-unification, is decidable and that for any two unifiable patterns, a most general unifier can be computed.

To define L_λ formulas, we extend the notion of terms. We assume o is a member of the set of primitive types. A *predicate* p is a constant of type $\tau_1 \to \cdots \to \tau_n \to o$ where $\tau_1, \ldots, \tau_n$ do not contain o. The logical constants are given the following types: $\wedge$ (conjunction) and $\supset$ (implication) are both of type $o \to o \to o$; and $\forall_\tau$ (universal quantification) is of type $(\tau \to o) \to o$, for all types τ not containing o. The logical constants $\wedge$ and $\supset$ are written in the familiar infix form. A formula is either *atomic* or *non-atomic*. An atomic formula is of the form $(pt_1 \ldots t_n)$, where p is a predicate and $t_1, \ldots, t_n$ are terms of the types $\tau_1, \ldots, \tau_n$, respectively. *Non-atomic formulas* are of the form $B_1 \wedge B_2$, $B_1 \supset B_2$, or $\forall_\tau(\lambda x\ B)$, where B, B_1, and B_2 are formulas and τ is a type not containing o. The expression $\forall_\tau(\lambda x\ B)$ is written $\forall_\tau x\ B$ or simply as $\forall x\ B$ when types can be inferred from context. The formula $\forall x_1 \ldots \forall x_n\ B$, $n \geq 0$ is also written $\forall \overline{x_n} B$.

We define two sets of L_λ formulas, called $\mathcal{D}$ and $\mathcal{G}$, by placing restrictions on variables bound by quantification. These restrictions are similar to the one above on variables bound by λ-abstraction in patterns. A variable occurrence z in a formula is said to be *positive* (*negative*) if it is bound by a positive (negative) occurrence of a universal quantifier. A variable occurrence is *λ-bound* if it is bound by a λ-abstraction. A formula is in $\mathcal{D}$ (respectively $\mathcal{G}$) if every positive (respectively negative) variable occurrence z appears in a subterm of the form $zx_1 \ldots x_n$ where $n \geq 0$ and $x_1, \ldots, x_n$ are distinct either negative (respectively positive) variable occurrences or λ-bound variable occurrences bound within the scope of the binding for z. We call this restriction on z the *head restriction*. In addition, we require that formulas in $\mathcal{D}$ and $\mathcal{G}$ are closed. A formula in $\mathcal{D}$ or $\mathcal{G}$ is called, respectively, a D-formula or a G-formula.

Provability for L_λ can be given in terms of sequent calculus proofs. A *sequent* is a pair $\Gamma \longrightarrow G$, where G is a G-formula, and Γ is a finite (possibly empty) sets of D-formulas. The set Γ is this sequent's *antecedent* and G is its *succedent*. The expression B, Γ denotes the set $\Gamma \cup \{B\}$; this notation is used even if $B \in \Gamma$. The inference rules for sequents are presented in Figure 1. The following provisos are also attached to the two inference rules for quantifier introduction: in $\forall$-R c is a constant of type τ not occurring free in the lower sequent, and in $\forall$-L t is a term of type τ.

A *proof* of the sequent $\Gamma \longrightarrow G$ is a finite tree constructed using these inference rules such that the root is labeled with $\Gamma \longrightarrow G$ and the leaves are labeled with *initial sequents*, that is, sequents $\Gamma' \longrightarrow G'$ such that $G' \in \Gamma'$. The non-terminals in such a tree are instances of the inference figures in Figure 1. Since we do not have an inference figure for $\beta\eta$-conversion, we shall assume that in building a proof, two formulas are equal if they are $\beta\eta$-convertible.

