A Logic Programming Approach to Implementing Higher-Order Term Rewriting *

Amy Felty

INRIA Rocquencourt
Domaine de Voluceau
78153 Le Chesnay Cedex, France

Abstract

Term rewriting has proven to be an important technique in theorem proving. In this paper, we illustrate that rewrite systems and strategies for higher-order term rewriting, which includes the usual notion of first-order rewriting, can be naturally specified and implemented in a higher-order logic programming language. We adopt a notion of higher-order rewrite system which uses the simply typed λ-calculus as the language for expressing rules, with a restriction on the occurrences of free variables on the left hand sides of rules so that matching of terms with rewrite templates is decidable. The logic programming language contains an implementation of the simply-typed lambda calculus including $\beta\eta$-conversion and higher-order unification. In addition, universal quantification in queries and the bodies of clauses is permitted. For higher-order rewriting, we show how these operations implemented at the meta-level provide elegant mechanisms for the object-level operations of descending through terms and matching terms with rewrite templates. We discuss tactic style theorem proving in this environment and illustrate how term rewriting strategies can be expressed as tactic-style search.

1 Introduction

Much effort has gone into the study of first-order rewrite systems and as a result, there is currently a large body of knowledge about their properties. Implementors of theorem proving systems have been able to exploit this knowledge to implement effective strategies for reasoning about equality between first-order terms. More recently, the study of rewrite systems has included the more expressive higher-order rewrite systems. One direction involves extending early work by Aczel [1] and Klop [16] which uses λ-terms as a meta-language for expressing rewrite systems for object languages that include notions of bound variables. Such λ-terms can be used to elegantly express the *higher-order abstract*

* This paper appears in *Proceedings of the 1991 International Workshop on Extensions of Logic Programming*, Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, Springer-Verlag Lecture Notes in Artificial Intelligence, 1992.

syntax of these object languages [27, 18]. For example, the abstractions built into λ-terms can be used to represent quantification in formulas or abstraction in functional programs. Using this representation, many operations on formulas and programs can be naturally expressed as higher-order rewrite systems. Within theorem proving systems, capabilities for higher-order rewriting can thus provide a useful tool for the manipulation of formulas and programs.

In this paper, we adopt the definition of higher-order rewrite system given by Nipkow [23], though we give a different presentation of the notions of rewrite relation and of equality modulo a rewrite system. We then show how such rewrite systems as well as strategies for rewriting can be specified and implemented in a higher-order logic programming language.

The higher-order logic programming language used here is based on *higher-order hereditary Harrop formulas* [21]. This language replaces first-order terms in traditional languages such as Prolog with simply typed λ-terms, and first-order unification with higher-order unification. The rules of a higher-order rewrite system can be directly specified in this language, and the operation of higher-order 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. We shall show how these operations are essential for applying congruence rules to descend through terms in order to apply rewrite rules to subterms.

In Section 2 we define higher-order rewrite systems, and in Section 3 we present several examples. In Section 4 we describe the meta-logic and logic programming language, and in Section 5 we illustrate by example how rewrite systems can be specified in this language. Sections 6 and 7 illustrate how to integrate a general component for higher-order term rewriting into a tactic style theorem prover. The implementation discussed here builds on the logic programming implementation of tactic style theorem provers presented in Felty and Miller [8] and Felty [6]. Section 7 discusses how the operation of higher-order unification which is available in our logic programming language can be used to directly implement tactics for first-order term rewriting. We discuss both the power and limitations of this implementation technique. Finally, Section 8 concludes and discusses related work. Appendix A contains a proof of equivalence between our operational definition of rewriting and the corresponding inference system for equality modulo a rewrite system.

2 Higher-Order Rewrite Systems

As stated, the meta-language used to define higher-order rewrite systems is the simply typed λ-calculus. We present the notation used here and some basic properties. See Hindley and Seldin [13] 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 $a{:}\tau$. 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 $Mx_1 \ldots x_n$. In the former term, we say that $\lambda\overline{x_n}$ is its *binder* and M its *body*. 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.

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. The expression $[M_1/x_1, \ldots, M_n/x_n]$ will denote the simultaneous substitution of the terms $M_1, \ldots, M_n$ for distinct variables $x_1, \ldots, x_n$, respectively. We use Greek letter σ to denote substitutions, and write σM for the application of substitution σ to term M.

The relation of convertibility up to α, β, η is written as $=_{\beta\eta}$. A λ-term is in *$\beta\eta$-long form* if it is of the form $\lambda\overline{x_n}.hM_1 \ldots M_m$ where $n, m \geq 0$, h is either a constant or a variable of arity m, and terms $M_1, \ldots, M_m$ are also in $\beta\eta$-long form. We call h the *head* of this term. All λ-terms $\beta\eta$-convert to a term in $\beta\eta$-long form, unique up to α-conversion. 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. It will often be convenient to consider the $\beta\eta$-long form as the canonical representative of its equivalence class.

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. This subclass of simply typed λ-terms is defined in Miller [19] where it is also proved that unification of patterns is decidable and for any two unifiable patterns, a most general unifier can be computed.

A *rewrite rule* is defined to be a pair $l \rightarrow r$ such that l and r are λ-terms of the same primitive type, l is a pattern, but not a free variable, and all free variables in r also occur in l. This meta-language and the restriction to patterns on the left hand side in defining rewrite rules is, in fact, similar to that used by Klop [17] in defining Combinatory Reduction Systems and the same as that used by Nipkow [23]. By imposing this restriction on left hand sides, we guarantee that the rewrite relation will always be decidable. 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.

Definition 1 A *Higher-Order Rewrite System* (HRS) is a finite set of rewrite rules. Given an HRS H, a relation $\rightarrow_H$ on terms can be defined as follows: $M \rightarrow_H N$ if there are terms u, L, R such that $M =_{\beta\eta} uL$, $N =_{\beta\eta} uR$, and u, L, R are obtained as follows.

1. There is a rule $l \rightarrow r$ in H.
2. $x_1, \ldots, x_n$ are variables not occurring free in M, N, l, r.
3. σ is a substitution whose domain is the free variables of l and whose range consists of terms whose only free variables are $x_1, \ldots, x_n$.
4. L is the term with binder $\lambda\overline{x_n}$ and body σl.
5. R is the term with binder $\lambda\overline{x_n}$ and body σr.

Certain type constraints are left implicit in this definition. Let $\tau_1, \ldots, \tau_n$ be the types of $x_1, \ldots, x_n$, respectively. Let τ_0 be the type of l and r, and τ the type of M and N. The term u must have the type $((\tau_1 \rightarrow \cdots \rightarrow \tau_n) \rightarrow \tau_0) \rightarrow \tau$. We call the term u the *context*, and the terms L and R the *left closure* and *right closure*, respectively. Note that such closures are always closed terms. We write $\overset{*}{\leftrightarrow}_H$ to denote the reflexive, symmetric, transitive closure of this relation.

Given a rule $l \rightarrow r$ in H, it must of course be the case that $l \rightarrow_H r$. To see how, let $z_1, \ldots, z_n$ be the free variables of l, and let $x_1, \ldots, x_n$ be n variables not occurring free in l of the same types as $z_1, \ldots, z_n$, respectively. Let σ be the substitution that maps z_i to x_i for $i = 1, \ldots, n$, and L the term with binder $\lambda\overline{x_n}$ and body σl. Let w be a variable distinct from $z_1, \ldots, z_n$. Then with $\lambda w.w\overline{z_n}$ as context and L as left closure, we have $l \rightarrow_H r$.

To illustrate further the above definition, we consider $\beta\eta$-convertibility for the untyped λ-calculus expressed as a higher-order rewrite system. Note that in this example, the simply typed λ-calculus is the meta-language while the untyped λ-calculus is the object language. We introduce a primitive type tm for untyped terms and two constants app and abs of type $tm \rightarrow tm \rightarrow tm$ and $(tm \rightarrow tm) \rightarrow tm$, respectively, used to code untyped terms. The rewrite system consists of the following two rewrite rules corresponding to β and η-conversion in the untyped λ-calculus.

$$app\ (abs\ M)\ N \rightarrow MN$$
$$abs\ \lambda x.(app\ M\ x) \rightarrow M$$

In the second rule, the bound variable 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. Consider the term $(abs\ \lambda x.(app\ (abs\ \lambda y.(app\ x\ y))\ x))$ which is the encoding of the untyped λ-term $\lambda x.((\lambda y.xy)x)$. Using either the β or η rewrite rule, we obtain the term $(abs\ \lambda x.(app\ x\ x))$. We obtain this result by the η rule with substitution $[z/M]$, context $\lambda w.(abs\ \lambda x.(app\ (wx)\ x))$, and left closure $\lambda z.(abs\ \lambda y.(app\ z\ y))$, and

by the β rule with substitution $[\lambda y.(app\ z\ y)/M, z/N]$, context $\lambda w.(abs\ \lambda x.wx)$, and left closure $\lambda z.(app\ (abs\ \lambda y.(app\ z\ y))\ z)$.

A first-order rewrite system (see Dershowitz [5], for example) can be described as an HRS such that all terms and all subterms in rewrite rules are of primitive type. For this subset of rewrite systems we can in fact restrict Definition 1 so that the type of the left and right closures is also primitive, and thus the outermost binder is empty. In this way, we obtain a definition that is equivalent to the one often given for first-order rewriting.

