A META LANGUAGE FOR TYPE CHECKING AND INFERENCE
An Extended Abstract

Amy Felty and Dale Miller
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA

Abstract

We present a logic, L_λ, in which terms are the simply typed λ-terms and very restricted, second-order quantification of functional variables is allowed. This logic can be used to directly encode the basic type judgments of a wide variety of typed λ-calculi. Judgments such as "term M is of type A" and "M is a proof of formula A" are represented by atomic propositions in L_λ while inference rules and axioms for such type judgments become simple quantified formulas. Theorem proving in L_λ can be described simply since the necessary unification of λ-terms is decidable and if unifiers exist, most general unifiers exist. Standard logic programming techniques can turn the specification of inference rules and axioms in L_λ into implementations of type checkers and inferrers. Several different typed λ-calculi have been specified in L_λ and these specifications have been directly executed by a higher-order logic programming language. We illustrate such encoding into L_λ by presenting type checkers and inferrers for the simply typed λ-calculus and the calculus of constructions.

This extended abstract will be presented at the 1989 Workshop on Programming Logic, Bålstad, Sweden.

Comments are welcome. Address correspondence to the authors at the address above or at "felty@linc.cis.upenn.edu" or "dale@linc.cis.upenn.edu."

1 Introduction

The class of *higher-order hereditary Harrop* (hohh) formulas [MNS87, MNPS] has been investigated as foundation for a logic programming language. This class of formulas was initially developed to see how richly first-order Horn clauses could be extended while still maintaining many of the desirable properties of the logic programming paradigm. This extension has provided an approach to understanding the nature of such programming abstractions as higher-order programming, modules, and abstract data types within logic programming. These formulas have also been used as a meta programming language. For example, in [HM88a, HM88b, MN87, PE88], hohh was used to specify meta programming tasks in the manipulation of simple functional programs. In [Fel87, FM88, HM88a, Pfe88], hohh was used to specify several different theorem provers, type systems, and proof systems for various object logics. Furthermore, the prototype programming language λProlog [EP89, MN88, NM88] that has been built using hohh as a foundation was used in each of these papers to directly implement and test their respective specifications, thereby, providing program transformers, type checkers, and theorem provers.

Higher-order hereditary Harrop formulas extends first-order Horn clauses in the following ways:

- Quantification of predicate variables at certain occurrences within formulas is permitted.
- Unrestricted quantification of function variables at all (non-predicate) types is permitted.
- First-order terms are replaced by simply typed λ-terms at all types.
- Equality of λ-terms is determined up to $\beta\eta$-conversion.
- Universal quantifiers and implications are permitted (with certain restrictions) in the body of clauses and in goals.

While there is a theoretical setting in which all these extensions belong to a well motivated whole, a complete implementation of this logic yields an interpreter with very many diverse built in operations. In the case of the meta programming tasks mentioned above, many of those operations are not needed. In particular, predicate quantification plays a very small role and the occurrences and type of quantified function variables could often be greatly restricted.

Constructing a smaller logic programming language in which these features are removed is of interest for several reasons. First, removing predicate quantification makes the proof theoretic properties of the logic programming language much simpler to establish. More importantly, from the point of view of implementing logics, the restriction on occurrences of quantified functional variables in terms can greatly simplify the unification processes that need to be implemented. As we shall mention below, in our proposed sublogic of hohh, unification of simply typed λ-terms will be decidable and have most general unifiers: neither of these properties are true of unification in the full logic. While the unification of richer classes of simply typed λ-terms has several important uses [HM88b, HL78, MN87, Pfe88] it is not necessary for the kinds of implementations we shall describe here. Finally, since the

$$\frac{\mathcal{Q}\exists x \vdash_D L}{\mathcal{Q} \vdash_D \forall x\, L} \qquad \frac{\mathcal{Q} \vdash_D L_1 \qquad \mathcal{Q} \vdash_D L_2}{\mathcal{Q} \vdash_D L_1 \wedge L_2} \qquad \frac{\mathcal{Q} \vdash_G L_1 \qquad \mathcal{Q} \vdash_t L_2}{\mathcal{Q} \vdash_D L_1 \supset L_2} \qquad (1,2,3)$$