We extend L_λ with equality, reducibility, join, and redex relations at primitive types by introducing four constants $=_\tau$, $\stackrel{*}{\to}_\tau$, $\downarrow^*_\tau$, and $\to_\tau$, respectively, of type $\tau \to \tau \to o$ for every primitive type τ except o. Subscripts will be omitted when type information is not important or can be inferred from context. This

$$\frac{B, C, \Gamma \longrightarrow G}{B \wedge C, \Gamma \longrightarrow G} \wedge\text{-L} \qquad \frac{\Gamma \longrightarrow B \qquad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C} \wedge\text{-R}$$

$$\frac{\Gamma \longrightarrow B \qquad C, \Gamma \longrightarrow G}{B \supset C, \Gamma \longrightarrow G} \supset\text{-L} \qquad \frac{B, \Gamma \longrightarrow C}{\Gamma \longrightarrow B \supset C} \supset\text{-R}$$

$$\frac{[t/x]B, \Gamma \longrightarrow G}{\forall_\tau x\ B, \Gamma \longrightarrow G} \forall\text{-L} \qquad \frac{\Gamma \longrightarrow [c/x]B}{\Gamma \longrightarrow \forall_\tau x\ B} \forall\text{-R}$$

Fig. 1. Left and right introduction rules for L_λ

language will be called E_λ, and has the additional inference rules in Figure 2. The first four rules express reflexivity, symmetry, transitivity, and congruence of equality. In the CONG rule, h is a variable or constant of arity n, and for $i = 1, \ldots, n$, M_i and N_i are terms of arity m_i. Also, the universally quantified variables in the premises must not occur free in the conclusion, and must be of the appropriate type for the terms in the premises to be well-formed. In addi-

$$\Gamma \longrightarrow M = M \quad \text{REFL} \qquad \frac{\Gamma \longrightarrow M = N}{\Gamma \longrightarrow N = M} \text{SYM}$$

$$\frac{\Gamma \longrightarrow M = P \qquad \Gamma \longrightarrow P = N}{\Gamma \longrightarrow M = N} \text{TRANS}$$

$$\frac{\Gamma \longrightarrow \forall \overline{x_{m_1}}(M_1 \overline{x_{m_1}} = N_1 \overline{x_{m_1}}) \quad \cdots \quad \Gamma \longrightarrow \forall \overline{x_{m_n}}(M_n \overline{x_{m_n}} = N_n \overline{x_{m_n}})}{\Gamma \longrightarrow hM_1 \ldots M_n = hN_1 \ldots N_n} \text{CONG}$$

$$\frac{\Gamma \longrightarrow M \xrightarrow{*} P \qquad \Gamma \longrightarrow N \xrightarrow{*} P}{\Gamma \longrightarrow M \downarrow^* N} \text{JOIN}$$

Fig. 2. Rules for equality in E_λ

tion, E_λ has corresponding REFL, TRANS, and CONG rules for the $\xrightarrow{*}$ relation. Finally, the last rule expresses the meaning of the join relation. There are no rules for the $\rightarrow$ relation. It will be used to express rewrite rules.

We define a set of formulas called $\mathcal{G}'$ that, unlike Γ and $\mathcal{G}$ may contain free variables. A formula is in $\mathcal{G}'$ if all its constants are either logical connectives or one of the equality, reducibility, join, or redex predicates. In addition, we

place a head restriction on negative variable occurrences similar to that on $\mathcal{G}$ but we also extend it to free variables. In particular, every negative variable occurrence or free variable occurrence z must appear in a subterm of the form $zx_1 \ldots x_n$ where $n \geq 0$ and $x_1, \ldots, x_n$ are distinct positive variable occurrences or λ-bound occurrences bound within the scope of the binding for z. A formula in $\mathcal{G}'$ is called a G-condition. Such formulas will be used to express conditions on rewrite rules.