In addition to the "operational" definition of rewriting above, we define the logical notion of equality modulo an HRS H. We formalize this notion in terms of an inference system. We will see that the operational behavior of the programs in Sections 5 and 6 will correspond quite closely to constructing proof trees in this inference system. Formulas in this system are universally quantified equalities between terms of primitive type, *i.e.*, of the form $\forall x_1 \ldots \forall x_n (M = N)$, often also written $\forall \overline{x_n}(M = N)$. The inference rules include the following two rules which are the usual rules for universal elimination and introduction.

$$\frac{\forall x\, A}{[t/x]A}\ \forall\text{-E} \qquad\qquad \frac{[y/x]A}{\forall x\, A}\ \forall\text{-I}$$

In the $\forall$-E rule t must be a term of the same type as x, and in $\forall$-I the usual proviso that the variable y cannot appear free in $\forall x A$ holds. In addition, for every rule $l \to r$ in H, there is an axiom of the form:

$$\forall \overline{x_n}(l = r)$$

where $x_1, \ldots, x_n$ are the free variables in l. In addition, we have the following congruence rule.

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

Here 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. We also have the usual rules for reflexivity, symmetry, and transitivity, with the additional restriction that the rules are only applied to terms at primitive type. For HRS H, we write $\vdash_H A$ if formula A is provable in the inference system consisting of the above rules.

The corresponding inference system often given for equality modulo first-order equations (as in Dershowitz [5], for example) can be seen to be contained in this inference system. The first-order congruence rule, for instance, can be seen to be a special case of the (CONG) rule given here. In the first-order case, h is always a constant and the arity of the arguments to h is always 0. In addition, instead of a single axiom for each rewrite rule, axioms are often included for

every instance of a rewrite rule. As a result, universal quantifiers do not appear in formulas and the $\forall$-E and $\forall$-I rules are not needed.

In Appendix A, we prove the equivalence at primitive types of the operational definition of rewriting using $\overset{*}{\leftrightarrow}_H$ given by Definition 1 and the logical definition given by the above inference system. In particular, we show that for arbitrary terms M and N of arity n, and for distinct variables $x_1, \ldots, x_n$ not free in M or N that $\vdash_H \forall \overline{x_n}(M\overline{x_n} = N\overline{x_n})$ if and only if $M \overset{*}{\leftrightarrow}_H N$. In the next section, we present further examples of higher-order rewrite systems which express common operations in both functional programming and theorem proving.

3 Further Examples of Higher-Order Rewrite Systems

We consider a simple functional programming language consisting of primitive datatypes for booleans and natural numbers, constructs for lists, function abstraction, application, a conditional statement, a fix point operator, and the *let* operator as in ML. Hannan and Miller [12] give a specification of evaluation for this language in terms of inference rules in a meta-language similar to the one that will be used to implement rewriting. As in that paper, we use a higher-order abstract syntax for functional programs. As in the rewrite system for $\beta\eta$-convertibility in the last section, we use a single primitive type, tm, for terms in this language. We use the *abs* and *app* constructs as in that section and introduce new constants and their types for the remaining program constructs.

$$\begin{array}{rlrl}
true & : tm & hd & : tm \to tm \\
false & : tm & tl & : tm \to tm \\
0 & : tm & empty & : tm \to tm \\
s & : tm \to tm & if & : tm \to tm \to tm \to tm \\
nil & : tm & fix & : (tm \to tm) \to tm \\
cons & : tm \to tm \to tm & let & : (tm \to tm) \to tm \to tm
\end{array}$$

Clearly not all terms of 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 following rewrite system expresses evaluation in this language.

$$\begin{array}{rlrl}
hd\ (cons\ M\ N) & \to M & if\ true\ M\ N & \to M \\
tl\ (cons\ M\ N) & \to N & if\ false\ M\ N & \to N \\
empty\ nil & \to true & app\ (abs\ M)\ N & \to MN \\
empty\ (cons\ M\ N) & \to false & fix\ M & \to M\ (fix\ M) \\
 & & let\ M\ N & \to MN
\end{array}$$

Note that as a rewrite system, these rules express non-deterministic evaluation. Nothing about order of evaluation in specified. In Section 6, we will see that different rewriting strategies when given these rewrite rules as a parameter correspond to various strategies for evaluating functional programs.

In theorem proving, formulas are often pre-processed in order to obtain a certain form before applying inference rules. Many such pre-processing steps can be expressed as higher-order rewrite systems. Nipkow [23] discusses rewrite systems for putting first-order classical formulas in negation normal form, in prenex normal form, and for moving quantifiers inward. As another example, in an intuitionistic or classical logic with connectives $\wedge, \supset, \forall$, any formula can be put into the equivalent form $\forall x_1 \ldots \forall x_n\ (A_1 \wedge \cdots \wedge A_m \supset B_1 \wedge \cdots \wedge B_p)$ where $n, m, p \geq 0$, $A_1, \ldots, A_m$ have the same form recursively, and $B_1, \ldots, B_p$ have the same form recursively with no outermost quantification. (Clauses in the logic programming language described in the next section will always be written in this form, with $p = 1$.) We describe a rewrite system for putting first-order formulas in this form. We use primitive type *tm* for first-order terms, introduce type *form* for first-order formulas, and introduce the following constants and their types for the connectives.

$$\begin{aligned} \wedge &: \mathit{form} \to \mathit{form} \to \mathit{form} \\ \supset &: \mathit{form} \to \mathit{form} \to \mathit{form} \\ \forall &: (\mathit{tm} \to \mathit{form}) \to \mathit{form} \end{aligned}$$

We use the usual infix notation for $\wedge$ and $\supset$. The following rewrite system performs the desired operation.

$$\begin{aligned} (\forall A) \wedge (\forall B) &\to \forall\ \lambda x.(Ax \wedge Bx) & \qquad A \supset (\forall B) &\to \forall\ \lambda.x(A \supset Bx) \\ (\forall A) \wedge B &\to \forall\ \lambda x.(Ax \wedge B) & \qquad A \supset (B \supset C) &\to (A \wedge B) \supset C \\ A \wedge (\forall B) &\to \forall\ \lambda x.(A \wedge Bx) \end{aligned}$$

As a final example, we consider a rewrite system expressing proof reductions for normalization in natural deduction. Again, we consider only the $\wedge, \supset, \forall$ connectives. We express proofs as λ-terms using the primitive type *prf* and the following constants.

$$\begin{aligned} \wedge\text{-I} &: \mathit{prf} \to \mathit{prf} \to \mathit{prf} & \qquad \supset\text{-I} &: (\mathit{prf} \to \mathit{prf}) \to \mathit{prf} \\ \wedge\text{-E}_1 &: \mathit{prf} \to \mathit{prf} & \qquad \supset\text{-E} &: \mathit{prf} \to \mathit{prf} \to \mathit{prf} \\ \wedge\text{-E}_2 &: \mathit{prf} \to \mathit{prf} & \qquad \forall\text{-I} &: (\mathit{tm} \to \mathit{prf}) \to \mathit{prf} \\ & & \qquad \forall\text{-E} &: \mathit{prf} \to \mathit{tm} \to \mathit{prf} \end{aligned}$$

Note that the $\supset$-introduction rule is represented by a constant that takes a function from proofs to proofs as argument, and the $\forall$-introduction rule takes a function from first-order terms to proofs. The first argument to $\forall$-E is the proof of the premise and the second is the substitution term. Details on this representation of natural deduction proofs can be found in Felty [8]. The required reductions can be expressed as follows.

$$\begin{aligned} \wedge\text{-E}_1\ (\wedge\text{-I}\ P\ Q) &\to P & \qquad \supset\text{-E}\ (\supset\text{-I}\ P)\ Q) &\to PQ \\ \wedge\text{-E}_2\ (\wedge\text{-I}\ P\ Q) &\to Q & \qquad \forall\text{-E}\ (\forall\text{-I}\ P)\ M &\to PM \end{aligned}$$

Natural deduction proofs are strongly normalizable and thus any complete reduction strategy using these rules will reduce an arbitrary proof to its normal

form. For intuitionistic logic, this property remains true even if the $\vee$ and $\exists$ connectives are added. In this case, many more reduction rules are needed to handle permutations of the $\vee$-elimination and $\exists$-elimination rules with other rules. These permutations can also be described as rewrite rules. For a complete specification of the reductions for full intuitionistic first-order logic using the representation of proof trees as given here, see Felty [6].

4 The Meta-Logic and Language

We now present the meta-language used to specify and implement rewrite systems in the remainder of this paper. The terms of the meta-logic are the simply typed λ-terms, the same language used to define higher-order rewrite systems. We often speak of a fixed *signature* or a finite set of constants and variables, usually denoted Σ. As before, equality between λ-terms is taken to mean $\beta\eta$-convertible. We assume that the symbol o is always a member of the fixed set of primitive types. Following Church [3], o is the type for propositions. The logical constants are given the following types: $\wedge$ (conjunction), $\vee$ (disjunction), and $\supset$ (implication) are of type $o \to o \to o$; and $\forall_\tau$ (universal quantification) and $\exists_\tau$ (existential quantification) are of type $(\tau \to o) \to o$, for all types τ. A formula is a term of type o. The logical constants $\wedge$, $\vee$, and $\supset$ are written in the familiar infix form. The expression $\forall_\tau(\lambda z\ M)$ is written $\forall_\tau z M$ or simply $\forall z M$ when the type τ can be inferred from context.