$$\frac{\mathcal{Q}\forall x \vdash_G L}{\mathcal{Q} \vdash_G \forall x\, L} \qquad \frac{\mathcal{Q} \vdash_G L_1 \qquad \mathcal{Q} \vdash_G L_2}{\mathcal{Q} \vdash_G L_1 \wedge L_2} \qquad \frac{\mathcal{Q} \vdash_D L_1 \qquad \mathcal{Q} \vdash_G L_2}{\mathcal{Q} \vdash_G L_1 \supset L_2} \qquad (4,5,6)$$

$$\frac{\mathcal{Q} \vdash_t L}{\mathcal{Q} \vdash_D L} \qquad \frac{\mathcal{Q} \vdash_t L}{\mathcal{Q} \vdash_G L} \qquad (7,8)$$

$$\frac{\mathcal{Q}\forall x \vdash_t L}{\mathcal{Q} \vdash_t \lambda x\, L} \qquad \frac{\mathcal{Q} \vdash_t t_1 \qquad \cdots \qquad \mathcal{Q} \vdash_t t_n}{\mathcal{Q} \vdash_t (y t_1 \ldots t_n)} \qquad \mathcal{Q} \vdash_t (x y_1 \ldots y_n) \qquad (9,10,11)$$

Figure 1: Inference Rules for Defining the Syntax of L_λ.

sublanguage is syntactically defined, these restrictions can be used by compilers of the full language to provide for special compilation of the sublanguage.

2 A Logic for Encoding λ-Calculi

The types of L_λ are the primitive types, among which is the type of propositions o, and all functional types built from these primitive types. The order of a type is defined as: $ord(\tau) = 0$ if τ is primitive and $ord(\tau_1 \to \ldots \to \tau_n \to \tau) = 1 + \max_i ord(\tau_i)$ where $n \geq 1$ and τ is primitive.

The logic has the logical connectives $\supset$ (implication) and $\wedge$ (conjunction) both of type $o \to o \to o$ and, for every type τ not containing o, $\forall_\tau$ of type $(\tau \to o) \to o$ (universal quantification). A signature, Σ, is a list $\langle c_1 : \tau_1, \ldots, c_n : \tau_n \rangle$ $(n \geq 1)$ of non-logical constants with their type such that whenever $c_i = c_j$ then $\tau_i = \tau_j$. The order of Σ is $ord(\Sigma) = \max_i ord(\tau_i)$. In this abstract we shall assume that $ord(\Sigma) \leq 2$ for all signatures Σ. A constant of type $\tau_1 \to \ldots \to \tau_n \to o$ $(n \geq 0)$ is called a predicate constant.

Let Σ be a signature. We define the three binary relations $\vdash_t, \vdash_D, \vdash_G$ that relate a quantifier prefix $\mathcal{Q}$ to a formula or term L in β-normal form such that the nonlogical constants in L are members of Σ and its free variables are bound in $\mathcal{Q}$. The inference rules in Figure 1 define the extension of these three relations. The provisos on these inference rules are listed as follows: in $(1, 4, 9)$ the variable x must not appear in the prefix $\mathcal{Q}$; in $(7, 8)$ the formula L must be of propositional type o; in (10) y must be either a constant (in Σ) or a variable universally quantified in $\mathcal{Q}$; and in (11) x must be a variable existentially quantified in $\mathcal{Q}$ and $y_1, \ldots, y_n$ must be distinct universally bound variables of $\mathcal{Q}$ such that they are bound to the right of the binding occurrence of x in $\mathcal{Q}$.