3 Higher-Order Conditional Rewrite Systems

A *conditional equation* is defined to be a triple $G \Rightarrow l = r$ such that G is a G-condition whose atomic formulas contain only the equality predicate, l and r are patterns having the same primitive type, l is not a free variable, and all free variables in r also occur in l or G. We say that an occurrence of a free variable in a G-condition is in *reduced-term position* if it occurs in an atomic formula on the right hand side of a binary relation and the atomic formula is on the right hand side of an even number of implications. A *conditional rewrite rule* is defined to be a triple $G \Rightarrow l \rightarrow r$ such that all the atomic formulas of the G-condition G contain only reducibility, join, and redex predicates, l and r are patterns having the same primitive type, l is not a free variable, and all free variables in r also occur in l or have occurrences in reduced-term position in G. A *Higher-Order Conditional Rewrite System* (HCRS) is a finite set of conditional rewrite rules. In Nipkow [12] and Felty [5], higher-order rewrite rules without conditions are defined using patterns on the left hand side and arbitrary λ-terms on the right. The specification of such rewrite systems in a metalanguage slightly more general than L_λ is discussed in [5]. This notion of higher-order rewriting extends the usual notion of first-order term rewriting. Conditional rewrite rules as defined here extend first-order conditional rewrite rules (as defined in Dershowitz et. al. [3], for example) in an analogous way. In first-order rewrite rules, the condition G is often defined to be a conjunction of atomic formulas using equality or one of the reducibility predicates. Thus, our definition extends the definition by allowing an arbitrary formula from $\mathcal{G}'$.

For higher-order rewrite rules without conditions, the fact that unification of patterns is decidable guarantees that the rewrite relation is decidable. Conditional rewriting, even in the first-order case is more complicated and not always decidable, and thus will not be decidable in our case either. Note that we do, however, retain the property that it is decidable whether a given term matches a left hand side of a rewrite rule.

In writing rewrite rules, we adopt the convention that tokens beginning with upper case initial letters are free variables. Tokens that begin with lower case letters other than those bound by λ are constants.

To illustrate, we consider an example which expresses evaluation in a simple functional programming language consisting of primitive datatypes for booleans and natural numbers, a conditional statement, constructs for lists, function

abstraction, application, a fix point operator, and the *let* operator as in ML. We introduce a primitive type tm for terms of this functional language and introduce the constants shown with their types in Figure 3 to represent the constructs of the language. We will write $-$ and $<$ as infix operators. Clearly not all terms of

$$
\begin{array}{rl}
true : tm & \qquad nil : tm \\
false : tm & \qquad cons : tm \to tm \to tm \\
if : tm \to tm \to tm \to tm & \qquad hd : tm \to tm \\
0 : tm & \qquad tl : tm \to tm \\
s : tm \to tm & \qquad empty : tm \to tm \\
< : tm \to tm \to tm & \qquad app : tm \to tm \to tm \\
- : tm \to tm \to tm & \qquad abs : (tm \to tm) \to tm \\
gcd : tm \to tm \to tm & \qquad let : (tm \to tm) \to tm \to tm \\
 & \qquad fix : (tm \to tm) \to tm
\end{array}
$$

Fig. 3. Constants for Representing Functional Programs

type tm correspond to valid programs. Some form of type checking is needed. We only discuss evaluation here and assume terms to be evaluated correspond to valid programs. The rewrite rules expressing evaluation are given in Figure 4. Evaluation in the first-order fragment of this language is given by all but the last

$$
\begin{array}{rl}
 & M < M \to false \\
 & M < (s\ M) \to true \\
 & (s\ M) < M \to false \\
 & (s\ M) < (s\ N) \to M < N \\
 & (s\ M) - (s\ N) \to M - N \\
 & 0 - M \to 0 \\
 & M - 0 \to M \\
(N < M \xrightarrow{*} true) \Rightarrow & (gcd\ M\ N \to gcd\ (M - N)\ N) \\
(M < N \xrightarrow{*} true) \Rightarrow & (gcd\ M\ N \to gcd\ M\ (N - M)) \\
 & gcd\ M\ M \to M \\
 & if\ true\ M\ N \to M \\
 & if\ false\ M\ N \to N \\
 & hd\ (cons\ M\ N) \to M \\
 & tl\ (cons\ M\ N) \to N \\
 & empty\ nil\ \to true \\
 & empty\ (cons\ M\ N) \to false \\
 & abs\ \lambda x.(app\ M\ x) \to M \\
\forall x.((x \to N) \supset (Mx \xrightarrow{*} P)) \Rightarrow & (app\ (abs\ M)\ N \to P) \\
\forall x.((x \to N) \supset (Mx \xrightarrow{*} P)) \Rightarrow & (let\ M\ N \to P) \\
\forall x.((x \to (fix\ M)) \supset (Mx \xrightarrow{*} P)) \Rightarrow & (fix\ M \to P)
\end{array}
$$