A proposition whose $\beta\eta$-long form is such that the head h is not a logical constant will be called an *atomic* formula. The head h is called a *predicate*. In this section, A denotes a syntactic variable for atomic formulas. We now define two new classes of propositions, called *goal formulas* and *definite clauses*. Let G be a syntactic variable for goal formulas and let D be a syntactic variable for definite clauses. These two classes are defined by the following mutual recursion.

$$G := A \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid D \supset G \mid \exists_\tau x G \mid \forall_\tau x G$$

$$D := A \mid G \supset A \mid \forall_\tau x D$$

Definite clauses are also called *higher-order Hereditary Harrop* formulas (hohh for short). There is one final restriction: if an atomic formula is a definite clause, it must have a constant as its head. The heads of atomic goal formulas may be either variable or constant. Note that the top-level form of a definite clause is either $\forall\overline{x_n}A$ or $\forall\overline{x_n}(G \supset A)$ where $n \geq 0$ and the head of A is a constant. In either case, the atomic formula A is called the *head* of the clause, and G is called the *body*. A *logic program* or just simply a *program* is a finite set of closed definite clauses.

We also consider a subset of this logic called hh^ω which only includes the $\wedge$, $\supset$, and $\forall_\tau$ connectives, where τ is a type not containing occurrences of o. Thus this language doesn't allow quantification over predicates. This restricted language will be sufficient for specifying higher-order rewrite systems as we will

see in the next section. In Sections 6 and 7, we will use the full hohh for implementing rewriting strategies.

From properties about hohh presented in Miller et al. [21], a sound and complete (with respect to intuitionistic logic) *non-deterministic* interpreter can be implemented by employing the following six *search operations*. Here, the interpreter is attempting to determine if the goal formula G follows from the program $\mathcal{P}$.

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

OR: If G is $G_1 \vee G_2$ then try to show that either G_1 or G_2 follows from $\mathcal{P}$.

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

INSTANCE: If G is $\exists_\tau x G'$ then pick some closed λ-term M of type τ and try to show $[M/x]G'$.

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 $\mathcal{P}$. If neither case holds then G does not follow from $\mathcal{P}$.

An interpreter for hh^ω uses the same operations, but doesn't require OR or INSTANCE. An implementation of an interpreter must make many choices which are left unspecified in the high-level description above. We discuss some of the choices made by the logic programming language λProlog since we later present several λProlog programs. For example, 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 $\mathcal{P}$ using a depth-first search paradigm to handle failures. Logic variables as in Prolog are used in the INSTANCE operation and in forming a universal instance in the BACKCHAIN operation. These variables can later be instantiated through unification. Here, unification on simply typed λ-terms is required. λProlog implements a depth-first version of the unification search procedure described in Huet [14]. Most unification problems we shall encounter in executing programs in this paper are simple. In particular, although at times there may be more than one unifier, there will generally be a finite number that can be easily enumerated by the unification procedure.

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.

In addition to these restrictions, λProlog contains some extensions that we will make use of here. First, a degree of polymorphism is permitted by allowing type declarations to contain type variables (written as capital letters). Second, negation by failure and the cut operator (!) as in Prolog are added. The cut is a goal which always succeeds and commits the interpreter to all choices made since the parent goal was unified with the head of the clause in which the cut occurs.

5 Specifying Rewrite Systems

In this section, we discuss the specification of higher-order rewrite systems in hh^ω. The specification of syntax of terms is direct since both the meta-language of higher-order rewrite rules and our specification language hh^ω contain simply typed λ-terms. 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. To specify rewriting at a particular primitive type τ, we introduce the infix arrow $\longrightarrow_\tau$ to serve as a predicate of type $\tau \to \tau \to o$. The specification of rewrite rules as clauses is then straightforward: using these predicates, we take the universal closure over the free variables of the rewrite rule. For example, the rewrite rules for for $\beta\eta$-convertibilty of the untyped λ-calculus are expressed as the formulas below.

$$\forall M \forall N (app\ (abs\ M)\ N \longrightarrow_{tm} MN)$$
$$\forall M (abs\ \lambda x.(app\ M\ x) \longrightarrow_{tm} M)$$

Generally, in executing rewrite goals, we will 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 the first clause for example, M and N will be replaced with logic variables, and a term that represents a β-redex will unify with the pattern on the left of the arrow. The substitution of N for the bound variable in M is achieved by application of λ-terms at the meta-level MN.

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.

Rewrite rules express reducibility in one step. To express reducibility in 0 or more steps, we introduce the predicate $\longrightarrow^*_\tau$ and include the following formula for each primitive type τ: $(M \longrightarrow_\tau N) \supset (M \longrightarrow^*_\tau N)$.

To specify congruence, we introduce an hh^ω formula for each constant in Σ. For example, the following two formulas are included for the *app* and *abs* constants.

$$(M \longrightarrow^*_{tm} P) \wedge (N \longrightarrow^*_{tm} Q) \supset (app\ M\ N \longrightarrow^*_{tm} app\ P\ Q)$$
$$\forall x((x \longrightarrow^*_{tm} x) \supset (Mx \longrightarrow^*_{tm} Nx)) \supset (abs\ M \longrightarrow^*_{tm} abs\ N)$$

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 rewrites to Nx. Operationally, in trying to solve a goal of the form $(abs\ M' \longrightarrow^*_{tm} abs\ N')$ 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 $(c \longrightarrow^*_{tm} c)$ stating that this constant reduces to itself. This can be considered as the dynamic addition of a new constant to the object-level signature and a congruence rule for that constant. Then, β-reduction at the meta-level of $M'c$ performs the substitution of the new item 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 $(c \longrightarrow^*_{tm} c)$ 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.

Note that in the presence of a clause representing a reflexive rule at type *tm*: $\forall M(M \longrightarrow^*_{tm} M)$, the addition of the new clause $(c \longrightarrow^*_{tm} c)$ is redundant since any subgoal of this form can be proved using either this new clause or the reflexive axiom. In the presence of the reflexive axiom, the congruence rule for *abs* could be simply specified as follows.

$$\forall x(Mx \longrightarrow^*_{tm} Nx) \supset (abs\ M \longrightarrow^*_{tm} abs\ N)$$

Alternatively, if we restrict to a particular signature, a formula specifying reflexivity can be omitted as long as we include congruence clauses for all atomic constants in Σ, and specify congruence for functional constants such as *abs* so that formulas explicitly include assumptions about the congruence of new constants. For example, if our signature for untyped terms includes constants f and a in addition to *app* and *abs*, we must include the clauses $(f \longrightarrow^*_{tm} f)$ and $(a \longrightarrow^*_{tm} a)$. Operationally, the inclusion of a reflexive rule can be more efficient since it can be used to prove the equivalence of two arbitrary terms of primitive type. Without it, congruence rules must be used to descend through the entire term, applying congruence rules for constants at the leaves. An advantage of this latter approach is that it also verifies that a term is well-formed in a particular signature.

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

$$\forall f(\forall P((P \longrightarrow^*_{tm} P) \supset (fP \longrightarrow^*_{tm} fP)) \supset (Mf \longrightarrow^*_{tm} Nf)) \supset (g\ M \longrightarrow^*_{tm} g\ N)$$

Operationally, after backchaining on the above clause, instead of an atomic clause, the clause $(P \longrightarrow^*_{tm} P) \supset (fP \longrightarrow^*_{tm} fP)$ would be dynamically added by AUGMENT, serving as a congruence rule for the new function symbol f.

We can in fact define a general function for specifying congruence rules for a particular signature. For signature item a of type τ, the following function

defined by induction on the structure of τ yields the necessary congruence rule.

$$[\![a : \tau]\!] = \begin{cases} a \longrightarrow^*_\tau a & \text{if } \tau \text{ is a primitive type} \\ \forall_{\tau_1} x([\![x : \tau_1]\!] \supset [\![ax : \tau_2]\!]) & \text{if } \tau \text{ is } \tau_1 \to \tau_2 \end{cases}$$

(This function is similar to the one used by Felty [7] to code a dependent typed λ-calculus in hh^ω and by Miller [20] to specify equality and substitution for simply typed λ-terms.)

To complete the specification, we introduce the predicate $\longleftrightarrow^*_\tau$ at each primitive type τ. The following clauses express the reflexive, symmetric, transitive closure of $\longrightarrow^*_{tm}$.

$$\begin{aligned} (N \longrightarrow^*_{tm} M) &\supset (N \longleftrightarrow^*_{tm} M) \\ (N \longleftrightarrow^*_{tm} M) &\supset (M \longleftrightarrow^*_{tm} N) \\ (M \longleftrightarrow^*_{tm} P) \wedge (P \longleftrightarrow^*_{tm} N) &\supset (M \longleftrightarrow^*_{tm} N) \end{aligned}$$

Operationally, we may often want to attempt to reduce a term until there are no longer any rewrite rules that apply. Equality of terms under a strongly normalizing rewrite system can be checked this way, *i.e.*, two terms are each reduced to their normal form and then checked to see if they are identical. In such a case, we can replace the formulas for symmetry and transitivity with clauses that have better operational behavior. We introduce a new predicate *normal* of type $tm \to tm \to o$ which relates a term to its normal form, and include the following formulas.

$$\begin{aligned} (M \longrightarrow^*_{tm} N) \wedge (M = N) &\supset (\mathit{normal}\ M\ N) \\ (M \longrightarrow^*_{tm} P) \wedge (\mathit{normal}\ P\ N) &\supset (\mathit{normal}\ M\ N) \end{aligned}$$

If the clauses are ordered so that the rewrite rules appear before the congruence rules, then whenever a goal of the form $(\mathit{normal}\ M\ N)$ is given such that M is a closed term and N is a logic variable, N will get instantiated to the normal form of M. The first clause handles the case when M is already in normal form, while the second clause handles the case when M reduces in one or more steps to some term P. A recursive call is made to further reduce P if possible.