Let $\emptyset$ be the empty quantifier prefix. We define $L_\lambda(\Sigma) := \langle \mathcal{D}, \mathcal{G} \rangle$ where $\mathcal{D}$ is the set of β-normal formulas L such that $\emptyset \vdash_D L$ and $\mathcal{G}$ is the set of β-normal formulas L such that $\emptyset \vdash_G L$. Members of $\mathcal{D}$ are called *definite* formulas while members of $\mathcal{G}$ are called *goals*. The provability relation that will interest us with respect to $L_\lambda(\Sigma)$ is whether or

not $\mathcal{P} \vdash_I G$, where $\mathcal{P}$ is a finite subset of $\mathcal{D}$, $G \in \mathcal{G}$ is closed, and $\vdash_I$ denotes intuitionistic provability. Under the assumption that Σ contains "enough constants," that is, there is a closed Σ-term of all primitive types mentioned in Σ, then it can be shown that provability in this sense can be characterized by cut-free sequential proofs such that all occurrences of sequents contain antecedents that are finite subsets of $\mathcal{D}$ and succedents are singleton sets containing a formula from $\mathcal{G}$. In other words, even though the formulas of L_λ are restricted in a very strong fashion, it is possible to develop a complete proof system which needs to only consider formulas within this restricted class. Cut-free proofs can also be further restricted to being simply the *uniform proofs* described in [MNPS] and, as a consequence, L_λ can be viewed as the foundation of a logic programming language.

The force of the restrictions to the L_λ syntax is contained in the inference rule (11) of Figure 1. Variables that are existentially quantified in the prefixes that are on the left of the turnstile relations in Figure 1 correspond to those variables that will be substituted for in the course of building a proof. In other words, they correspond to "logical variables" in the jargon of logic programming implementations. Rule (11) greatly restricts the occurrences of such functional variables. The result of this restriction is that in the construction of proofs, only very simple β-redexes occur: if t is substituted for x into an occurrence of the form $(xy_1 \dots y_n)$ the result is simply $(ty_1 \dots y_n)$. Since x was quantified to the left of where the variables $y_1, \dots, y_n$ were universally quantified, the term t contains no free occurrences of the variables $y_1, \dots, y_n$. Thus, t can be taken (via α- and η-conversion) to be of the form $\lambda y_1 \dots \lambda y_n\, t'$. Thus, the β-reduction of the redex $(ty_1 \dots y_n)$ would simply yield the term t'. If t had been normal then this simple operation of essentially dropping the λ-abstractions on t would immediately yield the β-normal formula t'.

Since β-conversion for this logic is very simple, the unification problems that would arise in a theorem prover for this logic are also very simple. In particular, we state the following (these are corollaries of theorems found in [Mil88a, Mil88b]):

1. Unification is decidable: Given two (possibly open) atoms of $L_\lambda(\Sigma)$, A_1 and A_2, it is decidable whether or not there exists a substitution θ such that θA_1 is $\beta\eta$-convertible to θA_2.

2. When unifiers exist, most general unifiers (mgu) exist and can be effectively computed.

In [MNPS], it was shown that theorem proving for hohh (and thus also for L_λ) with respect to higher-order intuitionistic logic can be accomplished by a simple goal-directed search, similar to that considered in logic programming. The choice of a depth-first or breadth-first interpreter provides either an incomplete or complete theorem prover. As we shall see, an incomplete theorem prover can still provide complete implementations in several applications.

The higher-order logic programming language λProlog can be used to implement L_λ. The LP2.7 and eLP implementations of this language [EP89, MN88] are depth-first interpreters. Their unification packages contains extensions to Huet's unification process [Hue75] that are necessary to have mgus computed [Mil88b].

3 Specifying the Untyped λ-Calculus

In this section we encode both the untyped λ-terms and certain operations on them into L_λ. For the presentation of our examples in this section and the rest of this abstract, we shall use the syntax of λProlog to present examples. Fortunately, very little of the λProlog syntax is needed here. In particular, a signature member, say $f : a \to b \to c$, is represented as simply the line

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

The symbol `->` is right associative. As an example, we need the following signature, named UT, in this section:

```
type   abs        (tm -> tm) -> tm.
type   app        tm -> tm -> tm.
type   copy       tm -> tm -> o.
type   subst      tm -> (tm -> tm) -> tm -> o.
type   redex      tm -> tm -> o.
type   red1       tm -> tm -> o.
type   conv       tm -> tm -> o.
```

In this signature, only two primitive types appear: `tm` for the encoding of untyped λ-terms and `o` for propositions of $L_\lambda(UT)$.