Fig. 4. Rewrite Rules Expressing Evaluation in a Simple Functional Language

four rules. These rules are straightforward and it is easy to see that they satisfy the necessary constraints. The left and right hand sides are patterns since none

of the free variables are applied to any arguments, and the two conditions are G-conditions.

Now consider the last four rules. The two constants *app* and *abs* are used to code function application and abstraction. The first rule specifies η-reduction of λ-terms. On the left hand side, the bound variable x will not occur in instances of M as is required by the η-rule: any instance of M containing x would cause the variable x in the above rule to be renamed to avoid variable capture. The second rule specifies β-conversion. A term of the form $(app\ (abs\ M)\ N)$ is a β-redex whose reduced form is P as long as the condition on the left is satisfied. This condition states that for an arbitrary x, under the addition of the rewrite rule that rewrites x to N, it must be the case that Mx reduces to P. Note that instances of P cannot contain free occurrences of the variable x bound by universal quantification for the same reason as stated above for M. Thus all occurrences of x in Mx must be rewritten to N in order for this condition to succeed. The *let* construct corresponds to a *let* statement in ML. In a term of the form $(let\ M\ N)$, it is intended that the bound variable at the head of M will be assigned the value N in the body. In other words, this term is an abbreviation for the application MN. This reduction is expressed by the third rule above and is similar to the rule for β-reduction. The rule which expresses the unfolding of a fixpoint operator is also similar, but here x rewrites to $(fix\ M)$.

It is easy to see that the left and right hand sides of the above four rules are patterns, since there are no arguments to any of the free variables. In the conditions, the free variable M is applied to x which is bound by a positive occurrence of a universal quantifier. In the latter three rules, although P occurs on the right but not on the left of the rewrite rule, it occurs in reduced-term postition in the condition. Thus these conditions are G-conditions, and all four rules are valid conditional rewrite rules.

Note that as a rewrite system, these rules express non-deterministic evaluation. Nothing about order of evaluation in specified. In Felty [5], we show how to implement various rewriting strategies in the λProlog logic programming language. Such strategies, when given these rewrite rules as a parameter, correspond to various strategies for evaluating functional programs.

4 An Interpreter for L_λ

In the next section, we will talk about specifying higher-order conditional rewrite systems in L_λ. We will discuss the operational reading of these specifications with respect to a logic programming interpreter for L_λ and provide some insight into implementation. We provide a high-level description of this interpreter here.

A *definite clause* is a D-formula of L_λ, and a *program* is a set of definite clauses. A *goal* is a G-formula. From properties about L_λ presented in Miller [10], a sound and complete (with respect to intuitionistic logic) *non-deterministic* interpreter can be described by the following *search operations*. Here, the inter-

preter is attempting to determine if the goal formula G follows from the program Γ.

AND: If G is $G_1 \wedge G_2$ then try to show that both G_1 and G_2 follow from Γ.

AUGMENT: If G is $D \supset G'$ then add D to the current program and try to show G'.

GENERIC: If G is $\forall_\tau x G'$ then pick a new constant c of type τ and try to show $[c/x]G'$.

