Enriching A Meta-Language With Higher-Order Features

MS-CIS-88-45
LINC LAB 118

John Hannan
Dale Miller

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

June 1988

Acknowledgements:
This research was supported in part by NSF grants CCR-87-05596, MCS-8219196-CER, IRI84-10413-A02, DARPA grant N00014085-K-0018, and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027

ENRICHING A META-LANGUAGE WITH HIGHER-ORDER FEATURES *

Preliminary Report
June 1988

John Hannan† Dale Miller‡
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

ABSTRACT

Various meta-languages for the manipulation and specification of programs and programming languages have recently been proposed. We examine one such framework, called natural semantics, which was inspired by the work of G. Plotkin on operational semantics and extended by G. Kahn and others at INRIA. Natural semantics makes use of a first-order meta-language which represents programs as first-order tree structures and reasons about these using natural deduction-like methods. We present the following three enrichments of this meta-language. First, programs are represented not by first-order structures but by simply typed λ-terms. Second, schema variables in inference rules can be higher-order variables. Third, the reasoning mechanism is explicitly extended with proof methods which have proved valuable for natural deduction systems. In particular, we add methods for introducing and discharging assumptions and for introducing and discharging parameters. The first method can be used to prove hypothetical propositions while the second can be used to prove generic or universal propositions. We provide several example specifications using this extended meta-language and compare them to their first-order specifications. We argue that our extension yields a more natural and powerful meta-language than the related first-order system. We outline how this enriched meta-language can be compiled into the higher-order logic programming language λProlog.

1 INTRODUCTION

We examine the specification and manipulation of programming languages in the framework of natural semantics, which was initially inspired by the work of G. Plotkin in operational semantics [27] and extended by G. Kahn and others at INRIA [16, 2, 1]. These meta-languages represent programs as first-order tree structures and provide a reasoning style similar to that of natural deduction. One strength of natural semantics is that it can be compiled directly into PROLOG by using first-order terms to represent programs and by using unification and backchaining to implement the natural deduction-style reasoning. We shall show how natural semantics can be extended by introducing higher-order terms (simply typed λ-terms) directly into the meta-language. Along with such an extension, we extend the underlying reasoning mechanism with two kinds of

*This is a revised version of a paper submitted to the Workshop on Meta-Programming in Logic Programming, Bristol, June 1988. Comments are welcomed. Address correspondence to John Hannan at the above address or at "hannan@linc.cis.upenn.edu."

†Supported by a fellowship from the Corporate Research and Architecture Group, Digital Equipment Corporation, Maynard, MA.

‡Supported by NSF grant CCR-87-05596 and DARPA N000-14-85-K-0018.

introduction and discharge rules. We argue that this extension yields a higher-level description of many program manipulations and provides a more natural specification of these tasks. Many low-level routines for manipulating program code, such as substitutions for free variables, changing bound variable names, maintaining a context, *etc.*, are essentially moved to the meta-language and need not be written into the specification.

1.1 Motivation

The process of designing and implementing new programming languages could be greatly enhanced with a meta-programming tool that provides a generic framework in which designers can develop and experiment with new programming languages and their properties. Programs like compilers, interpreters, type checkers, and type inferencers are standard tools that designers implement and test. For our purposes we are concerned with two levels of programming languages: the object-languages and the meta-languages in which compilers, interpreters, *etc.* for the object languages are implemented. The work in natural semantics [16] can be viewed as a proposal for a general and flexible meta-language in which such meta-programs can be written for a wide class of object-languages. This work has also demonstrated how such apparently disparate tasks as compiling and type checking can be presented in a unified framework, one akin to natural deduction-style theorem provers. In this framework, properties of object-level programs are given by a set of axioms and inference rules in which the simplest propositions often have immediate and declarative readings. Such propositions are, for example, of the form $\Gamma \vdash E : \tau$, denoting the property that "in context Γ program phrase E has type τ," or of the form $\Gamma \vdash E \rightarrow F$, denoting the property that "in context Γ program phrase E has value F (or compiles to program F)." Reasoning about programs, then, is performed by a restricted kind of theorem proving in this meta-language.

Since, as we shall argue, higher-order expressions and higher-order reasoning arise naturally in meta-level manipulations of program code, a higher-order extension to natural semantics could be useful. While such higher-order extensions are not essential to capture the meaning of these program manipulations, having certain higher-order operations, such as abstraction, β-conversion, and unification of λ-terms, as primitives permits the specification of some manipulations to be given at a higher and more general level than in natural semantics. Since resulting specifications are often more perspicuous, establishing their formal correctness properties will hopefully also be easier.

1.2 Related Work

The seminal work on a structured approach to operational semantics is by G. Plotkin [27]. This work introduced the general approach of describing semantics with inference rules. Much of the work reported in this paper was motivated by the research in natural semantics done subsequently by G. Kahn and many others at INRIA. They chose to study semantics of programming languages by developing proof systems similar to the ones we develop in this paper. A crucial difference is that they view programs as first-order structures which can be manipulated by a first-order language (e.g., PROLOG). While the use of a first-order language may lead directly to efficient implementations, the logical aspects of program systems are not always elucidated in a strictly first-order setting. This point will become obvious, for example, when we discuss the manipulation of recursive objects.

Other efforts with similar goals have considered denotational instead of operational or natural semantics. Some of the earliest work in this area is due to Mosses and his SIS system [22]. SIS is a compiler generator which takes as input a specification of the denotational semantics of an object

language and produces a compiler for this language. There have been numerous other efforts with similar goals and their contributions are well documented in the literature [9, 15]. While this abundance of work has produced some fruitful results certain limitations appear inherent with this approach. First the mathematical machinery required to specify a denotational semantics can become burdensome for practical language definitions. Furthermore, denotational semantics, in general, does not appear to be a convenient technique for specifying parallelism or nondeterminism. Many of the techniques used in natural semantics do not seem to suffer from these deficiencies.

From the perspective of reasoning in a higher-order setting this work shares much with several other projects. In [14, 10, 19] the authors argue that higher-order unification and logic programming can elegantly be used to manipulate programs in semantically meaningful ways. In [7] a logic programming language containing not only higher-order terms but also the ability to introduce and discharge assumptions and parameters is used to specify and implement various natural deduction-style theorem provers. Many techniques from that paper find immediate applications in this paper. The meta-language outlined in the next section is essentially an application of the general notion of *higher-order abstract syntax* to a particular program manipulation system [25]. This meta-language can also be specified in the much richer proof system specification language of LF [11]. Although we outline briefly how this meta-language can be implemented in the λProlog logic programming language [7, 18, 23], it should also be possible to provide an immediate implementation in the theorem proving system Isabelle [24].

2 A HIGHER-ORDER META-LANGUAGE

To extend the meta-language proposed in [16] we use a higher-order abstract syntax for the representation of programs as data objects and for specifying proof rules [25]. The use of higher-order features actually provides a "high-level" approach to specifying program manipulations. The specification language described here could be translated or "compiled" into either natural semantics or into first-order Prolog, though we will not attempt to do either. This translation could, however, be important since the removal of high-level features (here, higher-order features) usually results in more efficient implementations of specifications. Our purpose in the current work, however, is to concentrate on the inherent logical content of various program manipulations and not to address the efficiency of the programs which implement them.

The abstract syntax for programs and types of the object language is based on the simply typed λ-calculus. We shall represent programs as simply typed terms by introducing an appropriate set of constants from which we can construct terms denoting programs. In general, for each programming language construct we introduce a new constant which is used to build a term representing this construct. We also define new base types (or sorts) corresponding to the different categories of the object language. For example, a simple functional language might require two sorts, one for object-level terms and one for object-level types. We provide an example of such an abstract syntax in the next section. In the rest of this section, we present the proof and reasoning components of our meta-theory.

2.1 An Abstract Proof System

Given a representation of programs as terms we now describe the general structure of a proof system for manipulating these terms. We consider a natural deduction calculus patterned after Gentzen proof systems [8]. The propositions of this system will typically be binary statements of the form $\vdash E : \tau$ or $\vdash E \rightarrow F$. Here, of course, we are thinking of E, F, τ as variables which might range over λ-terms. Although propositions can have more complex structure, we shall restrict them to

be λ-terms with a constant symbol as their head.