This example illustrates the general manner in which higher-order rewrite systems can be specified rather directly in hh^ω. All of the other examples of rewrite systems given in the previous section, for instance, can be specified in the same way. Several of these rewrite systems have some notion of object-level types. For example, although we expressed $\beta\eta$-convertibility for untyped terms, the same rewrite system can express this relation for the simply typed λ-calculus. Also, although the rewrite relation for evaluation of the functional language given in Section 3 doesn't rely on type information, the programming language itself is a typed language. In addition, the rewrite relation on first-order formulas could equally apply to a many-sorted first-order logic where terms are required to be correctly typed. We now consider a slight modification of the $\beta\eta$-conversion on untyped terms, and illustrate how to include typing information

in order to restrict rewriting to typable terms. To represent types, we introduce the primitive type ty and the arrow $\Rightarrow$ of type $ty \to ty \to ty$. To express rewriting and equivalence at a particular type, we replace the binary relations on terms with ternary relations which include an argument for the type. For example, to express rewriting in 0 or more steps, we introduce the predicate rew^*_{tm} of type $ty \to tm \to tm \to o$. Using this predicate, congruence for app and abs can be expressed as follows.

$$(rew^*_{tm}\ A \Rightarrow B\ M\ P) \wedge (rew^*_{tm}\ A\ N\ Q) \supset (rew^*_{tm}\ B\ (app\ M\ N)\ (app\ P\ Q))$$
$$\forall x((rew^*_{tm}\ A\ x\ x) \supset (rew^*_{tm}\ B\ Mx\ Nx)) \supset (rew^*_{tm}\ A \Rightarrow B\ (abs\ M)\ (abs\ N))$$

As before, congruence rules for constants can be given also, in this case with their types. Let nat be a constant of type ty and let f represent a unary function from nat to nat, and a a constant of this type. The corresponding clauses are $(rew^*_{tm}\ nat \Rightarrow nat\ f\ f)$ and $(rew^*_{tm}\ nat\ a\ a)$, respectively. Typing information can similarly be added to the formulas specifying the β and η rewrite rules, as well as to reflexivity, symmetry, and transitivity. In a specification of reduction for typed terms, if we omit the reflexive rule and rely on the congruence rules only, in addition to verifying that terms are well-formed, these rules also verify that terms are correctly typed. In fact, they can also be used to infer types when a logic variable is given for the type argument in a goal.

6 Implementing Tactics for Rewriting

We have now seen how higher-order rewrite systems can be directly specified in our logic programming language and how the basic operations of this logic programming language implement basic operations of higher-order term rewriting. In this and the next section, we go one step further and present λProlog programs for implementing various rewriting techniques and strategies.

In this section, since we present several λProlog programs, we will adopt the syntax of this language. The comma (`,`), semicolon (`;`), and arrow (`=>`) represent $\wedge$, $\vee$, and $\supset$, respectively, while `:-` denotes the converse of `=>` and is used to write the top-level implication in clauses. λ-abstraction is written using backslash `\` as an infix operator, and universal quantification is written using the constant `pi` in conjunction with a λ-abstraction. As in the previous section, we assume universal closure over all variables written as tokens with an upper case initial letter. Finally, a signature member, say f of type $a \to b \to c$ is represented as simply the line:

```
type   f       a -> b -> c.
```

Using this syntax and the infix symbols `-->` and `-->*` for the $\longrightarrow_{tm}$ and $\longrightarrow^*_{tm}$ relations in the previous section, the clauses for the β-reduction rule and congruence for abs, for example, would be written:

```
(app (abs M) N) --> (M N).
(abs M) -->* (abs N) :- pi x\ ((x -->* x) => ((M x) -->* (N x))).
```

Tactic style theorem provers were first built in the early LCF systems and have been adopted as a central mechanism in such notable theorem proving systems as Edinburgh LCF [10], HOL [11], Nuprl [4], and Isabelle [24]. All of these systems are implemented in the functional programming language ML. Tactic style theorem provers for a variety of logics can also be specified and implemented in the logic programming language used here (see Felty and Miller [8] and Felty [6]). In such an implementation, basic tactics express the inference rules of a particular logic and serve as the primitive operations of a theorem prover, while tacticals provide high-level control over search. Tactics and tacticals can be combined to build more complex strategies and partially automated procedures. In this section, we illustrate that strategies for rewriting can be naturally expressed as logic programs and easily integrated into this framework. Term rewriting can thus be added to an existing tactic theorem prover for a particular logic, providing a flexible means for reasoning about the equality relation of that logic.

We begin in subsection 6.1 by presenting the basic data structures for goals of the tactic theorem prover. Then, in subsection 6.2, we slightly modify the specification of rewrite and congruence rules given in Section 5 to provide basic tactics for rewriting. In subsection 6.3, we present tacticals which implement the general interpreter for tactic theorem proving. Finally, in subsection 6.4, we build on these procedures to implement general and specialized strategies for rewriting. In summary, subsections 6.1 and 6.3 represent a slight extension of the general tactic interpreter given by Felty and Miller [8, 6], while subsections 6.2 and 6.4 illustrate how to add mechanisms for higher-order rewriting.

Although the specifications in the previous section used only hh^{ω}, in this section we will make use of some aspects outside the scope of this language. For example, predicate quantification of the more expressive language hohh can be used to obtain elegant implementations of the high-level control procedures. It will also be convenient to make some uses of the polymorphism of λProlog as well as the cut (`!`) operator for finer control.

6.1 Data Structures for Goals

In Felty [6], we have one goal constructor corresponding to each of the search operations of the logic programming interpreter. Those that will be used here are given with their types below.

```
type    tt      goal.
type    &&      goal -> goal -> goal.
type    ==>>    o -> goal -> goal.
type    all     (A -> goal) -> goal.
```

The goal `tt` corresponds to the trivially satisfied goal, `&&` to the AND search operation, `==>>` to AUGMENT, and `all` to GENERIC. We use infix notation for `&&` and `==>>`. We call goals constructed from these constants *compound* goals, in contrast to *primitive* rewriting goals which we will present in the next subsection. In addition, we have two constructors for building *goal trees*.

```
type    **      goal -> goal -> goal.
type    lf      goal -> goal.
```

The ** operator is the (infix) node constructor and takes two trees as arguments. Leaf nodes are indicated by `lf` and take a compound goal as an argument. Such trees form the top-level goal structure, *i.e.*, node and leaf constructors appear only at the top level and not inside compound goals built from `&&`, `==>>`, or `all`.

6.2 Basic Tactics for Rewriting

Instead of expressing equivalence at each primitive type τ as we did in Section 5 using $\longleftrightarrow_\tau^*$, we introduce the infix relation `eq` of polymorphic type `A -> A -> goal`. In practice, we will have primitive rewriting goals of the form `(M eq N)` where `M` is a closed term and `N` is a variable. Thus `A` will be instantiated with the type of `M`. In addition, it will always be a primitive type. Using a polymorphic type here allows us to describe a general package for rewriting independent of the primitive types of a particular rewrite system. For simplicity, we use a single relation instead of three distinct relations as in the previous section. Both rewrite and congruence rules will be expressed using `eq`.

A tactic in this setting is a binary relation on goals and has type `goal -> goal -> o`. The first goal will be the "input" goal or goal to be proved, and the second the "output" goal whose instances are the subgoals which must subsequently be proved. Generally, when applying a tactic, the first subgoal is at least partially instantiated, and the second goal is a logic variable.

Specifying rewrite rules as tactics is straightforward. The tactics for $\beta\eta$-conversion for the untyped λ-calculus for example are as follows.

```
rew ((app (abs M) N) eq (M N)) tt.
rew ((abs x\ (app M x)) eq M) tt.
```

If the input goal to the `rew` tactic is an instance of one of the first arguments in the above clauses, then the goal succeeds and there are no remaining subgoals. As stated above, the term on the right of the arrow in a rewriting goal will often be a variable. Thus the input goal as a whole will be only partially instantiated. The tactic succeeds if the term on the left is a β- or η-redex. The variable on the right is then instantiated to its reduced form. It is also possible to specify each rewrite rule with a distinct name, so that it becomes possible to control which rewrite rule is attempted. To do so, we may replace the name `rew` above with `beta` and `eta`, for example.