The simply typed terms are represented by having application be juxtaposition (associating to the left) and by using a back slash as an infix operator denoting λ-abstractions. Tokens with initial capital letters will denote either bound or free variables. All other tokens will denote constants (these generally will also be declared in a signature prior to their appearing in formulas). Using the two constants, `abs` and `app` above, all closed untyped λ-terms can be encoded. For example, the combinators I, K, S, Y, are encoded as the following closed terms of type `tm`:

```
(abs X\X)
(abs X\(abs Y\X))
(abs X\(abs Y\(abs Z\(app (app X Z) (app Y Z)))))
(abs X\(app (abs Y\(app X (app Y Y))) (abs Y\(app X (app Y Y)))))
```

This encoding is essentially the one used by Meyer in [Mey81]: `abs` corresponds to the function Ψ, for coercing functions into terms, and `app` corresponds to the function Φ for coercing terms into functions.

In representing formulas of L_λ, we use the following syntax. The definite formula $G \supset D$ will be written as `D :- G.` (the definite formulas $G_1 \supset G_2 \supset D$ and $(G_1 \wedge G_2) \supset D$ as `D :- G1, G2.`) while the goal formula $D \supset G$ will be written as `D => G`. The goal formula $\forall_\tau x G$ will be written as `pi X\G` (`pi` denotes $\forall$) while the corresponding definite formula $\forall_\tau x D$ will be written as simply `D`. Here, the free variables of `D` will be implicitly quantified around `D`. In the examples in this abstract, the type τ can always be recovered from context and, hence, are not inserted into formulas.

The specification of the substitution of one term for an abstracted variable of another term can be easily specified using the following code:

```
copy (app M N) (app P Q) :- copy M P, copy N Q.
copy (abs M) (abs N) :- pi X\ (copy X X => copy (M X) (N X)).

subst T M N :- pi X\ (copy X T => copy (M X) N).
```

The closed atom `(subst M P N)` is provable from these three clauses if and only if `M` and `N` denote closed untyped λ-terms, `P` is an abstraction over such a term, and `N` is the result of substituting `M` for the abstracted variable in `P`. In an operational sense, `subst` performs the following operation: first, the `pi` quantifier generates a new constant, say `c`, then assumes `(copy c T)`, that is, when recursive calls to `copy` see `c`, it should be replaced by `T`. The goal `(copy (M c) N)` is then called. Thus, the bound variable of `M` is replaced by the constant `c` and then the resulting term, that is `(M c)`, is copied into `N`. The result of this copying is, of course, the result of substituting `T` for the bound variable of `M` into the body of `M`. This operational reading of `subst` over commits to the way information can "flow" in this program. It is also possible to do generalization, in the sense that given closed terms for the first and third argument of `subst`, the second argument can be found. For example, the open formula

```
subst (abs X\X) M (abs Y\(app (app Y (abs X\X)) (abs X\X)))
```

is provable if `M` is instantiated with any one of the following terms:

```
X\(abs Y\(app (app Y X) X))
X\(abs Y\(app (app Y X) (abs Z\Z)))
X\(abs Y\(app (app Y (abs Z\Z)) X))
X\(abs Y\(app (app Y (abs Z\Z)) (abs Z\Z))).
```

Given `subst`, it is easy to specify $\beta\eta$-convertibility of the object-level -terms, that is, of the untyped λ-terms. The predicate `redex` specifies to to reduce β-redexes and η-redexes.

```
redex (app (abs N) M)   P :- subst M N P.
redex (abs X\(app M X)) M.
```

The predicate `red1` relates two λ-terms if one arises from the other by replacing exactly one redex.

```
red1 M        N          :- redex M N.
red1 (abs M)   (abs N)   :- pi X\(copy X X => red1 (M X) (N X)).
red1 (app M N) (app P N) :- red1 M P.
red1 (app M N) (app M P) :- red1 N P.
```