The proof system of our meta-language comes equipped with four built-in inference figures. The first has the structure:

$$\frac{A_1}{A_0}$$

in which the λ-terms representing the propositions in A_0 and A_1 are $\beta\eta$-convertible. By virtue of this rule, we generally think of any two λ-terms as equal if they are $\beta\eta$-convertible. The second inference figure is:

$$\frac{A_1 \quad A_2}{A_1 \ \& \ A_2}$$

This rule is called *conjunction introduction.* When implementing this inference rule, we interpret it in the following backward fashion: to establish the proposition in $A_1 \ \& \ A_2$, establish the two separate propositions found in A_1 and A_2.

The remaining two rules deal with introduction and discharge. To specify the introduction and discharge of assumptions needed to prove hypothetical propositions we use the following inference figure.

$$\frac{\begin{array}{c}(A_1)\\ \vdots \\ A_2\end{array}}{A_1 \Rightarrow A_2}$$

That is, to prove $A_1 \Rightarrow A_2$, first assume that there is a proof of A_1 and attempt to build a proof for A_2 from it. If such a proof is found, then the implication is justified and the proof of this implication is the result of discharging the assumption about A_1. This rule is called *implication introduction.* Proving a universally quantified proposition has a similar structure, suggesting the following inference figure.

$$\frac{A[x \mapsto c]}{(\forall x)A}$$

Here, to prove a universal instance, a new parameter (c) must be introduced and the resulting generic instance of the quantified formula must be proved. Of course, after that instance is proved, the parameter must be discharged, in the sense that c cannot occur free in A or in any undischarged hypotheses. This rule is called *universal introduction.*

A specification of a meta-level program will be a collection of atomic propositions which will denote axioms and a collection of inference figures, none of which introduce the symbols $\&, \Rightarrow, \forall$. Of course, the premises to user supplied inference figures can contain instances of these symbols. When providing examples of inference figures later in this paper, we shall drop references to the connective & in premises. Inference figures of the form

$$\frac{A_1 \ \& \ A_2}{A_0} \quad \text{will simply be written as} \quad \frac{A_1 \quad A_2}{A_0} \ .$$

A proof in this language will be understood in the standard sense of proofs in natural deduction [8, 28].

2.2 An Implementation of the Meta-Language

Following the observation described in [16] that natural semantics has an intimate connection to logic programming, we show how the preceding four inference figures are related to logic programming. First-order Horn clauses, however, are not strong enough to directly implement these

inference rules. First, the notion of equality between terms would be that of simple tree equality, not that of $\beta\eta$-conversion. Horn clauses also do not provide a mechanism for directly implementing the introduction and discharge of parameters and assumptions. It is not difficult to modify our proof system so that the explicit references to introducing and discharging assumptions could be eliminated in favor of treating basic propositions as essentially sequents. That is, a proposition $\vdash Prop$ would be replaced by a proposition $\Gamma \vdash Prop$, in which Γ is used to store assumptions. This is, for example, used in natural semantics to handle contexts. For the examples in this paper we actually used this approach to implement them in λProlog. We were required to do this since version 2.6 of λProlog does not fully support implication in goal clauses. (See appendix B.) A more serious challenge to Horn clauses is that they cannot naturally implement the universally quantified proposition.

There is, however, a generalization of Horn clauses which adds both implications and universal quantifiers to the body of clauses and permits quantification over higher-order variables. This extension, called *higher-order hereditary Harrop formulas* [20] has (partially) been implemented in the λProlog system. λProlog does, in fact, provide a natural implementation language for these inference rules. For example, the user can specify inference rules by directly writing program clauses containing conjunction, implication, and universal quantifiers, since these are understood on a primitive level of λProlog. For example, clauses of the form

$$A_0 \ :- \ A_1 \ \& \ (\forall x)(A_2 \Rightarrow A_3).$$

can be used to represent complex inference figures. Free (higher-order) variables here are assumed to be universally quantified over the scope of the full clause. Instead of using this kind of syntax to present example inference rules later in this paper, we shall continue to use the more graphically oriented inference figures. All the examples presented in this paper have been implemented and tested in a version of λProlog. The λProlog code for these examples may be found in the appendices.

3 MINI-ML AND TYPE INFERENCE

To introduce the features of our meta-language we present a description of type inferencing for a variant of mini-ML [2], which is a subset of Standard ML containing just the functional part of that language. (It does not contain exceptions, pattern matching, datatype declarations or modules.) This presentation demonstrates how an abstract syntax for a functional language can be constructed using simply typed lambda terms and also how the unique features of our meta-language can be exploited in the manipulation of programs. We take care in making the distinction between terms and types at the object (mini-ML) level and terms and types at the meta-level. We refer to the latter as meta-terms and meta-types. We have two base meta-types, *tm* and *tp*, representing object-level terms and types, respectively.

To define our abstract syntax for mini-ML we begin by giving a signature for some meta-terms that we use to construct terms and types at the object level. (See figure 1.) Notice that the constants **lamb**, **let** and **fix** are higher-order, that is, they each require a functional argument of type $tm \rightarrow tm$. In the examples that follow M will be used as a higher-order variable of this meta-type. '$*$' and '$\rightarrow$' are the product space and function space constructors, respectively, for *tp*. We have overloaded the symbol '$\rightarrow$', using it at both the object and meta levels; its use, however, should be clear from context. The object types we consider are only monotypes (in the sense of [21] as we do allow type variables). Expressions with polytypes (i.e., monotypes that may be prefixed by universal quantification over type variables) do arise, however, in mini-ML. At the conclusion of this section we present a separate discussion of polytypes.

meta-term	meta-type
true, false	tm
$0, 1, 2, \ldots$	tm
','	$tm \rightarrow tm \rightarrow tm$
fst	$tm \rightarrow tm$
snd	$tm \rightarrow tm$
if	$tm \rightarrow tm \rightarrow tm \rightarrow tm$
'@'	$tm \rightarrow (tm \rightarrow tm)$
lamb	$(tm \rightarrow tm) \rightarrow tm$
let	$(tm \rightarrow tm) \rightarrow tm \rightarrow tm$
fix	$(tm \rightarrow tm) \rightarrow tm$

meta-term	meta-type
int	*tp*
bool	*tp*
'$\rightarrow$'	$tp \rightarrow tp \rightarrow tp$
'$*$'	$tp \rightarrow tp \rightarrow tp$

Figure 1: Signature for mini-ML abstract syntax

syntax	description
c	constants: integers, true, false, etc.
x	variables
(e_1, e_2)	pairing
(fst e), (snd e)	first and second projections
(if e_1 e_2 e_3)	if e_1 then e_2 else e_3
(e_1 @ e_2)	application
(lamb M)	$M = \lambda x.e$ (lambda abstraction)
(let M e_2)	$M = \lambda x.e_1$ (let $x = e_2$ in e_1)
(fix M)	$M = \lambda f.e$ (least fixed point operator)

Figure 2: Abstract Syntax for mini-ML

3.1 An Abstract Syntax for mini-ML

Figure 2 contains the higher-order abstract syntax for expressions in mini-ML. The first few lines treat constants, variables, pairing, projections and the conditional in a traditional manner. Application is made explicit with the infix operator '@'. For lambda abstraction we introduce the constructor **lamb** which takes a meta-term M of the form $\lambda x.e$, in which x and e are of meta-type tm, and produces a mini-ML term. Similar to **lamb**, the **let** construct uses a meta-term M of the form $\lambda x.e$ to represent the binding of an identifier. To accommodate recursion we introduce the **fix** construct which again uses an explicit abstraction to capture the binding. Thus, we employ the general principal that bindings at the object level have an associated abstraction at the meta-level. Two differences between our approach and INRIA's (aside from the use of higher-order syntax) are immediately obvious. First, our binding of identifiers is restricted to individual identifiers while the INRIA approach allows patterns (e.g., pairs) to be bound simultaneously. Second, we do not provide a **letrec** construct, but rather obtain an equivalent construct by combining **let** and **fix** [2].

This abstract syntax is essentially the embedding of the untyped lambda calculus into a simply typed calculus as described originally in [29], in terms of lattices, and later in [17]. Using the notation of [17] our meta-term **lamb** corresponds to the function Ψ, for coercing functions into terms. The meta-term '@' corresponds to the function Φ for coercing terms into functions. Thus our representation of mini-ML code is essentially the same as first encoding them as untyped lambda terms and then embedding them into the typed calculus using Φ and Ψ.