The congruence rules for *app* and *abs* are specified as follows.

```
cong ((app M N) eq (app P Q)) ((lf (M eq P)) ** (lf (N eq Q))).
cong ((abs M) eq (abs N))
     (lf (all x\ ((cong_const (eq x x) tt) ==>> (eq (M x) (N x))))).
```

This specification is similar to that in the previous section. The input/output relation between the two arguments to `cong` here corresponds to the relation of a head of a clause and its body in the previous specification. Here, the two

subgoals for *app* form two leaves in a goal tree. The clause for `abs` can be read as before: an abstraction `(abs M)` rewrites to `(abs N)` if for arbitrary `x` such that `x` rewrites to itself, `(M x)` rewrites to `(N x)`. The operational reading is more indirect. It will depend on how we process the goal constructors. For example, `all` and `==>>` will be implemented using the GENERIC and AUGMENT operations. The left hand side of the implication `(cong_const (eq x x) tt)` represents a congruence tactic for the new constant of type *tm* introduced for `x`. We use `cong_const` here instead of `cong`. For reasons that will become apparent later, we want to distinguish congruence at meta-level primitive and functional types. As before, we may wish to make even more distinctions and replace the name `cong` above with `cong_app` and `cong_abs` for example. The remaining clauses for congruence in the $\beta\eta$-convertibility example are those for the constants f and a.

```
cong_const (f eq f) tt.
cong_const (a eq a) tt.
```

6.3 The High-Level Search Primitives

Next, we present the high-level tacticals that will be useful for implementing rewriting tactics. The λProlog implementation is very natural and extends the usual meaning of tacticals by permitting them to have access to logic variables and the search operations.

```
maptac Tac tt tt.
maptac Tac (InG1 && InG2) (OutG1 && OutG2) :- maptac Tac InG1 OutG1,
                                              maptac Tac InG2 OutG2.
maptac Tac (all InG) (all OutG) :- pi x\ (maptac Tac (InG x) (OutG x)).
maptac Tac (D ==>> InG) (D ==>> OutG) :- D => (maptac Tac InG OutG).
maptac Tac InG OutG :- Tac InG OutG.

then Tac1 Tac2 InG OutG :- Tac1 InG MidG, maptac Tac2 MidG OutG.
orelse Tac1 Tac2 InG OutG :- Tac1 InG OutG; Tac2 InG OutG.
idtac G G.
repeat Tac InG OutG :- orelse (then Tac (repeat Tac)) idtac InG OutG.
try Tac InG OutG :- orelse Tac idtac InG OutG.
```

The `maptac` tactical descends through the structure of a compound goal and applies the argument tactic to the primitive goals.[2] The `then` tactical performs the composition of tactics. The `maptac` procedure is used in the second subgoal since the application of `Tac1` may result in an output goal (`MidG`) with compound structure. This tactical plays a fundamental role in combining the results of step-by-step proof construction. The substitutions resulting from applying these

[2] Note that the `maptac` clause for implication is not allowed by the definition of definite clauses given in Section 4 because `D` is a variable occurring on the left of `=>`. In λProlog, variables in this position are acceptable, but a runtime check is included to insure that `D` is instantiated to a definite clause before it is added to the program using the AUGMENT operation. In the examples here, it will always be instantiated to an atomic formula.

separate tactics get combined correctly since `MidG` provides the necessary sharing of logical variables between these two calls to tactics. The `orelse` tactical attempts to apply either `Tac1` or `Tac2`. The next tactical, `idtac`, simply returns the input goal unchanged. The `repeat` tactical repeatedly applies a tactic until it is no longer applicable. Finally, the `try` tactical prevents failure of the given tactic by using `idtac` when `Tac` fails.

It is worth noting the differences between the ML and λProlog implementations of the `then` tactical. The λProlog implementation of `then` reveals its very simple nature: `then` is very similar to the natural join of two relations. In ML, the `then` tactical applies the first tactic to the input goal and then maps the application of the second tactic over the list of intermediate subgoals. The full list of subgoals must be built as well as the compound validation function from the results. These tasks can be quite complicated, requiring some auxiliary list processing functions. In λProlog, the analogue of a list of subgoals is a nested `&&` structure. These are processed by the clause of `maptac` which handles `&&`. The `maptac` procedure is richer than the usual notion of a mapping function in that, in addition to nested `&&` structures, it can handle other goal structures corresponding to the λProlog search operations. For more on these tacticals and their comparison to the ML implementation, see Felty [6].

The `maptac` tactical above provides a uniform way in which to apply a tactic to all primitive goals in a goal structure. It will also be useful to be able to apply a tactic to a particular primitive goal while leaving others untouched. The tree structure on goals is provided for this purpose. The following tacticals provide operations which descend through a tree structure and choose a primitive goal to which a tactic will be applied.

```
left_node Tac (lf InG) (lf OutG) :- maptac Tac InG OutG.
left_node Tac (InG ** G) (OutG ** G) :- left_node Tac InG OutG.

first_node Tac (lf InG) (lf OutG) :- maptac Tac InG OutG.
first_node Tac (InG ** G) (OutG ** G) :- first_node Tac InG OutG, !.
first_node Tac (G ** InG) (G ** OutG) :- first_node Tac InG OutG.

tree_to_goal (lf G) G.
tree_to_goal (InG1 ** InG2) (OutG1 && OutG2) :- tree_to_goal InG1 OutG1,
                                                tree_to_goal InG2 OutG2.

untree Tac InG OutG :- then Tac tree_to_goal InG OutG.
left Tac InG OutG :- untree (left_node Tac) InG OutG.
first Tac InG OutG :- untree (first_node Tac) InG OutG.
```

The `left_node` tactical applies a tactic to the leftmost leaf goal, while the `first_node` tactical performs a depth-first search over the tree structure searching for the first leaf goal to which the tactic can successfully be applied. The cut (`!`) is essential for the desired behavior. This tactical should only succeed once and only on the first possible leaf. We can implement a tactical that succeeds if the tactic can be applied to *any* leaf goal by simply removing the cut. Operationally, leaf goals would be attempted in a left to right order, only going beyond

the first one if subsequent failure causes backtracking. The `tree_to_goal` tactical simply removes the tree structure from a goal. This tactical is used by the `untree` tactical which applies `tree_to_goal` after applying a tactic. It is useful when the remaining tree structure is no longer important after applying a tactic to a particular leaf. The `left` and `first` tacticals use `untree` to remove tree structure after applying a tactic to the leftmost leaf or first possible leaf, respectively.

6.4 Implementing Rewriting Strategies

This completes the implementation of the tacticals. We return to the implementation of rewriting. As discussed in the previous section, we do not need to implement a general tactic for reflexivity. Instead, we can repeatedly apply congruence rules. For example, the following tactic implements reflexivity for untyped λ-terms.

```
refl_tm InG OutG :- then (repeat (untree cong)) cong_const InG OutG.
```

This tactic applies congruence rules to descend through terms, applying congruence rules for constants at the leaves. In fact, we can write a general tactical used for implementing reflexive tactics that take tactics for congruence of constants at both functional and primitive types.

```
refl Cong Const InG OutG :- then (repeat (untree Cong)) Const InG OutG.
```

Reflexivity for a particular primitive type can then be defined using this tactical as in the following example for the untyped λ-calculus.

```
refl_tm InG OutG :- refl cong cong_const InG OutG.
```

Of course, it is also possible to implement the reflexive tactic directly.

```
refl (M eq M) tt.
```

The transitivity and symmetry rules are specified simply as the formulas below.

```
sym (M eq N) (lf (N eq M)).
trans (M eq N) ((lf (M eq P)) ** (lf (P eq N))).
```

Two common strategies for terminating rewrite systems are bottom-up and leftmost-outermost rewriting of subterms. A procedure for the bottom-up strategy is given as part of an implementation for first-order term rewriting in Isabelle [22]. Although it is written in ML, a very similar implementation can be given in our setting, since it can be defined using previously defined basic tacticals such as `then`, `orelse`, and others. We present the λProlog version here for illustration purposes. As noted above, a more significant difference appears in the implementations of these basic tacticals in the two settings. It should be noted that although this strategy is presented as part of an implementation of first-order rewriting in Isabelle, if given the appropriate rewrite and congruence rules as arguments, it can also serve as a tactic for higher-order rewriting in that setting. The following tacticals implement the bottom-up strategy.

```
left_rew Tac InG OutG :- then trans (left Tac) InG OutG.

bu Cong Rew Refl InG OutG :-
  then (bu_sub Cong Rew Refl)
       (orelse (then (left_rew Rew) (bu Cong Rew Refl)) Refl) InG OutG.

bu_sub Cong Rew Refl InG OutG :-
  try (left_rew (then (untree Cong) (bu Cong Rew Refl))) InG OutG.
```

The `left_rew` tactical is useful for procedures that involve many rewriting steps. It applies transitivity to obtain two subgoals, and then applies `Tac` to solve the first. The remaining subgoal allows further rewrite steps to be performed. The `bu` and `bu_sub` tacticals take a congruence tactic, a rewriting tactic, and a reflexive tactic as arguments. The `bu` tactical proceeds in two steps. First `bu_sub` is applied to perform bottom-up rewriting to all of the subterms. Then a rewrite is attempted on the resulting term. If it succeeds, the `bu` procedure is repeated. If it fails, then the reflexive tactic is applied to complete the rewriting. The `bu_sub` tactical first applies a congruence, and then `bu` is applied recursively to all of the subgoals. The application of `Cong` is within the scope of `untree` since the tree structure must be removed in order to apply `bu` uniformly to all of the subgoals generated by the application of a congruence rule. A top-level `try` is used so that if the term is a constant and the congruence tactic fails, the procedure as a whole terminates successfully. Using this tactical, a tactic for bottom-up rewriting for the untyped λ-calculus is then implemented by the following tactic.

```
bu_tm (M eq N) OutG :- bu cong rew refl_tm (M eq N) OutG.
```

A tactic for leftmost-outermost rewriting can be written similarly.

```
lo Cong Rew Refl InG OutG :-
  then (repeat (left_rew (lo_rew Cong Rew Refl))) Refl InG OutG.

lo_rew Cong Rew Refl InG OutG :-
  orelse Rew (then (then Cong (first (lo_rew Cong Rew Refl)))
                   Refl) InG OutG.
```

Here, the `lo` tactical implements the top-level loop, while `lo_rew` searches for the leftmost-outermost subterm that can be rewritten. In `lo_rew`, a rewrite is attempted directly on the goal. If that fails, a congruence is applied and `lo_rew` is called recursively in the scope of `first`. Thus, `lo_rew` is attempted on all of the subgoals in succession looking for the first possible subterm to which a rewrite can be applied. If successful, the reflexive tactic is applied to complete all remaining subgoals. If no subterms can be rewritten, the tactical fails. The `lo` tactical repeatedly calls `lo_rew` until no more rewrites are possible. In this case, the reflexive tactic is called to reduce the current goal to `tt`.