Finally, `conv` is the reflexive, symmetric, and transitive closure of `red1`.

```
conv M M.
conv M N :- conv N M.
conv M N :- conv M P, conv P N.
conv M N :- red1 M N.
```

These five operations, `copy`, `subst`, `redex`, `red1`, and `conv` follow easily from simply considering the above signature for the untyped λ-terms. In the rest of this abstract, we consider encoding typed λ-calculi. In each case, the meta logic will contain the types `tm` to encode terms and `ty` to encode types. In some cases, an additional type `ki` will be used to encode kinds. Depending on the kinds of term, type, and kind abstractions in a λ-calculus, it might be necessary to write a `subst`-style operator for, say terms into terms, types into types, types into terms, etc. In each case, the necessary substitution operation on the encoded structures can be written as described above. In general, if a language contained primitive types `a` and `b`, then it would be straightforward to write predicates of the following kind:

```
type   copy_aa       a -> a -> o.
type   copy_bb       b -> b -> o.
type   subst_aa      a -> (a -> a) -> a -> o.
type   subst_ab      a -> (a -> b) -> b -> o.
type   subst_ba      b -> (b -> a) -> a -> o.
type   subst_bb      b -> (b -> b) -> b -> o.
type   redex_a       a -> a -> o.
type   redex_b       b -> b -> o.
type   red1_a        a -> a -> o.
type   red1_b        b -> b -> o.
type   conv_a        a -> a -> o.
type   conv_b        b -> b -> o.
```

Not all of these predicates might be necessary. For example, there might not be any signature items that allow the construction of a term of type `b` that contains a subterm of type `a`. In such a case, `subst_ab` would not be needed. The actual formulas to specify these predicates is easily derived from the signature specifying the enoding of object level terms, types, and kinds.

4 Encoding Typed λ-Terms and Their Proof Systems

In this section, we show how terms and types, and sometimes kinds, of a λ-calculus can be encoded as simply typed λ-terms, and illustrate how the axioms and inference rules of a proof system for type checking and inference in the calculus are specified as formulas of L_λ. We first consider the simply typed λ-calculus (ST) as an object language. Since the object language and the terms of the meta language are the same, we distinguish between them by calling the types and terms of L_λ meta types and meta terms.

We will need the meta types `ty` and `tm` to represent the types and terms, respectively, of ST. The constructors for terms and types are:

```
type    -->     ty -> ty -> ty.
type    abs     (tm -> tm) -> ty -> tm.
type    app     tm -> tm -> tm.
```

The arrow (`-->`) will be used to construct functional types in the usual way. As in Section 2, abstraction in terms will be represented using λ-abstraction in the meta language. Here we also include the type of the bound variable as an argument to `abs`. The constant `app` is the application operator. For example, using this syntax, the term $\lambda f : i \to i \lambda n : i.(f(fn))$ is represented as

```
(abs F\ (abs N\ (app F (app F N)) i) (i --> i)).
```

To specify type checking and inference for ST in L_λ, we introduce the following `has_type` predicate that takes an ST term and type as arguments:

```
type    has_type        tm -> ty -> o.
```

with the intended meaning that the atomic proposition `(has_type M A)` is provable when the term `M` has type `A`. We adopt the convention that the letters `M` and `N` will denote terms, and `A` and `B` will denote types of the object language.

Let ST be the name of the signature of four constants listed above. Then type checking and inference in ST is specified by the two $L_\lambda(ST)$ formulas below. These formulas can be considered an encoding of a type assignment proof system for ST similar to those found in [HS86, DM82].

```
has_type (abs M A) (A --> B) :-
  pi X\(has_type X A => has_type (M X) B).

has_type (app M N) B :-
  has_type M (A --> B),
  has_type N A.
```

The first formula encodes the fact that an abstraction `(abs M A)` has functional type `A --> B` if for arbitrary term `X`, `X` has type `A` implies that `(M X)` has type `B`. This clause would instruct a logic programming interpreter to first extend the current signature with a new constant, say `c`, then extend the definition of `has_type` with the clause `(has_type c A)`, and then attempt to prove that the β-normal form of `(M c)`, the result of replacing the bound variable of `M` with the name `c`, has type `B`. The additions to the signature and program are, of course, local modifications: a correct implemenation of L_λ would need to reset both the signature and the program when this goal is finished.