BACKCHAIN: If G is atomic, we consider the current program. If there is a universal instance of a definite clause which is convertible to G then we are done. If there is a definite clause with a universal instance of the form $G' \supset G$ then try to show G' follows from Γ. If neither case holds then G does not follow from Γ.

An implementation of an interpreter must make many choices which are left unspecified in the high-level description above. We discuss a few of the choices made by the logic programming language λProlog, which contains an implementation of L_λ.

First, the order in which conjuncts and disjuncts are attempted and the order for backchaining over definite clauses is determined exactly as in conventional Prolog systems: conjuncts and disjuncts are attempted in the order they are presented. Definite clauses are backchained over in the order they are listed in Γ using a depth-first search paradigm to handle failures. Logic variables as in Prolog are used in forming a universal instance in the BACKCHAIN operation. These variables can later be instantiated through unification. In this case, it is β_0-unification that is required.

The presence of logical variables in an implementation also requires that GENERIC be implemented slightly differently than is described above. In particular, if the goal $\forall_\tau x G'$ contains logical variables, the new constant c must not appear in the terms eventually instantiated for the logical variables which appear in G' or in the current program. Any implementation must take this constraint into account.

5 Specifying Rewrite Systems

In this section, we discuss the specification of higher-order rewrite systems in L_λ. Unlike E_λ, L_λ does not have the equality, reducibility, join, and redex relations or the inference rules for them as primitives. Here, we introduce constants for these relations and provide program clauses to specify the inference rules.

We will assume that all terms from a given object language contain only constants from a fixed signature, say Σ, which at least includes all of the constants in the rewrite rules. As an example, we take as a signature the set of constants declared in Figure 3 and illustrate the specification of the rewrite system for evaluation given in Figure 4.

To specify rewriting at a particular primitive type τ, we introduce the infix relations $=_\tau$, $\stackrel{*}{\rightarrow}_\tau$, $\downarrow^*_\tau$, and $\rightarrow_\tau$ to serve as predicates of type $\tau \rightarrow \tau \rightarrow o$. Our specification will be a set of definite clauses from which we can attempt to prove goals representing rewriting queries. The specification of rewrite rules as clauses is straightforward: we replace the rewriting relations of E_λ with the new rewriting predicates, we replace $\Rightarrow$ with $\supset$, and we take the universal closure over the free variables of the rewrite rule, including those in the condition. To illustrate, the three clauses below specify a first-order rule without and with a condition, and a higher-order rule with a condition. All subscripts on the reducibility and redex relation should be tm. We omit them for readability.

$$\forall M \forall N\, ((s\ M) < (s\ N) \rightarrow M < N)$$
$$\forall M \forall N ((N < M \stackrel{*}{\rightarrow} true) \supset (gcd\ M\ N \rightarrow gcd\ (M - N)\ N))$$
$$\forall N \forall M \forall P (\forall x.((x \rightarrow N) \supset (Mx \stackrel{*}{\rightarrow} P)) \supset (app\ (abs\ M)\ N \rightarrow P))$$

Note that we do in fact obtain D-formulas by simply taking the universal closure at the top level. Any free variable occurrence in the rewrite rule becomes a positive variable occurrence in the closure. In the condition, both negative and free variable occurrences become positive variable occurrences in the closure. As a formula in $\mathcal{G}'$, it was the negative and free variables that had to satisfy the head restriction, while in a D-formula, it is the positive occurrences that must satisfy this restriction.

Generally, in executing rewrite goals, we will often have a closed term on the left of the arrow and a variable on the right to be instantiated with the result of the rewrite. In using the first clause for example, M and N will be replaced with logic variables which get instantiated by matching the left hand side of the query with the pattern $(s\ M) < (s\ N)$. When the second clause is used in backchaining, we will then have to solve the subgoal specifying the condition. Note that if M and N are instantiated by the backchain operation, the terms on both the left and right of this subgoal will be instantiated. Backchaining on the third clause will provide instances of M and N. The subgoal that must be proved is slightly more complex. First, a GENERIC search operation will be applied to generate a new constant, say c, for x. Then the AUGMENT operation will add a clause stating that this constant c rewrites to the given instance of N. Then, in this new context, it must be shown that Mc rewrites to some term instantiating P. By the restriction on the GENERIC operation, this term cannot contain c, thus the rewrite rule for c must be applied for every instance of c, replacing each one by N.