Throughout this paper we will avoid discussing primitive operations of mini-ML, such as $+$, $-$,

etc. They are, of course, important to have in the full language but including them here is neither difficult nor illuminating. (See appendix B for examples of how we treat primitive operations.)

The restriction that bindings cannot be over patterns is used only as a matter of convenience here: it does not reduce the expressiveness of our version of mini-ML. Most expressions that bind patterns can be transformed into equivalent ones that do not use patterns. For example consider the expression found in [2] that contains a pair of mutually recursive function definitions:

$$\begin{array}{rcl} \textsf{letrec } (even, odd) & = & ((lambda\ x\ (\textsf{if } x = 0 \textsf{ then } true \textsf{ else } odd(x-1))), \\ & & \ (lambda\ x\ (\textsf{if } x = 0 \textsf{ then } false \textsf{ else } even(x-1)))) \\ \textsf{in } even(3) & & \end{array}$$

(We use the notation $(lambda\ x\ E)$ to denote the "first-order" term representing a lambda abstraction. This distinguishes it from our use of λ in simply typed terms.) This recursive definition can be rewritten in our abstract syntax as follows.

$$\begin{array}{l} (\textsf{let } \lambda f((\textsf{fst } f)\ 3) \quad (\textsf{fix } \lambda f(\ \lambda x(\textsf{if } x = 0 \ \textsf{ true } \ (\textsf{snd } f)(x-1)), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda x(\textsf{if } x = 0 \ \textsf{ false } \ (\textsf{fst } f)(x-1))))) \end{array}$$

In this example, we have systematically dropped the apply "@" operator in order to make this example more readable. (There should be 14 occurrences of "@" in this example.) In words, the first expression contains a pair of (mutually) recursive functions and the second expression contains a recursively defined pair of functions. A general program transformer taking expressions of the former kind into equivalent ones of the latter kind is given in appendix A. The simplicity of the transformer given there helps justify our claims that our meta-language allows us to naturally specify manipulations of programs.

3.2 Environments versus Abstractions

Before presenting the type inference proof procedure we make another distinction between our method and typical approaches to natural semantics. This distinction concerns the treatment of identifiers. The typical approach to analyzing programs uses an environment (or context) to denote a finite mapping from identifiers to some domain (e.g., types or terms). When analyzing an abstraction, the bound variable is stripped from the abstraction and the identifier which names that bound variable is added to the context. The meaning of such an identifier within the body of the abstraction is then determined by "looking up" the value associated with the identifier in the current environment. We refer to this technique as the environment approach.

Given our commitment to representing program abstractions using abstractions with λ-terms and to equating such terms when they are $\beta\eta$-convertible, it is impossible to access the bound variable name of a λ-term at the meta-level, since such an operation would return different answers on equal terms. A combination of the $\forall$ and $\Rightarrow$ propositions, however, can provide a very simple solution to this problem. When an abstraction is encountered, typically within lamb, let and fix constructions, a $\forall$ judgement is used to introduce a new parameter. That parameter is then substituted into the abstraction using β-conversion. The value or type to be associated with this new parameter is then introduced as an assumed proposition. In this way, the newly introduced identifier is used to stand for the name of the bound variable.

This relation between the environment approach and our technique is similar to an observation by Plotkin about evaluations in the SECD machine [26]. There two different evaluation functions were defined: the awkward *Eval* function defined in terms of closures and the simpler *eval* defined

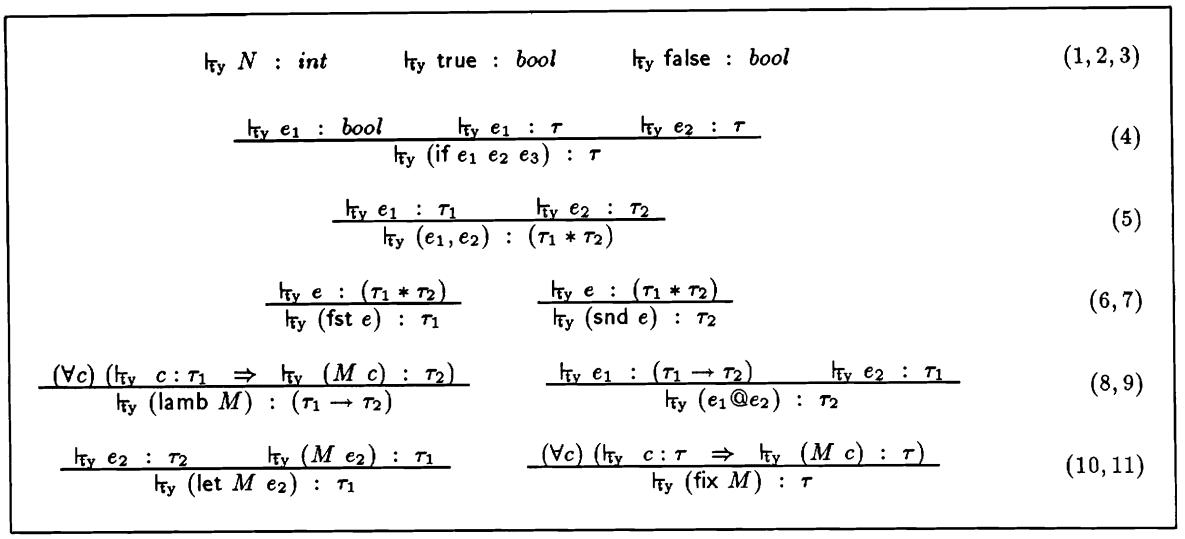


Figure 3: Type Inference for Mini-ML

using substitution (β-conversion, here). While these two functions were shown to be equivalent, introducing the simpler definition for evaluation allowed properties of the SECD machine to be described much more naturally than with the first, more cumbersome, definition. Similarly, we believe that the use of abstractions and substitution in our meta-language will often produce this kind of advantage over programs using the environment approach.

3.3 A Type Inference System

The proof system for type inference in our formulation of mini-ML is given in figure 3. A proof of the proposition $\vdash_{\tau y} E : \tau$, in which E is a closed expression given in the above abstract syntax, states that E has type τ. To be precise we should prove certain properties about this typing system, e.g., soundness, completeness and principal typing [4, 12]. However, due to the preliminary nature of this work we prefer to provide an informal discussion of this system. The first three clauses (actually axioms) are for typing the constants of the language; here N denotes any integer. The next clause gives the usual typing for the conditional statement. Clause 5 gives the typing for pairs. Clauses 6 and 7 give the typings for the corresponding projections.

Clause 8 is the typing rule for lambda abstraction and it is a bit different from the usual typing rule using environments. In the environment approach, typing the (first-order) term (*lambda* x E) would first require adding the type assignment $x : \tau_1$ to the environment, then computing the type of E in this new environment to be τ_2, and then finally inferring the type of the original term to be $\tau_1 \rightarrow \tau_2$. Our rule uses β-reduction and operationally works as follows. Given the term (lamb M) we first pick a new constant c and assume it has type τ_1 (i.e., we introduce the assumption $\vdash_{\tau y} c : \tau_1$). Under this assumption we then type (the $\beta\eta$-normal form of) the term (M c). If M is of the form $\lambda x.e$ then the β-reduction is, in this case, equivalent to the substitution $e[x \mapsto c]$. If we infer the type τ_2 for this term then we infer the type of the original term to be $\tau_1 \rightarrow \tau_2$. Informally, this infers the correct type because every occurrence of x bound by this abstraction has been replaced by a term c whose type will be inferred to be τ_1. Although this is in many ways similar to the environment approach, it avoids the need to access the names of bound variables.

Clause 9 is the usual typing rule for application. Clause 11 for fixed points uses the same technique as **lamb**, though in this case we know that M must be of type $\tau \rightarrow \tau$ for some τ. Clause 10 requires some explanation. The more standard implementation of type inference for **let** first infers the principle type for e_2, then generalizes that type with a universal quantifier over type variables, yielding a polytype. Later in the typing of the abstraction M, various universal instances of this polytype could be made for instances of the abstracted variable of M. Our meta-language, however, contains no method for generalizing a free variable into a bound variable, and so this kind of implementation is not possible here. Instead, we avoid inferring a polytype for e_2 explicitly. Clause 10 requires that e_2 have some type, but that type is then ignored. β-reduction is used to substitute e_2 into the abstraction M, and then the type of the result is inferred. If e_2 is placed into several different places in M, each of those instances will again have a type inferred for them; this time the types might be different. Therefore, e_2 could be polymorphic in that its occurrences in M might be at several different types.