Consider the term $(abs\ \lambda g.(app\ (abs\ \lambda x.(app\ g\ x))\ a))$. Using the `bu_tm` procedure above, this term is reduced to $(abs\ \lambda g.(app\ g\ a))$ by reducing the

η-redex $(abs\ \lambda x.(app\ g\ x))$ to g. Using `lo_rew`, we obtain the same reduced term, but in this case, the outer β-redex $(app\ (abs\ \lambda x.(app\ g\ x))\ a)$ is reduced.

As another example, let APP be the following term representing the program for appending two lists in our functional language.

$$(fix\ \lambda F.(abs\ \lambda l_1.(abs\ \lambda l_2.\ (if\ (empty\ l_1)\ l_2\ (cons\ (hd\ l_1)\ (app\ (app\ F\ (tl\ l_1))\ l_2))))))$$

Using the `lo` strategy, the term $(app\ (app\ APP\ (cons\ 0\ nil))\ (cons\ (s\ 0)\ nil))$ will reduce to $(cons\ 0\ (cons\ (s\ 0)\ nil))$, while the `bu` strategy will loop, repeatedly applying the rewrite rule for fix and expanding the definition of the function. The `lo` strategy, in fact, corresponds to lazy evaluation of this language.

The `bu` and `lo` tacticals implement general strategies that can be applied with any rewrite system as a parameter. It may also be useful to write strategies specialized to a particular rewrite system. As an example, we implement a strategy for normalizing typed λ-terms as implemented in the Elf higher-order logic programming language [25]. This strategy involves recursion over types, so we use the specification below of the rewrite, congruence, and equivalence tactics that includes an extra argument for types. Here, we replace the binary relation `eq` with the ternary relation of the same name, and use the arrow `-->` of type `ty -> ty -> ty` as the constructor for object level types.

```
beta (eq A (app (abs M) N) (M N)) tt.
eta (eq (A --> B) (abs x\ (app M x)) M) tt.
cong_app (eq B (app M N) (app P Q))
   ((lf (eq (A --> B) M P)) ** (lf (eq A N Q))).
cong_abs (eq (A --> B) (abs M) (abs N))
   (lf (all x\ ((cong_const (eq A x x) tt) ==>> (eq B (M x) (N x))))).
cong_const (eq (nat --> nat) f f) tt.
cong_const (eq nat a a) tt.
refl_tm InG OutG :- refl (orelse cong_app cong_abs) cong_const InG OutG.
sym (eq A M N) (lf (eq A N M)).
trans (eq A M N) ((lf (eq A M P)) ** (lf (eq A P N))).
```

This strategy will also use the following rule, which expresses the fact that a term M at functional type reduces to N if for arbitrary x, the term $(app\ M\ x)$ reduces to $(app\ N\ x)$.

```
eq_arrow (eq (A --> B) M N) (lf (all x\ ((cong_const (eq A x x) tt) ==>>
                                          (eq B (app M x) (app N x))))).
```

This rule can alternatively be defined in terms of the others as the following tactic.

```
eq_arrow InG OutG :- then (left_rew (then (untree sym) eta))
                          (then (right_rew eta) cong_abs) InG OutG.
```

Here `right_rew` is the dual to `left_rew` and can be defined similarly. The rewrite strategy we will implement can be described as follows. First, repeatedly

apply `eq_arrow` until the resulting term has primitive type. If the term is a constant, it is reduced; otherwise it is an application. If the left term is a constant then repeat the procedure on the right term. Otherwise perform a weak head reduction and repeat the procedure on the resulting term. The following tactics implement weak head reduction and the general strategy, respectively.

```
whr InG OutG :-
  orelse (left_rew beta) (then cong_app (left whr)) InG OutG.

reduce InG OutG :-
  then (repeat (then (repeat (untree eq_arrow))
                     (orelse (then cong_app (left cong_const))
                             whr))) refl_tm InG OutG.
```

The main loop of the `reduce` tactic consists of first repeatedly applying the `eq_arrow` tactic and then applying one of the two arguments to `orelse`. The first applies `cong_app` and completes the first of the two subgoals if the left term is a constant. The second applies `whr`. In either case, the loop is repeated on the remaining goal. If neither of these steps can be applied, the reduction is complete and the outer `repeat` terminates. The remaining subgoal is reduced to `tt` by the reflexive tactic.

We complete this section with a few words about establishing formal correctness of such tactics. A proof of correctness of a direct specification of $\beta\eta$-convertibility for untyped terms is given in Felty [6]. This specification is similar to those in Section 5, and thus correctness of those given here, including the one that includes type information, should follow similarly. Informally, a correctness result states that two terms M and N are equivalent modulo $\beta\eta$-conversion if and only if, for the corresponding terms encoded using *app* and *abs*, say M' and N', we can show that $M' \longleftrightarrow^*_{lm} N'$. Using the implementations in this section, such a correctness result would involve the `eq` relation and would depend on the correctness of the implementation of the tacticals. Although it has not been done, it should be straightforward to establish this correctness, in particular for those tacticals that are specified as hohh formulas. It will follow directly that any tactic *Tac* written using tacticals and taking the primitive operations for congruence and rewriting as arguments will be sound. That is, we will be able to state that if (*Tac* (*M'* `eq` *N'*) `tt`) is provable in hohh, then $M =_{\beta\eta} N$. Note that proving that a particular tactic implements a desired strategy requires establishing another level of correctness involving details about the execution of the logic programming interpreter.

7 Using Higher-Order Unification to Implement First-Order Rewriting

In this section, we discuss a technique for implementing first-order rewriting that makes extensive use of the operation of higher-order unification on λ-terms

in λProlog. This technique is quite powerful for those cases in which it can be applied. We present the technique and then discuss its limitations.

First, consider a rewrite system such that all terms are of the same primitive type, say `tm`, and consider the following simple tactic.

```
ho_sub ((C X) eq (C Y)) (X eq Y).
```

We assume that `X` and `Y` have type `tm` and that `C` has type `tm -> tm`. This tactic expresses the fact that for any `C`, the term `C` applied to `X` is equivalent to `C` applied to `Y` if `X` is equivalent to `Y`. Operationally, if `X` is equivalent to `Y`, then any term containing 0 or more occurrences of `X` is equivalent to the term such that these occurrences are replaced by `Y`. When this tactic is applied, the terms `C` and `X` are determined by unification. There may be many possible instances arising from the unification of the term to be rewritten against the pattern `(C X)`, and this tactic will succeed once for each one. Note that when `C` is instantiated to a vacuous abstraction, `X` and `Y` could be instantiated to arbitrary terms of type `tm`, which are not necessarily subterms of the term to be rewritten. Operationally, the λProlog interpreter would leave `X` a variable in this case, resulting in a subgoal `(X eq Y)` where both `X` and `Y` are variables. We will in fact not want to consider this case. In λProlog, we can rule out this solution as follows.

```
ho_sub ((C X) eq (C Y)) (X eq Y) :- not (C = z\P).
```

Here, `not` is the negation by failure operator. To see how vacuous instances are ruled out, consider instances of the variable `P`. If such an instance were to contain a free occurrence of the variable `z`, the bound variable name in the pattern `z\P` would have to be changed to avoid capture. Thus, any instance of `P` will not contain any free occurrences of the bound variable in the above pattern.

Suppose we only want to consider those instances such that `X` is a term that can be rewritten directly by some rewrite rule to `Y`. We can do so with the following tactic.

```
ho_rew Rew InG OutG :- then ho_sub Rew InG OutG.
```

Here, the parameter `Rew` should be instantiated to the tactic that applies one of the desired rewrite rules. It is then easy to implement a tactic that applies all rewrite rules until no more can be applied.

```
ho_reduce Rew Refl InG OutG :- then (repeat (left_rew (ho_rew Rew)))
                                    Refl InG OutG.
```

Note that the order in which subterms are rewritten depends on the order in which instances of `C` and `X` are generated by λProlog. The only control provided to the user is whether unification performs the imitation operation before the projection operation (as in Huet's procedure [14]), or vice versa. Thus, such a tactic is of little use if finer control over the order in which subterms are rewritten is important. For strongly normalizing rewrite systems, the order often does

not matter. In this case, the tactic represents a compact implementation of a complete rewrite strategy, and for first-order rewriting, provides an alternative to the `bu` and `lo` procedures in the previous section.