For readability, in the remainder of this and the next sections, we will often leave off outermost universal quantification, and assume universal closure over all variables written as tokens with initial upper case letters.

To specify congruence, we introduce a D-formula for each constant in Σ. These D-formulas have the same form as those for rewrite rules with conditions. For example, the following two formulas are included for the *app* and *abs*

constants.

$$(M \stackrel{*}{\rightarrow} P) \wedge (N \stackrel{*}{\rightarrow} Q) \supset (app\ M\ N \rightarrow app\ P\ Q)$$
$$\forall x((x \rightarrow x) \supset (Mx \stackrel{*}{\rightarrow} Nx)) \supset (abs\ M \rightarrow abs\ N)$$

The clause for *abs* states that an abstraction $(abs\ M)$ rewrites to $(abs\ N)$ if for arbitrary x such that x rewrites to itself, Mx reduces to Nx. Operationally, in trying to solve a goal of the form $(abs\ M' \rightarrow abs\ N')$ where, say, M' is a closed term and N' a logic variable, we can use this clause to descend through the abstraction in M'. The GENERIC operation will generate a new meta-level signature item, say c, and the AUGMENT operation will add the atomic formula $(c \rightarrow c)$. This can be viewed as the dynamic addition of a new constant to the object-level signature and a reflexive rule for it. Then, β-reduction at the meta-level of $M'c$ performs the substitution of the new item c for the outermost bound variable in M'. In effect, the new signature item plays the role of the name of the object level bound variable. The atomic clause $(c \rightarrow c)$ can be used during the search for a term $N'c$ that is reachable by some number of rewrite steps from $M'c$. N' is the abstraction not containing c.

A congruence rule for a new constant of functional type is more complex. For example, if we had a function constant h whose type is $((tm \rightarrow tm) \rightarrow tm) \rightarrow tm$, its corresponding congruence clause would be:

$$\forall f(\forall P \forall Q((P \stackrel{*}{\rightarrow} Q) \supset (fP \rightarrow fQ)) \supset (Mf \stackrel{*}{\rightarrow} Nf)) \supset (h\ M \rightarrow h\ N)$$

Operationally, after backchaining on the above clause, instead of an atomic clause, the clause $(P \stackrel{*}{\rightarrow} Q) \supset (fP \rightarrow fQ)$ would be dynamically added by AUGMENT, serving as new congruence rule for the new function constant f.

We can in fact define a general function for specifying congruence rules for a particular signature. For signature item a of type τ, $[\![a; a : \tau]\!]^-$ yields the necessary congruence rule.

$$[\![M; N : \tau]\!]^- = \begin{cases} M \rightarrow_\tau N & \text{if } \tau \text{ is a primitive type} \\ \forall x \forall y([\![x; y : \tau_1]\!]^+ \supset [\![Mx; Ny : \tau_2]\!]^-) & \text{if } \tau \text{ is } \tau_1 \rightarrow \tau_2 \end{cases}$$

$$[\![M; N : \tau]\!]^+ = \begin{cases} M \stackrel{*}{\rightarrow}_\tau N & \text{if } \tau \text{ is a primitive type} \\ \forall x([\![x; x : \tau_1]\!]^- \supset [\![Mx; Nx : \tau_2]\!]^+) & \text{if } \tau \text{ is } \tau_1 \rightarrow \tau_2 \end{cases}$$

These functions are a (corrected) version of those used by Miller [11] to specify equality and substitution for simply typed λ-terms and are similar to those used by Felty [4] to code a dependent typed λ-calculus in a higher-order intuitionistic logic.