We do not need a rule for typing identifiers because any identifier occurring in a term is replaced via β-reduction with either *(i)* a term explicitly typed via an assumption (**lamb**, **fix**) or *(ii)* a term whose type has already been inferred (**let**). (Recall that we are typing only closed expressions.) Note that the three clauses that make significant use of higher-order features correspond precisely to the three clauses in the environment approach that extend the environment. This is not surprising as these are the only three clauses that introduce identifiers and bindings. An implementation of this system is given in appendix B.

3.4 The Subsumes Relation for Polytypes

As a second example of using our meta-language to manipulate ML types, we present a proof system for the subsumes relation on polytypes [21]. For this purpose, we now introduce a higher-order constant for constructing ML types, namely the type quantifier **forall** which is of meta-type $(tp \rightarrow tp) \rightarrow tp$. Any term of type tp which does not contain an instance of this constant is a monotype. A term of type tp in which all of occurrences of **forall** are in its prefix (that is, no occurrence of **forall** is in the scope of $*$ or $\rightarrow$) is called a *polytype* (a monotype is a polytype). It is possible to construct terms (of meta-type tp) that are neither monotypes nor polytypes, but these will not interest us here. In the following discussion, the greek letter τ will represent a monotype and σ a polytype. Before defining the subsumes relation we define an auxiliary definition.

Definition 1 (Instance of a Polytype) *τ is an instance of polytype* (**forall** $\lambda t_1(\ldots($**forall** λt_n $(\tau'))\ldots))$ *if there exists a substitution S of the variables $t_1, \ldots, t_n$ into monotypes such that $\mathcal{S}(\tau') = \tau$.*

The subsumes relation on polytypes is then given by the following.

Definition 2 (Subsumes) *Let σ_1 and σ_2 be two polytypes. σ_1 subsumes σ_2, written $\sigma_1 \sqsubseteq \sigma_2$, if every instance of σ_2 is also an instance of σ_1.*

For example, the polytype (**forall** $\lambda t.t$) subsumes all other polytypes. An informal operational description of this definition is the following. Given σ_1 and σ_2, erase the quantifiers of each yielding two monotypes, τ_1 and τ_2. Then $\sigma_1 \sqsubseteq \sigma_2$ iff there exists a substitution $\mathcal{S}$ such that $\mathcal{S}(\tau_1) = \tau_2$. Since the erasure of bound variables is another operation not available in our meta-language, we need to approach the implementation of subsumes differently.

In our meta-language we can construct a simple proof system for the subsumes relation; it is given in figure 4. The first clause states the obvious: any polytype subsumes itself. The second

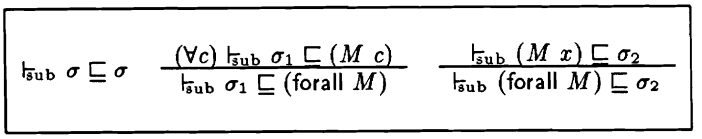


Figure 4: Subsumes Relation for Polytypes

clause produces a 'canonical' instance of σ_2. This step is essentially like the process of erasing a type quantifier. The meta-level universal quantifier used in this clause ensures that, after removing the quantifiers on σ_2, revealing a monotype, any future substitution does not affect this monotype (its free variables are, in a sense, protected). The third clause is used to build an instance of the first type by stripping off a quantifier (replacing a bound (type) variable with a free one).

Notice that these three proof rules have a simple declarative reading. Assume that types are interpreted as sets of objects of that type, that forall is interpreted as intersection, and $\sqsubseteq$ as subset. The second clause states that a type is a subset of the intersection of a family of types if it is a subset of all members of the family. The third clauses similarly states that if some member of a family is a subset by a given type, then the intersection of that family is a subset of that type.

4 DYNAMIC SEMANTICS

In mini-ML the evaluation of an expression E always yields some 'canonical' value α. Following [16] we refer to a formal specification of an evaluator for a language as the language's *dynamic semantics.* We characterize the dynamic semantics of an object language via judgements of the form $\vdash E \longrightarrow \alpha$ in which E is an expression of the object language and α is the result of evaluating E. By providing rules corresponding to the operational behavior of the language (with the general guideline of having one rule for each programming language construct) we can specify the declarative aspects of interpreters (or evaluators) for the language, isolated from control issues. As mentioned previously this provides a convenient tool for analyzing and experimenting with new programming languages.

We now present a dynamic semantics for mini-ML, using the same higher-order abstract syntax as given in the previous section. Propositions in our system are of the form $\vdash_{ml} E \longrightarrow \alpha$ in which E and α are expressions in mini-ML and α is the result of evaluating E. The dynamic semantics for our version of mini-ML is given in figure 5. Many of these rule are similar to the ones given in figure 2 of [16], except that our rules do not make explicit reference to an environment. We highlight here, then, only the important differences between the two, which principally revolve around the treatment of variable bindings. To aid in the discussion we present those rules from [16] which differ significantly from our own. These are given in figure 6 with numbers referring to the corresponding rules of figure 5. An implementation of our system is given in appendix B.

First consider rule (9) for handling abstractions. In the environment approach, an explicit closure is created for preserving the current environment. This ensures static scoping. Closures are not used in our specification since no environment is maintained: neither the universal nor the implicational propositions are used in this example. Static scoping is ensured in our model because β-reduction, as a means of propagating binding information, guarantees that the identifiers occurring free within a lambda abstraction are replaced (with their associated value) prior to manipulating the abstraction. The two rules for application (10) are somewhat similar, though in our model e_1 evaluates to a lambda abstraction rather than a closure. Also use of β-conversion

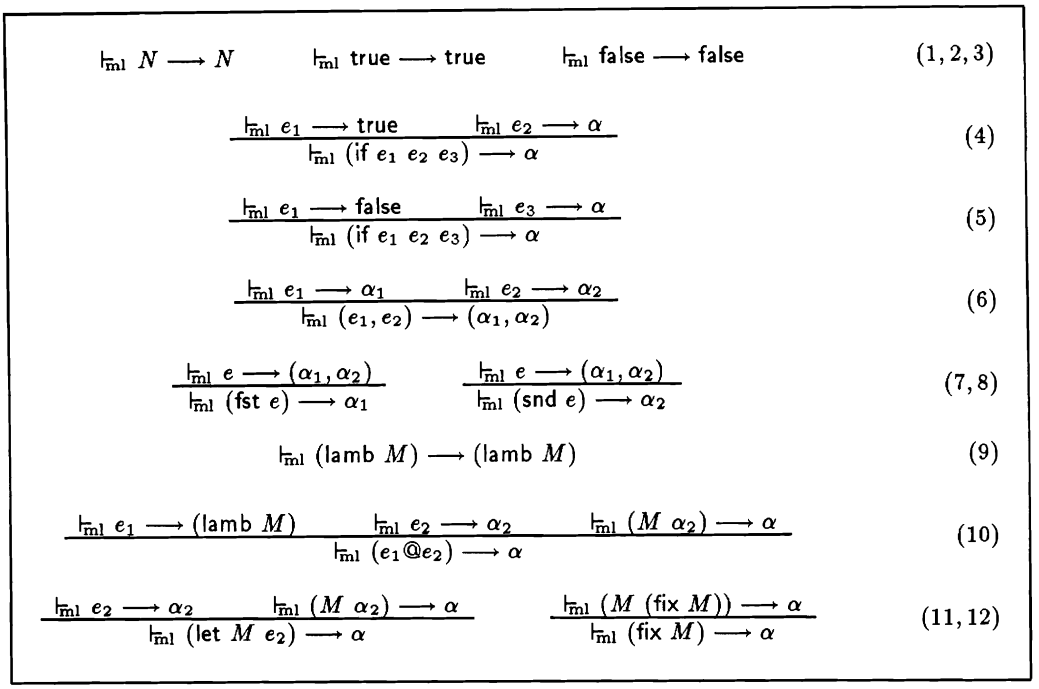
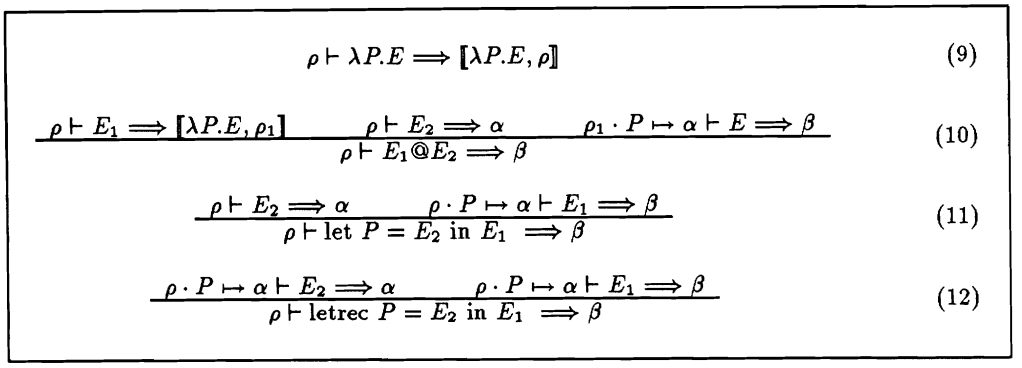