As an example, consider the term $(tl\ (cons\ 0\ (cons\ (hd\ (cons\ 0\ nil))\ nil)))$ in the first-order fragment of the functional language presented in Section 2. If `rew` is a tactic implementing all of the rewrite rules of this example, then applying `(ho_reduce rew refl_tm)` with imitation before projection will perform the following series of rewrites:

$$(tl\ (cons\ 0\ (cons\ (hd\ (cons\ 0\ nil))\ nil))) \\ \rightarrow (tl\ (cons\ 0\ (cons\ 0\ nil))) \rightarrow (cons\ 0\ nil)$$

In the first rewrite step, the instances of `C` and `X` are $\lambda w.(tl\ (cons\ 0\ (cons\ w\ nil)))$ and $(hd\ (cons\ 0\ nil))$, respectively. Projection before imitation performs the two rewrite steps in the opposite order.

$$(tl\ (cons\ 0\ (cons\ (hd\ (cons\ 0\ nil))\ nil))) \\ \rightarrow (cons\ (hd\ (cons\ 0\ nil))\ nil) \rightarrow (cons\ 0\ nil)$$

To generalize this technique to first-order rewrite systems of more than one primitive type, we simply include an `ho_sub` tactic for every possible type of `C`. For example, with two types `t1` and `t2`, up to four tactics may be needed for the types `t1 -> t1`, `t1 -> t2`, `t2 -> t2`, and `t2 -> t1`.

To see why this technique is not adequate for higher-order rewriting, consider the term $(abs\ \lambda x.(abs\ \lambda y.(tl\ (cons\ x\ (cons\ (hd\ (cons\ y\ nil))\ nil)))))$. There are two instances of `X` in `ho_sub` which are abstractions over terms that can be rewritten by a rewrite rule. One such instance, for example is $\lambda z.(hd\ (cons\ z\ nil))$. The corresponding instance of `C` is

$$\lambda w.(abs\ \lambda x.(abs\ \lambda y.(tl\ (cons\ x\ (cons\ wy\ nil))))).$$

(Note that these instances of `C` and `X` correspond to the context and left closure of Definition 1 in Section 2.) However, there is no instance of `X` at primitive type `tm` to which a rewrite rule can be applied. In fact, universal quantification at the meta-level as it is used in congruence rules is essential for descending past abstractions so that rewriting can be performed on subterms. Here, the term $(tl\ (cons\ c_1\ (cons\ (hd\ (cons\ c_2\ nil))\ nil)))$ can be obtained by two applications of the congruence rule for *abs*, where c_1 and c_2 are the new constants generated by the GENERIC operation. Note that this term is first-order and so it is possible to apply `ho_reduce` at this point to obtain the reduced term $(cons\ c_2\ nil)$. So, by the tactic:

```
then (then (untree cong_abs) (untree cong_abs)) (ho_reduce rew refl_tm)
```

the original term rewrites to $(abs\ \lambda x.(abs\ \lambda y.(cons\ y\ nil)))$. Thus, although this method is not adequate for general higher-order rewriting, it can be employed for rewriting first-order subsets of higher-order rewrite systems.

8 Conclusion and Related Work

The Isabelle theorem prover [24] also contains a specification language which is essentially hh^ω. Thus higher-order rewrite systems could be expressed in much the same way there. In Isabelle, however, the language used to specify inference rules is distinct from that used to implement tacticals and tactics, namely ML. Here, the same language is used for both specification and implementation. It is the interpreter described in Section 4 that provides control by giving operational interpretation directly to the logical connectives.

Although we use the same meta-language as Nipkow [23] to define higher-order rewrite rules, both the definition of the rewrite relation and the notion of equality modulo an HRS are defined differently here. One difference, for example, is in the definition of the rewrite relation. Instead of using contexts with an abstraction indicating where in a term a rewrite is performed, Nipkow uses positions in first-order abstract trees. This notion of position is similar to the one often used in defining first-order rewriting (as in Dershowitz [5] for example). Despite these differences, both the notions of rewrite relation and equivalence modulo a rewrite system are equivalent to those given here for terms and rewrite rules at primitive type.

In Nipkow [23], the notion of critical pair is extended to higher-order rewrite systems, and it is shown that the critical pair lemma extends to the higher-order case. This result gives rise to a procedure for completion. Higher-order logic programming should provide a suitable framework for implementing such completion procedures.

In Pfenning [26], the notion of higher-order patterns is extended to the dependent-type λ-calculus LF, and to the Calculus of Constructions (CC). It should be straightforward to adopt LF as the meta-language for expressing rewrite systems and extend the notion of higher-order rewriting accordingly. While the restriction to primitive types extends naturally to LF, in CC it does not since quantification over types is permitted. Despite this fact, it would be interesting to study ways in which CC could be adopted as a meta-language for rewriting.

It is also possible to encode terms of CC in the simply typed λ-calculus, and to specify and implement provability for this calculus in hh^ω [9]. Using this encoding, notions of convertibility of CC terms could be expressed as higher-order rewrite systems as defined here, and incorporated into such an implementation.

Another direction of study in higher-order rewriting has been to consider the interaction of first-order rewrite systems with reduction rules for various typed λ-calculi (Breazu-Tannen and Gallier [2] and Jouannaud and Okada [15] for example). In this work, no distinction between the object and meta-level is made, but properties about the hybrid systems are studied. The rewrite systems studied there could in fact be encoded as rewrite systems in the setting defined here and thus implemented directly. Known properties of these systems would provide knowledge about the corresponding implementations. For example, from results shown by Breazu-Tannen and Gallier, we know that we can add the *app*

and *abs* constants and the rewrite rule for β-conversion given in Section 2 to any terminating first-order rewrite system and obtain a terminating rewrite system.

Acknowledgements

The author would like to thank Dale Miller and the reviewers for helpful comments on this work. This research was supported in part by ESPRIT Basic Research Action 3245 "Logical Frameworks: Design, Implementation, and Experiment."

References

1. Peter Aczel. A general church-rosser theorem. Technical report, University of Manchester, 1978.
2. Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1):3–28, 1991.
3. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
4. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.
5. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier-MIT Press, 1989.
6. Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, August 1989.
7. Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks.* Cambridge University Press, 1991.
8. Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, Argonne, IL, May 1988.
9. Amy Felty and Dale Miller. A meta language for type checking and inference: An extended abstract. Presented at the 1989 Workshop on Programming Logic, Bålstad, Sweden, 1989.
10. Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.
11. Mike Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, July 1985.
12. John Hannan and Dale Miller. Enriching a meta-language with higher-order features. In *Workshop on Meta-Programming in Logic Programming*, Bristol, June 1988.
13. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

14. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

15. Jean-Pierre Jouannaud and Mitsuhiro Okada. A computation model for executable higher-order algebraic specification languages. In *Sixth Annual Symposium on Logic in Computer Science*, pages 350–361, Amsterdam, July 1991.

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

17. J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1991.

18. Dale Miller. Abstract syntax and logic programming. In *Proceedings of the Second Russian Conference on Logic Programming*. Springer-Verlag LNAI series, September 1991. To appear.

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

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

21. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

22. Tobias Nipkow. Equational reasoning in Isabelle. *Science of Computer Programming*, 12:123–149, 1989.

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

24. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.

25. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

26. Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Sixth Annual Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, July 1991.

27. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

A Equivalence of Operational and Logical Rewriting

Lemma 2 Let H be a higher-order rewrite system. Let M and N be terms of type $\tau_1 \to \tau_2$ and x a variable of type τ_1 not free in M or N. Then:

1. $M \to_H N$ if and only if $Mx \to_H Nx$.
2. $M \overset{*}{\leftrightarrow}_H N$ if and only if $Mx \overset{*}{\leftrightarrow}_H Nx$.

Proof: We prove (1). The proof of (2) then follows by induction on the number of rewrite steps. We begin with the forward direction and assume that $M \to_H N$. Then there is a context u and left and right closures L and R as in Definition 1 such that $M =_{\beta\eta} uL$ and $N =_{\beta\eta} uR$. Let u' be the term $\lambda w.uwx$ where w is a variable distinct from x and not free in u. Then $Mx =_{\beta\eta} uLx =_{\beta\eta} u'L$ and $Nx =_{\beta\eta} uRx =_{\beta\eta} u'R$. Thus, by definition with context u', we have $Mx \to_H Nx$.

Next, suppose $Mx \to_H Nx$. Then there is a context v and left and right closures L and R as in Definition 1 such such that $Mx =_{\beta\eta} vL$ and $Nx =_{\beta\eta} vR$. Let v' be the term with bound variable x and body v, and v'' be the term $\lambda w.\lambda z.v'zw$ for variables z, w not free in v'. Then $v''Lx =_{\beta\eta} v'xL =_{\beta\eta} vL =_{\beta\eta} Mx$ and similarly $v''Rx =_{\beta\eta} v'xR =_{\beta\eta} vR =_{\beta\eta} Nx$. Since x does not occur free in M, N, or v'', we have $v''L =_{\beta\eta} M$ and $v''R =_{\beta\eta} N$. Thus, by definition with context v'', $M \to_H N$.

Lemma 3 Let H be a higher-order rewrite system. Let M and N be terms of type τ_1 and P and Q terms of type $\tau_1 \to \tau_2$.

1. If $M \to_H N$, then $PM \to_H PN$.
2. If $P \to_H Q$, then $PM \to_H QM$.
3. If $P \overset{*}{\leftrightarrow}_H Q$ and $M \overset{*}{\leftrightarrow}_H N$, then $PM \overset{*}{\leftrightarrow}_H QN$.