The remaining rules of Figure 2 are specified in a straightforward manner, by including the following clauses at each primitive type.

$$(M \stackrel{*}{\rightarrow} P) \wedge (P \stackrel{*}{\rightarrow} N) \supset (M \stackrel{*}{\rightarrow} N)$$
$$(M \downarrow^* P) \wedge (N \downarrow^* P) \supset (M \stackrel{*}{\rightarrow} N)$$
$$(M \stackrel{*}{\rightarrow} N) \supset (M = N)$$
$$(M \stackrel{*}{\rightarrow} N) \supset (N = M)$$

Note that we do not include a clause specifying reflexivity. We do not need one when we include "reflexivity" clauses $(a \rightarrow a)$ for each constant a of primitive type in the signature. To rewrite a term to itself, congruence rules must be used to descend through the entire term, applying reflexivity rules for constants at the leaves. For efficiency reasons, we may want to include a reflexive rule $\forall M(M \stackrel{*}{\rightarrow} M)$. It can be used to prove the equivalence of two arbitrary terms of primitive type in a single step. An advantage of the former approach is that it can also be used to verify that a term is well-formed in a particular signature.

6 Some Concluding Remarks

As stated earlier, the formulation of rewrite rules in Nipkow [12] and Felty [5] is slightly different than that given here. There are no conditions and right hand sides are not restricted to be patterns. If terms are always matched against left hand sides, decidability of the rewrite relation is not affected. In this setting, the last three rules in Figure 4 can be expressed more directly, and perhaps more naturally, as follows.

$$\begin{aligned} app\ (abs\ M)\ N &\rightarrow MN \\ fix\ M &\rightarrow M\ (fix\ M) \\ let\ M\ N &\rightarrow MN \end{aligned}$$

In applying one of these rules, instead of having to satisfy a condition which adds a rewrite rule for some new constant c, and must then reduce the term Mc in a new context, application of λ-terms is used to directly substitute the appropriate term for the bound variable in M. In that setting, a logic programming language with unification more powerful than $\beta_0\eta$-unification is needed, such as λProlog.

Any rewrite system expressible in the more general setting can in fact be expressed as conditional rewrite rules as defined here. A function can be defined to translate rules with arbitrary λ-terms on the right to conditional rules with patterns on the right. Such a function is defined by induction over types in a manner similar to the one in the previous section for generating congruence clauses.

Within theorem proving systems, capabilities for higher-order rewriting can provide a useful tool for the manipulation of formulas and programs. In [5], we illustrate how to integrate a general component for higher-order term rewriting into a tactic style theorem prover. The implementation discussed there builds on the logic programming implementation of tactic style theorem provers presented in Felty [6], and provides a setting for implementing both general and specific rewriting strategies. These techniques can be easily extended to conditional rewriting.

References

1. Peter Aczel. A general church-rosser theorem. Technical report, University of Manchester, 1978.

2. J. A. Bergstra and J. W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32:323–362, 1986.

3. N. Dershowitz, M. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, pages 31–44. Springer-Verlag Lecture Notes in Computer Science, 1987.

4. Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 215–251. Cambridge University Press, 1991.

5. Amy Felty. A logic programming approach to implementing higher-order term rewriting. In Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, *Proceedings of the January 1991 Workshop on Extensions to Logic Programming*, pages 135–161. Springer-Verlag Lecture Notes in Artificial Intelligence, 1992.

6. Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, To appear.

7. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

8. Stéphane Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33:175–193, 1984.

9. J. W. Klop. Combinatory reduction systems. Technical Report Mathematical Centre Tracts Nr.127, Centre for Mathematics and Computer Science, Amsterdam, 1980.

10. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

11. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, Paris, France, June 1991. MIT Press.

12. Tobias Nipkow. Higher-order critical pairs. In *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349, Amsterdam, July 1991.