Figure 5: Dynamic Semantics for Mini-ML

instead of environment updating correctly models the notion of function application (with a call-by-value semantics). Similar comments apply to our rule for **let** (11).

Finally we have the rules for introducing recursion. We opt for a fixed point operator with its intuitive operational semantics (i.e., unfolding). This again makes explicit use of β-conversion since the higher-order variable M is applied to the term (**fix** M). The result of β-converting this expression substitutes the recursive call, namely (**fix** M), within the body of the recursive program, namely M. This rule in the environment approach is less perspicuous and relies on constructing an infinite structure. This technique appears to be motivated more by the underlying implementation language (MU-PROLOG, which supports such constructs) than by a logical description of recursion. This infinite structure results from the construction of a cyclic term in MU-PROLOG when encountering an occur-check situation implicit in the implementation of (12) in figure 6.

We have also specified a proof system providing a dynamic semantics for the Categorical Abstract Machine (CAM) [3]. As the CAM is a low-level stack-based machine, higher-order syntax provides little advantage in specifying its semantics. Values in the CAM must be explicitly maintained on a stack, thus forming a kind of environment; hence we could not dispense with environments. We were, however, able to avoid the use of infinite structures for handling the **rec** command. In the first-order system of [16], the **rec** command, which allows recursion, is handled by constructing a cyclic (hence, infinite) environment. We construct a higher-order object for the environment and then represent this recursive environment by a fixed point. This specification, we believe, provides a clear picture of the underlying stack manipulation of the CAM. An implementation of this system is given in appendix B.



$$\rho \vdash \lambda P.E \Longrightarrow [\lambda P.E, \rho] \tag{9}$$

$$\frac{\rho \vdash E_1 \Longrightarrow [\lambda P.E, \rho_1] \qquad \rho \vdash E_2 \Longrightarrow \alpha \qquad \rho_1 \cdot P \mapsto \alpha \vdash E \Longrightarrow \beta}{\rho \vdash E_1 @ E_2 \Longrightarrow \beta} \tag{10}$$

$$\frac{\rho \vdash E_2 \Longrightarrow \alpha \qquad \rho \cdot P \mapsto \alpha \vdash E_1 \Longrightarrow \beta}{\rho \vdash \text{let } P = E_2 \text{ in } E_1 \Longrightarrow \beta} \tag{11}$$

$$\frac{\rho \cdot P \mapsto \alpha \vdash E_2 \Longrightarrow \alpha \qquad \rho \cdot P \mapsto \alpha \vdash E_1 \Longrightarrow \beta}{\rho \vdash \text{letrec } P = E_2 \text{ in } E_1 \Longrightarrow \beta} \tag{12}$$

Figure 6: Dynamic Semantics for Mini-ML (using environments)

5 TRANSLATION FROM MINI-ML TO CAM

As a final example we take the translation from mini-ML to CAM given in [6] and specify it in our higher-order meta-language. The inference figures for this translation are given in figure 7. We were able to replace the use of environments with de Bruijn indices (the D's occurring in the proof rules). Such a simple addressing scheme is due partly to our restriction that bindings refer only to individual identifiers. When dealing with identifiers, our presentation is somewhat simpler than that of [6]. We give only an overview here of the functioning of this proof system. We have not presented the constructors for the abstract syntax for the CAM since they all have straightforward first-order types. An implemenation of this translation system is given in appendix C.

Given a mini-ML term e we define the *depth* of a subterm e_1 of e to be the number of variable bindings in e of which e_1 is in the scope. The proposition $D \vdash_{\text{tr}} e \longrightarrow C$ then has the declarative reading: "the mini-ML term e, occurring at a depth D in some term, translates to the CAM code C." The depth of a subterm is needed in order to generate the correct CAM code for accessing the value of mini-ML identifiers. Identifiers are translated into access paths into an environment on top of the CAM's stack. The precise nature of this environment is not important; we only note that it is, in general, a tree structure with values at its leaves. An access path is a sequence of fsts and snds for descending through this environment to retrieve a desired value. Due to the uniform manner in which identifiers are introduced into (our simplified) mini-ML the access path for an identifier has the form "$\mathsf{fst}^{d-1};\mathsf{snd}$" in which d is the usual de Bruijn index for the identifier [5]. We can compute this index during translation by noting that $d = D - D_1 + 1$ where D is the depth of the occurrence of the identifier and D_1 is the depth of the binding occurrence for the identifier. For example, in the term $\lambda x \lambda y.x$ the occurrence of the identifier x is at depth 2 and the binding occurrence of x is at depth 1 (the top level). The de Bruijn index for the occurrence of x is then computed to be 2 $(= 2 - 1 + 1)$. (Compare this with the same lambda term given in a syntax using de Bruijn indices: $\lambda\lambda.2$.)

To implement this translation in our meta-language we use a technique similar to our handling of lambda abstraction in the mini-ML type inference system. Consider rule 9 in figure 7. To translate the term (lamb M) we introduce a new parameter c and apply the meta-term M to it. This substitutes c for the abstracted variable of the term. Since the term (lamb M) represents the introduction of a new binding we must increment by one the depth D for translating subterms in M. The assumption $D + 1 \vdash_{\text{dp}} c$ asserts that c is an identifier which was abstracted at depth

$D+1$. This will be precisely the information required to produce the access path for this identifier (given by rule 4). When the subterm c is reached during the translation process the depth (D_1) of its binding occurrence is obtained from the assumptions of the form $D_1 \vdash_{dp} c$. Noting the above relation between de Bruijn indices and our depths we form rule 4 to generate the correct access path. This is essentially the rule for the "categorical combinator" $n!$ given in [3], though they work directly with de Bruijn indices and so their translation of identifiers into such indices is simpler.

The translations for **let** and **fix** (rules 11 and 12, respectively) use the same approach for manipulating the identifiers. The translations for the remaining constructs are almost identical to their counterparts in [6] and we do not discuss them here.