The second formula specifies the usual rule for application. An application `(app M N)` has type `B` if `M` has functional type `(A --> B)` and `N` has type `A`.

The logic program given by these formulas is complete for closed λ-terms in performing either type checking or type inference, even with respect to a depth-first interpreter. This fact follows by simple induction on the structure of terms of type `tm`. If the term is an

abstraction or application, one of the two clauses above is chosen, and the type checking or inference proceeds on the subterms. The only atomic terms are those that are introduced into the program by the rule for abstraction. It is also possible, in querying this logic program, to place free variables of type `ty` into terms. In the process of determining a type for such a term, the free `ty` variables will get instantiated appropriately. Even when such type information is lacking, a depth-first interpretation of these formulas will be complete for type inference.

Since the terms of L_λ are the simply typed λ-terms, we could have taken advantage of the meta language more directly in specifying type checking and inference for ST. Our goal, though, is to present a general method for encoding λ-calculi and their proof systems in L_λ. The method described above for ST does in fact generalize, and can be applied to many other systems including much richer λ-calculi. For another example, we show how to specify type inference and checking for the calculus of constructions (CC) [CH88, Hue87].

Again, the syntactic categories `tm` and `ty` are needed. The category `tm` will be for objects at the level of proofs or terms in CC and `ty` will be for CC propositions that specify the types of proofs. In addition, we introduce the meta type `ki` for the types of propositions, obtained by quantifying over the CC constant *Prop*, which we call kinds.

In our encoding of CC, there are four abstraction operators, four product operators, and four application operators. These operators are given by the following signature.

```
type    prop            ki.
type    prod_tyki       (ty -> ki) -> ki -> ki.
type    prod_tmki       (tm -> ki) -> ty -> ki.
type    prod_tyty       (ty -> ty) -> ki -> ty.
type    prod_tmty       (tm -> ty) -> ty -> ty.
type    abs_tyty        (ty -> ty) -> ki -> ty.
type    abs_tmty        (tm -> ty) -> ty -> ty.
type    abs_tytm        (ty -> tm) -> ki -> tm.
type    abs_tmtm        (tm -> tm) -> ty -> tm.
type    app_tyty        ty -> ty -> ty.
type    app_tytm        ty -> tm -> ty.
type    app_tmty        tm -> ty -> tm.
type    app_tmtm        tm -> tm -> tm.
```

The abstraction, product, and application operators are named according to the syntactic category of the objects that are abstracted or applied in each case. The `abs_tmtm` operator is analogous to `abs` in ST, and `app_tmtm` is analogous to `app`. The constant `prod_tmty` is a generalization of the arrow `-->` in ST: the ST type `A --> B` is the CC type `(prod_tmty X\B A)`. In addition, we introduce the constant `prop` to denote the CC constant *Prop*.

Using the above syntax, the polymorphic identity function $[A : Prop][x : A]x$ which has type $(A : Prop)(x : A)A$ is represented as the term `(abs_tytm A\(abs_tmtm X\X A) prop)`. The object level type of this term is represented by the term `(prod_tyty A\(prod_tmty X\A A) prop)`.

We encode the proof system for CC that appears in [Hue87] by introducing the three predicates:

```
type    has_type        tm -> ty -> o.
type    has_kind        ty -> ki -> o.
type    is_CCType       ki -> o.
```

As in ST, the atomic proposition `(has_type M A)` encodes the assertion that `M` has type `A`. Similarly `(has_kind A K)` asserts that type `A` has kind `K`. We will use the letters `K` and `L` to denote kinds. Finally, by `(is_CCType K)`, we mean that `K` belongs to the class of objects given by the constant *Type*, which in CC denotes the "type" of all kinds. We use `CCType` here to avoid confusion with the use of the word "type" to mean an object of syntactic category `ty`. Let CC denote the signature containing the above 16 constants.