Proof: To prove (1), we know that if $M \to_H N$, then there is a context u and left and right closures L and R as in Definition 1 such that $M =_{\beta\eta} uL$ and $N =_{\beta\eta} uR$. Let u' be the term $\lambda x.P(ux)$ where x is a variable not free in P or u. Then $u'L =_{\beta\eta} P(uL) =_{\beta\eta} PM$ and $u'R =_{\beta\eta} P(uR) =_{\beta\eta} PN$, and thus by definition with context u', $PM \to_H PN$. The proof of (2) is similar. (3) is proved by induction on the sum of the number of rewrite steps in $P \overset{*}{\leftrightarrow}_H Q$ and $M \overset{*}{\leftrightarrow}_H N$, and follows easily from (1) and (2).

We now define a specialized form of the rewriting relation and prove that the relation defined in Section 2 is contained in the transitive closure of this new relation. Although it is not essential, this relation simplifies the proof of Theorem 7.

Definition 4 Given an HRS H, we define the relation $\rightharpoonup_H$ on terms as follows: $M \rightharpoonup_H N$ if there is a context u and left and right closures L and R as in Definition 1 such that there is at most one free occurrence of the outermost bound variable in the body of the $\beta\eta$-long form of u.

We write $\stackrel{*}{\rightleftharpoons}_H$ to denote the reflexive, symmetric, transitive closure of this relation. It is easy to see that Lemma 3 also holds for $\rightharpoonup_H$ and $\stackrel{*}{\rightleftharpoons}_H$.

Lemma 5 Given HRS H and terms M, N, if $M \rightarrow_H N$, then $M \stackrel{*}{\rightleftharpoons}_H N$.

Proof: Since $M \rightarrow_H N$, there is a context u and left and right closures L and R as in Definition 1 such that $M =_{\beta\eta} uL$ and $N =_{\beta\eta} uR$. The proof is by induction on m, the number of free occurrences of the bound variable at the head of u in the body of u. If m is 0, then $M =_{\beta\eta} N$. For the case when $m > 0$, we can express u as an abstraction with bound variable x and body $[x/w]v$ for some variable w and term v that has exactly one free occurrence of w. Let u' be the term with binder $\lambda w.\lambda x.$ and body v. Thus $M =_{\beta\eta} uL =_{\beta\eta} u'LL$. By definition, with context $u'L$, we know that $u'LL \rightarrow_H u'LR$. Context $u'L$ has one fewer free occurrence of the bound variable at its head in its body than does u, so, by the induction hypothesis $u'LL \stackrel{*}{\rightleftharpoons}_H u'LR$. By definition, with context u', we know that $u'L \rightharpoonup_H u'R$. By Lemma 3, we have $u'LR \rightharpoonup_H u'RR$. Thus $u'LL \stackrel{*}{\rightleftharpoons}_H u'RR$ and we have our result.

Lemma 6 Let H be an HRS, and let M and N be terms of type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $n \geq 0$ and τ_0 is primitive. Let $x_1, \ldots, x_n$ be n distinct variables of type $\tau_1, \ldots, \tau_n$, respectively, not free in M or N. If $M \rightharpoonup_H N$, then $\vdash_H \forall \overline{x_n}(M\overline{x_n} = N\overline{x_n})$.

Proof: By Definition 4, there are a rule $l \rightarrow r$ in H, a substitution σ, variables $z_1, \ldots, z_m$, and a context u, such that (1) L and R are closures with binder $\lambda\overline{z_m}$ and bodies σl and σr, respectively, (2) there is at most one free occurrence of the outermost bound variable in the body of the $\beta\eta$-long form of u, and (3) $M =_{\beta\eta} uL$ and $N =_{\beta\eta} uR$. We show by induction on the structure of the $\beta\eta$-long form of u that $\vdash_H \forall \overline{x_n}(M\overline{x_n} = N\overline{x_n})$. The $\beta\eta$-long form of u has a binder of the form $\lambda w \lambda \overline{x_n}$ and body of form the $hP_1 \ldots P_p$ where h is a variable or constant and exactly one of the terms $h, P_1, \ldots, P_p$ contains exactly one occurrence of w.

We first consider the case where h is w. Then h, w, and L have the same type and $m = p$. Let σ' be the substitution that maps z_i to P_i for $i = 1, \ldots, m$, and let σ'' be the substitution that maps each element x in the domain of σ to $\sigma'(\sigma x)$. Then $M\overline{x_n} =_{\beta\eta} LP_1 \ldots P_n =_{\beta\eta} \sigma'(\sigma l) =_{\beta\eta} \sigma'' l$ and $N\overline{x_n} =_{\beta\eta} RP_1 \ldots P_n =_{\beta\eta} \sigma'(\sigma r) =_{\beta\eta} \sigma'' r$. The domain of σ'' is the free variables of l. Thus, by a series of $\forall$-E from the axiom corresponding to the rule $l \rightarrow r$, using σ'' as the substitution terms, we have $\vdash_H M\overline{x_n} = N\overline{x_n}$. Then, by a series of applications of $\forall$-I, we have our result.

We next consider the case where h and w are distinct. For $i = 1, \ldots, p$, let u_i be the term with bound variable w and body P_i. Note that u_i is smaller than u. (Also, note that since there is exactly one occurrence of w in u, all but one of the abstractions in $u_1, \ldots, u_p$ are vacuous.) Then $M\overline{x_n} =_{\beta\eta} uL\overline{x_n} =_{\beta\eta} h(u_1L)\ldots(u_pL)$ and $N\overline{x_n} =_{\beta\eta} uR\overline{x_n} =_{\beta\eta} h(u_1R)\ldots(u_pR)$. By definition, for $i = 1, \ldots, p$, using u_i as context, we know that $u_iL \rightharpoonup_H u_iR$. Let $\tau_1^i \to \cdots \to \tau_{m_i}^i \to \tau_0^i$ where τ_0^i is primitive be the type of u_iL, and $x_1, \ldots, x_{m_i}$ be m_i variables of type $\tau_1^i, \ldots, \tau_{m_i}^i$ that do not appear free in u_i. Thus, by the induction hypothesis, we can deduce $\forall\overline{x_{m_i}}(u_iL\overline{x_{m_i}} = u_iR\overline{x_{m_i}})$. By an application of (CONG) we have $h(u_1L)\ldots(u_pL) = h(u_1R)\ldots(u_pR)$. Then by n applications of $\forall$-I, we have our result.

Theorem 7 Let H be an HRS, and let M and N be terms of type $\tau_1 \to \cdots \to \tau_n \to \tau_0$ where $n \geq 0$ and τ_0 is primitive. Let $x_1, \ldots, x_n$ be n distinct variables of type $\tau_1, \ldots, \tau_n$, respectively, not free in M or N. Then $\vdash_H \forall\overline{x_n}(M\overline{x_n} = N\overline{x_n})$ if and only if $M \overset{*}{\leftrightarrow}_H N$.

Proof: We prove the forward direction by induction on the height of a proof tree. If the proof is a one node application of reflexivity, then n is 0 and M is N. Then, clearly $M \overset{*}{\leftrightarrow}_H N$. If $\forall\overline{x_n}(M\overline{x_n} = N\overline{x_n})$ is an axiom other than the reflexive axiom, then $M\overline{x_n} \to N\overline{x_n}$ is a rewrite rule. Thus $M\overline{x_n} \to_H N\overline{x_n}$ and by Lemma 2, $M \to_H N$.

If the last inference is $\forall$-E, then the premise has the form $\forall x\forall\overline{x_n}(M'x\overline{x_n} = N'x\overline{x_n})$ and the conclusion has the form $\forall\overline{x_n}(M't\overline{x_n} = N't\overline{x_n})$ for some substitution term t. By the induction hypothesis $M' \overset{*}{\leftrightarrow}_H N'$, and by Lemma 3 $M't \overset{*}{\leftrightarrow}_H N't$.

If the last inference is $\forall$-I, then the premise has the form $\forall\overline{x_n}(M'x\overline{x_n} = N'x\overline{x_n})$ and the conclusion has the form $\forall x\forall\overline{x_n}(M'x\overline{x_n} = N'x\overline{x_n})$. By the induction hypothesis $M'x \overset{*}{\leftrightarrow}_H N'x$, and by Lemma 2, $M' \overset{*}{\leftrightarrow}_H N'$. The case when the last rule application is (CONG) follows by the induction hypothesis and Lemma 3. The last two cases for symmetry and transitivity follow directly from the induction hypothesis.

Next, we assume that $M \overset{*}{\leftrightarrow}_H N$. By Lemma 5, we know that $M \overset{*}{\rightleftharpoons}_H N$. The proof is by induction on m, the number of rewrite steps using $\rightharpoonup_H$ to rewrite M to N. If m is 0, then $M =_{\beta\eta} N$. Thus $M\overline{x_n} =_{\beta\eta} N\overline{x_n}$ and the desired formula is provable by reflexivity followed by n applications of $\forall$-I. If $m > 0$, then there is a term P such that $M \overset{*}{\rightleftharpoons}_H P \rightharpoonup_H N$ or $M \overset{*}{\rightleftharpoons}_H P \leftharpoonup_H N$. By the induction hypothesis, $\vdash_H M = P$. By Lemma 6, either $\forall\overline{x_n}(P\overline{x_n} = N\overline{x_n})$ or $\forall\overline{x_n}(N\overline{x_n} = P\overline{x_n})$. By n applications of $\forall$-E, we obtain $P\overline{x_n} = N\overline{x_n}$ or $N\overline{x_n} = P\overline{x_n}$. In the latter case, by symmetry we obtain $P\overline{x_n} = N\overline{x_n}$. Then, in both cases, by transitivity followed by n applications of $\forall$-I, we get the desired result.