$$D \vdash_{tr} N \longrightarrow (\text{quote } N) \qquad (1)$$

$$D \vdash_{tr} \text{true} \longrightarrow (\text{quote true}) \qquad D \vdash_{tr} \text{false} \longrightarrow (\text{quote false}) \qquad (2,3)$$

$$\frac{D_1 \vdash_{dp} x}{D \vdash_{tr} x \longrightarrow \text{fst}^{D-D_1+1}; \text{snd}} \qquad (4)$$

$$\frac{D \vdash_{tr} e_1 \longrightarrow C_1 \qquad D \vdash_{tr} e_2 \longrightarrow C_2 \qquad D \vdash_{tr} e_3 \longrightarrow C_3}{D \vdash_{tr} (\text{if } e_1\ e_2\ e_3) \longrightarrow (\text{push}; C_1; \text{branch}(C_2, C_3))} \qquad (5)$$

$$\frac{D \vdash_{tr} e_1 \longrightarrow C_1 \qquad D \vdash_{tr} e_2 \longrightarrow C_2}{D \vdash_{tr} (e_1, e_2) \longrightarrow (\text{push}; C_1; \text{swap}; C_2; \text{cons})} \qquad (6)$$

$$\frac{D \vdash_{tr} e \longrightarrow C}{D \vdash_{tr} (\text{fst } e) \longrightarrow (\text{push}; C; \text{fst})} \qquad \frac{D \vdash_{tr} e \longrightarrow C}{D \vdash_{tr} (\text{snd } e) \longrightarrow (\text{push}; C; \text{snd})} \qquad (7,8)$$

$$\frac{(\forall c)\ (D+1 \vdash_{dp} c \ \Rightarrow\ D+1 \vdash_{tr} (M\ c) \longrightarrow C)}{D \vdash_{tr} (\text{lamb } M) \longrightarrow \text{cur}(C)} \qquad (9)$$

$$\frac{D \vdash_{tr} e_1 \longrightarrow C_1 \qquad D \vdash_{tr} e_2 \longrightarrow C_2}{D \vdash_{tr} (e_1 @ e_2) \longrightarrow (\text{push}; C_1; \text{swap}; C_2; \text{cons}; \text{app})} \qquad (10)$$

$$\frac{D \vdash_{tr} e_2 \longrightarrow C_2 \qquad (\forall c)\ (D+1 \vdash_{dp} c \ \Rightarrow\ D+1 \vdash_{tr} (M\ c) \longrightarrow C_1)}{D \vdash_{tr} (\text{let } M\ e_2) \longrightarrow (\text{push}; C_2; \text{cons}; C_1)} \qquad (11)$$

$$\frac{(\forall c)\ (D+1 \vdash_{dp} c \ \Rightarrow\ D+1 \vdash_{tr} (M\ c) \longrightarrow C)}{D \vdash_{tr} (\text{fix } M) \longrightarrow (\text{push}; \text{rec}(C))} \qquad (12)$$

Figure 7: Translation from mini-ML to CAM

6 CONCLUSION

We have presented an enrichment of the natural semantics meta-language. This language differs from previous work in natural semantics in three significant ways. First, we represent programs as simply typed λ-terms instead of first order structures. We demonstrated how this representation affords higher-level reasoning about programs, since many low-level manipulations, e.g., substitutions for free variables and changing bound variable names, are pushed into the meta-language and need not be explicitly stated in a specification. Second, we extended the reasoning mechanism with proof methods which have proved valuable in other natural deduction systems. We incorporated explicit methods for introducing and discharging assumptions and parameters which are used to

prove hypothetical and universal propositions. Typically we applied these two in tandem for introducing identifiers for object-level bound variables. Finally, the schema variables appearing in inference rules can be higher-order variables.

We presented several examples to support our claim that this enriched meta-language permits a high-level and elegant specification of program manipulations. From the specification point-of-view, we argued that the proof rules provided in this enriched meta-language were more perspicuous and we did not need to introduce any non-logical meta-level operations to implement all the examples considered above. An important concern for the researchers in natural semantics was the compiling of inference rules into efficient programs. Although we see no reason to believe that the specifications given here could not also be implemented efficiently, it seems probable that such compiling will be more involved than it is for compiling a first-order natural semantics.

By providing a purely logical framework in which one can naturally reason about properties of programs we expect that correctness properties for these systems will be easier to show than for systems using only first-order features. While correctness proofs have been given for translations in natural semantics [6], we believe that proofs for systems specified in our meta-language will be simpler due to the stronger meta-theory of our language. The construction of such correctness proofs for the meta-level programs in this paper has not yet been done. Finding such proofs is one of our next concerns.

Acknowledgements: We would like to thank Carl Gunter for first directing us towards the work on operational and natural semantics. We also like to thank Amy Felty, Elsa Gunter and Val Breazu-Tannen for several valuable discussions related to this paper.

REFERENCES

[1] D. Clément. The natural dynamic semantics of mini-standard ML. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pages 67–81, Springer-Verlag LNCS, Vol. 250, 1987.

[2] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: mini-ML. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 13–27, 1986.

[3] G. Cousineau, P-L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.

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

[5] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[6] J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First ACM Conference on Logic in Computer Science*, pages 193–205, 1986.

[7] A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In *Proceedings of the Ninth International Conference on Automated Deduction*, 1988.

[8] G. Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland Publishing Co., 1969.

[9] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

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

[11] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, 1987.

[12] R. Hindley. The completeness theorem for typing λ-terms. *Theoretical Computer Science*, 22(1–2):1–18, 1983.

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

[14] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order logic. *Acta Informatica*, 11:31–55, 1978.

[15] N. Jones, editor. *Semantics-Directed Compiler Generation.* Volume 94 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[16] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39, Springer-Verlag LNCS, Vol. 247, 1987.

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

[18] D. Miller and G. Nadathur. Higher-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, Springer-Verlag, 1986.

[19] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Fourth Symposium on Logic Programming*, IEEE Press, 1987.

[20] D. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Symposium on Logic in Computer Science*, pages 98–105, ACM Press, 1987.

[21] J. Mitchell and B. Harper. The essence of ML. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 28–46, 1988.

[22] P. Mosses. *SIS: a Compiler Generator System Using Denotational Semantics.* DAIMI MD-30, Aarhus University, Aarhus, Denmark, August 1979.

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

[24] L. Paulson. *The Foundation of a Generic Theorem Prover.* Technical Report 130, University of Cambridge, Cambridge, England, March 1988.

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

[26] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1976.

[27] G. Plotkin. *A Structural Approach to Operational Semantics.* DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[28] Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Uppsala, 1965.

[29] C. P. Wadsworth. The relation between computational and denotational properties for Scott's D_∞ models of the lambda-calculus. *SIAM Journal of Computing*, 5(3):488–521, 1976.

Appendix A: Transformation of Recursive Equations

In section 3 we presented examples of two different abstract syntaxes for representing recursive definitions. We present here a transformation specified in our meta-language for converting a recursive definition of the first kind (which we call recursive equations) to an equivalent definition of the second kind (which we call fixed point equations). As presented earlier the recursive equations actually appear embedded in a **letrec** expression which combines the recursive definition and an application. To isolate just the definition part we introduce two new constructors, '$\Leftarrow$' and **receq**, with meta-types $(tm * tm) \rightarrow (tm * tm) \rightarrow tm$ and $(tm \rightarrow tm \rightarrow tm) \rightarrow tm$, respectively. The intended meaning of '$\Leftarrow$' is to associate a pair of identifiers with a pair of (mutually recursive) function definitions. To facilitate the manipulation of this term we abstract out the names of these identifiers, yielding a meta-term. To coerce this meta-term back into a λ-term of type tm we use the **receq** constructor which takes this meta-term to a term. For example, a definition of the even/odd functions using recursive equations in this syntax would be:

$$\begin{array}{rcl}(\textsf{receq}\ \lambda F\lambda G.((F,G) & \Leftarrow & ((\textsf{lamb}\ \lambda X.(\textsf{if}\ \ (X=0)\ \ \textsf{true}\ \ (G\ (X-1))))\\ & & \ \ (\textsf{lamb}\ \lambda X.(\textsf{if}\ \ (X=0)\ \ \textsf{false}\ \ (F\ (X-1)))))))\end{array}$$

Again, we have dropped occurrences of the apply operator "@" to make this example more readable. Now we wish to transform this term into the term

$$\begin{array}{l}(\textsf{fix}\ \lambda F.(\ \lambda X.(\textsf{if}\ X=0\ \ \textsf{true}\ \ (\textsf{snd}\ F)(X-1)),\\ \qquad\qquad \lambda X.(\textsf{if}\ X=0\ \ \textsf{false}\ \ (\textsf{fst}\ F)(X-1)))))\end{array}$$

This type of transformation is similar to the "curry" transformation of [19] which transforms a function of one argument (representing a pair) to a function of two arguments (representing the two elements of the pair). Let $\vdash_{\textsf{rec}} R \longrightarrow F$ be the proposition stating that the recursive equation represented by R is equivalent to the fixed point equation represented by F. A proof system for this proposition (restricted to definitions of exactly two recursive equations) is given by the single axiom involving higher-order terms:

$$\begin{array}{l}\vdash_{\textsf{rec}} (\textsf{receq}\ \lambda F\lambda G.((F,G) \Leftarrow ((M_1\ F\ G),\ \ (M_2\ F\ G)))) \longrightarrow\\ \qquad (\textsf{fix}\ \lambda F.((M_1\ (\textsf{fst}\ F)\ (\textsf{snd}\ F)),\ \ (M_2\ (\textsf{fst}\ F)\ (\textsf{snd}\ F))))\end{array}$$

This proof rule makes a significant use of higher-order unification [13, 14] to generate the meta-terms M_1 and M_2, both of type $tm \rightarrow tm \rightarrow tm$. Transformations of this type and their reliance on higher-order unification are discussed in [10].