The abstraction rule for terms in CC is given by the $L_\lambda(CC)$ formula:

```
has_type (abs_tmtm M A) (prod_tmty B A) :-
  pi X\(has_type X A => has_type (M X) (B X)),
  pi X\(has_type X A => has_kind (B X) prop),
  has_kind A prop.
```

In the first subgoal of this formula, as in the abstraction rule for ST, `pi` and `=>` are used to introduce a new constant and assumption. Since abstractions are allowed in types, the new constant `c` may appear in `(B c)` as well as in `(M c)`. The last two subgoals insure that `A` and `B` are well-formed types. The other three abstraction rules are similar:

```
has_type (abs_tytm M K) (prod_tyty A K) :-
  pi X\(has_kind X K => has_type (M X) (A X)),
  pi X\(has_kind X K => has_kind (A X) prop),
  is_CCType K.

has_kind (abs_tmty A B) (prod_tmki K B) :-
  pi X\(has_type X B => has_kind (A X) (K X)).
  pi X\(has_type X B => is_CCType (K X)),
  has_kind B prop.

has_kind (abs_tyty A K) (prod_tyki L K) :-
  pi X\(has_kind X K => has_kind (A X) (L X)),
  pi X\(has_kind X K => is_CCType (L X)),
  is_CCType K.
```

There are also four rules for application. These rules will make use of a substitution operation as described in Section 3. For CC, we need the six substitution predicates `subst_tyki`, `subst_tmki`, `subst_tyty`, `subst_tmty`, `subst_tytm`, and `subst_tmtm` defined over the signature of CC. The application rule for CC terms is:

```
has_type (app_tmtm M N) C :-
  subst_tmty N B C,
  has_type M (prod_tmty B A),
  has_type N A.
```

The use of `subst_tmty` here determines that `C` is the type obtained by substituting the term `N` for the bound variable in `B`, an abstraction from terms to types. Operationally, there may be many solutions for `B`. In a depth-first interpreter, backtracking may be necessary to find the one such that `M` has type `(prod_tmty B A)`. Also, as in ST, `N` must have type `A`. The other three application rules are similar:

```
has_type (app_tmty M A) C :-
  subst_tyty A B C, has_type M (prod_tyty B K), has_kind A K.

has_kind (app_tytm A M) L :-
  subst_tmki M K L, has_kind A (prod_tmki K B), has_type M B.

has_kind (app_tyty A B) L :-
  subst_tyki B K L, has_kind A (prod_tyki K P), has_kind B P.
```

The remaining CC rules involve kind and type formation. They are:

```
is_CCType prop.

is_CCType (prod_tyki K L) :-
  pi X\(has_kind X L => is_CCType (K X)), is_CCType L.

is_CCType (prod_tmki K A) :-
  pi X\(has_type X A => is_CCType (K X)), has_kind A prop.

has_kind (prod_tyty A K) prop :-
  pi X\(has_kind X K => has_kind (A X) prop), is_CCType K.

has_kind (prod_tmty A B) prop :-
  pi X\(has_type X B => has_kind (A X) prop), has_kind B prop.
```

Various subsystems of CC are obtained by allowing rules of equality between types and kinds. For example, if types and kinds are to be considered equal up to $\beta\eta$-conversion, we can use a convertibility operation as presented in Section 2. For CC, this operation requires three `conv` predicates for the three syntactic categories: `conv_ki`, `conv_ty`, and `conv_tm`. We will also need three `red1` and two `redex` predicates. (There are no redexes for kinds.) All of these operations will be defined by recursion over the signature of CC. Below are the inference rules for type and kind equality that make use of the conversion operation.

```
has_kind A K :- has_kind A L, conv_ki K L.
has_type M A :- has_type M B, conv_ty A B.
```

This completes the specification for CC.