Appendix B: λProlog Source Code for Mini-ML Examples

We give below the λProlog code for the mini-ML type inference and dynamic semantics systems. This is given in four modules (a module is a named collection of declarations and definitions, providing scoping and structuring mechanisms in λProlog): the first gives the signature for the mini-ML abstract syntax; the second gives an implementation of the mini-ML type inference system; the third gives an implementation of the mini-ML dynamic semantics; and the last gives some examples. In λProlog one can accumulate definitions from other modules by explicitly *importing* those modules into another. The **import** declaration provides this capability and is used in the modules below. (Note: the module **lists** contains the usual list manipulation routines, including **member**.

Some additional comments regarding the signature may elucidate the example. We introduce the object types **mint** and **mbool** (for *int* and *bool*) and the object-type constructors **mand** and **mlist** (for × and *list*). We do this to maintain the distinction between the object types and the meta-types (of λProlog) which include types for *int*, *bool*, *etc.* Also we introduce the constructor **#** for coercing meta-level integers (as provided by λProlog) to object-level integers. This allows us to move easily between the object-level representation of integers and the meta-level representation. Similar comments apply to the primitive operators for addition, multiplication, *etc.*

As mentioned previously, the current version of λProlog does not fully implement hereditary Harrop formulas. Specifically, formulas of the form $A_1 \Rightarrow A_2$ are restricted to instances in which A_1 is the name of a module. Hence we cannot directly implement the proof systems given earlier in the paper. We must resort to supplying a context with our proof rules for type inference. Here, a context is a list of pairs (**tm**, **tp**) associating identifiers to types. Only rules 8 and 11 extend the context and the last clause for **infer** searches the environment to find the type of an identifier (really a universal constant).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%    mini-ML : Signature                                            %%%
%%%    Declaration of Type and Expression Constructors                %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module mldecl.

%%% Infix Declarations for Constructors
infix  40  tand  yfx.                  %%% product type constructor
infix  60  -->   xfy.                  %%% function type constructor
infix  40  &&    yfx.                  %%% product term constructor
infix  30  @     yfx.                  %%% application

%%% Kinds for object terms and types
kind  tm   type.                       %%% meta-type for terms
kind  tp   type.                       %%% meta-type for types
```

```
%%% Type Constructors
type  mand  tp -> tp -> tp.          %%% product space
type  -->   tp -> tp -> tp.          %%% function space
type  mbool tp.                      %%% meta-type bool
type  mint  tp.                      %%% meta-type int
type  mlist tp -> tp.                %%% meta-type list constructor

%%% Term Constructors
type  lamb  (tm -> tm) -> tm.        %%% Lambda Abstraction
type  cond   tm -> tm -> tm -> tm.   %%% Conditional
type  &&    tm -> tm -> tm.          %%% Products
type  @     tm -> tm -> tm.          %%% Application
type  let   (tm -> tm) -> tm -> tm.  %%% Let
type  fix   (tm -> tm) -> tm.        %%% Fixed Point
type  #     int -> tm.               %%% Coerces int to mint

%%% Primitives/Constants
type  tt    tm.                      %%% true
type  ff    tm.                      %%% false
type  fst   tm.                      %%% first projection
type  snd   tm.                      %%% first projection
type  ::    tm.                      %%% list constructor
type  nil   tm.                      %%% empty list
type  hd    tm.                      %%% head-of-list function
type  tl    tm.                      %%% tail-of-list function
type  empty tm.                      %%% empty-list function
type  ==    tm.                      %%% equality function
type  !=    tm.                      %%% inequality function
type  lt    tm.                      %%% integer <
type  gt    tm.                      %%% integer >
type  plus  tm.                      %%% integer +
type  minus tm.                      %%% integer -
type  times tm.                      %%% integer *

%%%   End of module mldecl   %%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     mini-ML : Type Inference                                      %%%
%%%     (infer Gamma Exp Type) succeeds if Exp has type               %%%
%%%     Type in "context" Gamma.                                      %%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module mltype.

import mldecl lists.

type  infer   (list (pair tm tp)) -> tm -> tp -> o.


%%% Numbers in parentheses correspond to the numbering given to the
%%% rules in figure 3.

infer Gamma (# N) mint.                              %%% (1) Numbers

infer Gamma tt mbool.                                %%% (2,3) Boolean constants
infer Gamma ff mbool.

infer Gamma (cond E1 E2 E3) T :-                     %%% (4) Conditional
        infer Gamma E1 mbool,
        infer Gamma E2 T,
        infer Gamma E3 T.

infer Gamma (E1 && E2) (T1 mand T2) :-               %%% (5) Product Introduction
        infer Gamma E1 T1,
        infer Gamma E2 T2.

infer Gamma (lamb M) (T1 --> T2) :-                  %%% (8) Abstraction
        pi C\(infer [(pair C T1)|Gamma] (M C) T2).

infer Gamma (E1 @ E2) T2 :-                          %%% (9) Application
        infer Gamma E1 (T1 --> T2),
        infer Gamma E2 T1.

infer Gamma (let M E2) T1 :-                         %%% (10) Let
        infer Gamma E2 T2,
        infer Gamma (M E2) T1.

infer Gamma  (fix M) T :-                            %%% (11) Recursion
        pi C\(infer [(pair C T)|Gamma] (M C) T).


%%% primitive operators %%%

infer Gamma fst ((T1 mand T2) --> T1).               %%% (6) First Projection
infer Gamma snd ((T1 mand T2) --> T2).               %%% (7) Second Projection

infer Gamma :: (T --> (mlist T) --> (mlist T)).      %%% list constructor
infer Gamma hd ((mlist T) --> T).                    %%% head of list
```

```
infer Gamma tl ((mlist T) --> (mlist T)).        %%% tail of list
infer Gamma nil (mlist T).                       %%% the empty list
infer Gamma empty ((mlist T) --> mbool).         %%% empty-list function

infer Gamma == (T --> T --> mbool).              %%% term equality
infer Gamma != (T --> T --> mbool).              %%% term inequality
infer Gamma lt (mint --> mint --> mbool).        %%% integer <
infer Gamma gt (mint --> mint --> mbool).        %%% integer >
infer Gamma plus (mint --> mint --> mint).       %%% integer +
infer Gamma minus (mint --> mint --> mint).      %%% integer -
infer Gamma times (mint --> mint --> mint).      %%% integer *


%%% Assumed Types (instead of '=>') %%%

infer Gamma C T :-
        member (pair C T) Gamma.

%%%   End of module mltype   %%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%        mini-ML : Evaluation Procedure                          %%%
%%%        (eval Exp V) succeeds if Exp evaluates to V.            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module mleval.

import mldecl lists.

type eval             tm -> tm -> o.
type evalbinaryop     tm -> tm -> tm -> tm -> o.
type evalunaryop      tm -> tm -> tm -> o.


%%% Numbers in parentheses correspond to the numbering given to the
%%% rules in figure 5.

eval (# N)  (# N).                               %%% (1) Numbers

eval tt tt.                                      %%% (2,3) constants
eval ff ff.
```

```
eval nil nil.

eval (cond E1 E2 E3) C :-                    %%% (4,5) Conditional
        eval E1 B,
        (B = tt, eval E2 C, ! ;
                 eval E3 C).

eval (E1 && E2) (C1 && C2) :-                %%% (6) Product
        eval E1 C1,
        eval E2 C2.

eval (lamb M) (lamb M).                      %%% (9) Abstraction

eval (E1 @ E2) C :-                          %%% (10) Application
        eval E1 (lamb M),
        eval E2 C2,
        eval (M C2) C.

eval (let M E2) C :-                         %%% (11) Let
        eval E2 C2,
        eval (M C2) C.

eval (fix M) C :-                            %%% (12) Recursion
        eval (M (fix M)) C.


%%% Primitive unary operators %%%
eval (Op @ E1) C :-
        member Op [fst, snd, hd, tl, empty],
        eval E1 C1,
        evalunaryop Op C1 C.

%%%  Primitive binary operators %%%
eval (Op @ E1 @ E2) C :-
        member Op [::, ==, !=, lt, gt, plus, minus, times],
        eval E1 C1,
        eval E2 C2,
        evalbinaryop Op C1 C2 C.

evalunaryop fst (C1 && C2) C1.               %%% (7) First Projection
evalunaryop snd (C1 && C2) C2.               %%% (8) Second Projection
evalunaryop hd  (:: @ C1 @ C2) C1.           %%% head of list
evalunaryop tl  (:: @ C1 @ C2) C2.           %%% tail of list
evalunaryop empty nil tt.                    %%% empty-list function
evalunaryop empty (:: @ C1 @ C2) ff.         %%%   ""

evalbinaryop :: C1 C2 (:: @ C1 @ C2).        %%% list construction
evalbinaryop == C C tt.                      %%% term equality
```

```
evalbinaryop == C1 C2 ff.                          %%%   ""
evalbinaryop != C C ff.                            %%% term inequality
evalbinaryop != C1 C2 tt.                          %%%   ""
evalbinaryop lt (# N1) (# N2) T :-                 %%% integer <
        (truth is (N1 < N2), T = tt, ! ; T = ff).
evalbinaryop gt (# N1) (# N2) T :-                 %%% integer >
        (truth is (N1 > N2), T = tt, ! ; T = ff).
evalbinaryop plus (# N1) (# N2) (# N) :-           %%% integer +
        N is (N1 + N2).
evalbinaryop minus (# N1) (# N2) (# N) :-          %%% integer -
        N is (N1 - N2).
evalbinaryop times (# N1) (# N2) (# N) :-          %%% integer *
        N is (N1 * N2).

%%%   End of module mleval   %%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%        mini-ML: Examples                                           %%%
%%%        Examples for testing type inferencing                       %%%
%%%            and evaluator.                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module mltest.

import mltype mleval.

%%% FUNCTION DEFINITIONS %%%

%%% factorial function
fact1 (fix Fact\ (lamb N\ (cond (== @ N @ (# 0))
                                (# 1)
                                (times @ N @ (Fact @ (minus @ N @ (# 1)))))))).

%%% append function
app1 (fix App\ (lamb K\ (lamb L\
                (cond (== @ K @ nil)
                      L
                      (:: @ (hd @ K) @ (App @ (tl @ K) @ L)))))).

%%% mutual recursion example: even/odd functions
evenodd1 (fix EO\
```

```
          ((lamb X\ (cond (== @ X @ (# 0))
                          tt
                          ((snd @ E0) @ (minus @ X @ (# 1))))) &&
           (lamb X\ (cond (== @ X @ (# 0))
                          ff
                          ((fst @ E0) @ (minus @ X @ (# 1)))))).

%%% TEST CASES %%%
test1 T :- fact1 F, infer [] F T.
test2 T :- fact1 F, infer [] (let Fact\(Fact @ (# 3)) F) T.
test3 V :- fact1 F, eval  (let Fact\(Fact @ (# 3)) F) V.
test4 T :- app1 A, infer [] A T.
test5 T :- app1 A, infer [] (let App\(App @ (:: @ (# 1) @ (:: @ (# 2) @ nil))
                                         @ (:: @ (# 3) @ (:: @ (# 4) @ nil))) A)
                            T.
test6 V :- app1 A, eval  (let App\(App @ (:: @ (# 1) @ (:: @ (# 2) @ nil))
                                         @ (:: @ (# 3) @ (:: @ (# 4) @ nil))) A)
                            V.
test7 T :- evenodd1 F, infer [] F T.
test8 V :- evenodd1 F, eval (let E0\((fst @ E0) @ (# 3)) F) V.

%%%   End of module mltest   %%%
```

Appendix C: λProlog Source Code for Mini-ML to CAM Translation

Comments similar to those given for the mini-ML example apply to this example as well. We do not provide a signature for the CAM here but only give the λProlog module corresponding to figure 7. We number the clauses in this module with the number of the corresponding proof rule.

We again must resort to supplying a context with our proof rules. For this example the context is a list of pairs (`tm`, `int`) associating identifiers to the depth at which they were bound. The three proof rules in figure 7 that use the schema for $\Rightarrow$ (9,11,12) correspond exactly to the three rules in our implementation that extend the context. The clauses `generate_path` generate the code ($cfst^n$; $csnd$) and the de Bruijn index in calculated as before, only now we must refer to the context for the value D_1.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%        ML-CAM Translation                                          %%%
%%%        This version uses de Bruijn indexing to                     %%%
%%%        generate code for identifiers.                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module camml.

import mldecl camdecl lists.


type  translate        tm -> prog -> o.
type  trans            (list (pair tm int)) -> int -> tm -> com -> o.
type  transop          tm -> oper -> o.
type  generate_path    int -> com -> o.
type  member           A -> (list A) -> o.



%%% (translate ML CAM) translates mini-ML expression 'ML' into CAM program
%%% 'CAM'.

translate E (program C) :-
        trans [] 0 E C.



trans Gamma Depth (# N) (quote (## N)).          %%% (1) Numbers

trans Gamma Depth tt (quote ctt).                %%% (2) true
```

```
trans Gamma Depth ff (quote cff).                    %%% (3) false

trans Gamma Depth (cond E1 E2 E3)                    %%% (5) Conditional
      (push & C1 & (branch C2 C3)) :-
        trans Gamma Depth E1 C1,
        trans Gamma Depth E2 C2,
        trans Gamma Depth E3 C3.

trans Gamma Depth (E1 && E2)                         %%% (6) Product
      (push & C1 & swap & C2 & cons) :-
        trans Gamma Depth E1 C1,
        trans Gamma Depth E2 C2.

trans Gamma Depth (Op @ E) (push & C & Camop) :-     %%% (7,8) Projections
        member Op [fst, snd],
        trans Gamma Depth E C,
        transop Op Camop.

trans Gamma Depth  (lamb E)  (cur C) :-              %%% (9) Lamb
        NewDepth is Depth + 1,
        pi K\(trans [(pair K NewDepth)|Gamma]
                    NewDepth  (E K) C).

trans Gamma Depth (let M E2)                         %%% (11) Let
      (push & C2 & cons & C1) :-
        trans Gamma Depth E2 C2,
        NewDepth is Depth + 1,
        pi K\(trans [(pair K NewDepth)|Gamma]
                    NewDepth  (M K) C1).

trans Gamma Depth (fix M) (push & (rec C)) :-        %%% (12) Fix
        NewDepth is Depth + 1,
        pi K\(trans [(pair K NewDepth)|Gamma]
                    NewDepth  (M K) C).

                                                     %%% (10) Application
trans Gamma Depth (Op @ E1 @ E2)                     %%% binary primitives
      (push & C1 & swap & C2 & cons & (op Camop)) :-
        member Op [==, !=, ::, plus, minus, times],
        trans Gamma Depth E1 C1,
        trans Gamma Depth E2 C2,
        transop Op Camop.

trans Gamma Depth (Op @ E1)                          %%% unary primitives
      (push & C1 & (op Camop)) :-
        member Op [hd, tl, empty],
        trans Gamma Depth E1 C1,
```

```
        transop Op Camop.

trans Gamma Depth (E1 @ E2)                    %%% General Apply
      (push & C1 & swap & C2 & cons & app) :-
        trans Gamma Depth E1 C1,
        trans Gamma Depth E2 C2.

trans Gamma Depth K C :-                       %%% (4) Identifier
        member (pair K D) Gamma,
        Distance is (Depth - D) + 1,
        generate_path Distance C.


%%% translations for primitive operators %%%
transop ==    cequal.
transop !=    cnequal.
transop ::    ccons.
transop plus  cplus.
transop minus cminus.
transop times ctimes.
transop hd    chd.
transop tl    ctl.
transop empty cempty.


%%% translate N+1 into (cfst^N & csnd) %%%
generate_path 1 csnd.
generate_path N (cfst & C) :-
        N1 is N - 1,
        generate_path N1 C.

%%%   End of module camml   %%%
```