In general, a type judgment in CC has the form $\Gamma \vdash M : A$ where Γ is an initial context. We can encode such contexts in several equivalent ways. One way is to add assumptions in the form of L_λ formulas. For example, let Γ be $[B : Prop][f : B \to B]$. This context corresponds to the assumptions:

```
has_kind b prop.
has_type f (prod_tmty X\b b).
```

Another possibility would be to encode the entire judgment $\Gamma \vdash M : A$ as the $L_\lambda(CC)$ formula:

```
pi B\(has_kind B prop => pi F\(has_type F (prod_tmty X\B B) => has_type M A)).
```

where `M` and `A` are the encoding of M and A into the L_λ abstract syntax for CC. In proving this formula, `pi` and `=>` are used to generate two new signature items for `B` and `F`, and assumptions are added about their types before attempting to prove `(has_type M A)`. A depth-first logic program interperter would not be able to handle the the last two rules above involving conversion: those two clauses are "left recursive" and cause any such interpreter to loop immediately. Hence, type checking and type inference would be incomplete with such rules. Alternatively, we could write this program to always convert CC objects to normal form before applying rules for kind and type checking. In such a program, the last two rules would not be needed, and using an inductive argument similar to the one used for ST, the program would be complete for type checking.

For CC, where terms are equated with proofs and types with formulas, in addition to type checking and type inference, we may want to do theorem proving, where the type or formula is specified and the proof or term must be discovered. For this task, a depth-first interpreter is of very limited use. Alternatively, we could implement inference rules as named tactics and then, using the tactic interpreter specified in [FM88], we could develop a tactic theorem prover for CC. Only a modest extension to L_λ would allow such tactic implementations.

5 Conclusion

We have illustrated the use of the language L_λ as a meta language for specifying type checking and inference for the object languages ST and CC. Of course, the techniques described here are very general and can be applied to many other typed λ-calculi. In particular, we have specified in L_λ and implemented in λProlog the second-order polymorphic λ-calculus, the ω-order polymorphic λ-calculus as presented in [Pfe88], and the logical framework [HHP87]. The specifications for these type systems are easily seen as simplifications of the implementation for CC given above.

For calculi with no abstractions over types, a more direct encoding of type judgments into L_λ is possible. For example, it is possible to translate typing judgments of ST and the logical framework directly into goal formulas of L_λ in such a way that the type judgment is valid if and only if the goal formula is intuitionistically provable. In such logics, it is not necessary to code the inference rules for `has_type`, for example, since the effects of these rules are already present in L_λ [Fel87].

Acknowledgements The authors would like to thank Thierry Coquand, Elsa Gunter, John Hannan, and Frank Pfenning for discussions regarding the work in this abstract. The

authors are supported in part by ARO grant DAA29-84-9-0027, NSF grant CCR-87-05596, DARPA grant N000-14-85-K-0018, and ONR grant N00014-88-K-0633.

References

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 207–212, 1982.

[EP89] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. May 1989.

[Fel87] Amy Felty. Implementing theorem provers in logic programming. November 1987. Dissertation Proposal, University of Pennsylvania, Technical Report MS-CIS-87-109.

[FM88] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, Argonne Ill., May 1988.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HM88a] John Hannan and Dale Miller. Enriching a meta-language with higher-order features. In *Workshop on Meta-Programming in Logic Programming*, Bristol, June 1988.

[HM88b] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue87] Gérard Huet. A uniform approach to type theory. In *Proceedings of the Institute on Logical Foundations of Functional Programming*, Austin, Texas, June 1987. To appear.

[Mey81] Albert Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, 1981.

[Mil88a] Dale Miller. Solutions to λ-term equations under a mixed prefix. September 1988. Unpublished draft.

[Mil88b] Dale Miller. Unification under a mixed prefix. September 1988. Unpublished draft.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[MN88] Dale Miller and Gopalan Nadathur. λProlog Version 2.7. July 1988. Distribution in C-Prolog and Quintus sources.

[MNPS] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. To appear in the Annals of Pure and Applied Logic.

[MNS87] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Lisp and Functional Programming Conference*, 1988.