Chapter 24
A Meta-Logic for Functional Programming

John Hannan *Dale Miller*

The University of Pennsylvania

Abstract

We define a meta-logic to serve as a formal framework in which meta-programming tasks for a simple functional language can be elegantly specified. Highlighting the relationships among meta-programming, logic programming and natural deduction, we consider both practical and theoretical concerns of program analysis and thus motivate our methods. Then, using techniques inspired by structural operational semantics and natural semantics, we investigate how, in a natural deduction setting, we can specify a wide variety of tasks that manipulate functional programs as data objects. Specifications of this sort are presented as sets of inference rules and are encoded as clauses in a higher-order, intuitionistic meta-logic. Program properties are then proved by constructing proofs in this meta-logic. We argue that the meta-logic provides clear and concise specifications that suggest intuitive descriptions of the properties or operations being described. In particular, the rich structure of functional programs, including a variety of binding and scoping mechanisms, can be naturally represented and analyzed. We support this claim by providing three example specifications for simple meta-programming tasks. From a practical standpoint, the meta-logic can be implemented naturally in a logic programming language and thus we can produce experimental implementations of the specifications. We expect that our efforts will provide new perspectives and insights for program manipulation tasks.

24.1 Introduction

Meta-programming, in its most general setting, is any programming task in which programs are treated as data objects. We typically distinguish between the meta-language, in which we write the meta-programs, and the object-language, in which the programs being manipulated are written. These two languages may in fact be the same language (giving rise to meta-circular interpreters, for example) but in general they can be two unrelated languages. Considering this definition of meta-programming we observe that many common programs or procedures can be classified as meta-programs: (*i*) editors treat programs as objects that are to be modified; most editors do not treat object programs as a special data type (distinct from arbitrary text), though some recent programming systems include editors in which they are [27]; (*ii*) compilers treat programs as the source and target objects of a translation process; (*iii*) interpreters treat programs as input data and produce as output the result of executing the program. A more narrow definition of meta-programming includes only meta-interpreters providing extensions to languages and program transformers, e.g., partial evaluators and abstract interpreters. In this chapter we assume the former, more encompassing, definition of meta-programming. This choice does not greatly affect the issues that arise in this study, but merely enlarges the set of examples from which we can select.

The focal point of this work is the definition and examination of a meta-logic for specifying meta-programming tasks over functional programs. We cannot hope to be completely general in this goal as we must make certain choices during our investigations: the choice of the functional programming language that we consider and the range of meta-programming tasks to be considered. We will use a small subset of Standard ML as our object language. Although we hope that the methods we present will accommodate a wide range of meta-programming tasks, we do not expect to define formally the limits of our methods.

The main purpose of this work is to demonstrate how, with the proper meta-logic, a natural deduction paradigm provides a suitable framework for manipulating and analyzing functional programs. Previous work has used inference rules to specify the dynamic semantics and other properties of programs [5, 25], but their emphasis has typically been more towards software engineering issues and less towards a study of the resulting proof system. We are concerned with defining and characterizing a formal meta-language via proof-theoretic methods and with understanding the nature of proofs that can be constructed in this language. From a practical standpoint this work finds immediate application in the development of programming languages and programming language environments. Such tasks require development tools that are both expressive and extensible (as well as other qualities as described below). The methods pursued in this work seem to be well suited in both regards.

An expectation of this work is that with a single meta-language, we can specify a wide variety of tasks that treat programs as objects. We have considered specifications for tasks such as evaluation, type inferencing and compilation, each presented by a set of inference rules. Additional work has considered strictness analysis and mixed evaluation. By describing these apparently disparate tasks in a unified framework we hope to gain insight into the similarities and differences among these tasks. From a practical standpoint, this uniform treatment of tasks suggests the possibility of integrating various tools. From a theoretical standpoint, a detailed analysis of a variety of tools can be performed using uniform techniques. Thus the same (meta-theoretic) analysis techniques used on the static semantics of a language could apply to a compiler for the language.

Another expectation of this work concerns an analysis of the proof-theoretic tools we describe. By exploiting our foundations in proof theory we hope to reason about meta-theoretic properties via proof transformations and manipulations. For example, we can show that certain program transformers have a correctness-preserving property by demonstrating an equivalence between certain classes of proof trees. Thus, using some well-established methods of proof theory we can express and prove important (meta-)properties of our meta-programs.

Finally, an important aspect of this work is its operational characteristic. We can interpret the meta-programming specifications as defining an operational semantics in the sense of [25]. Thus we provide an operational description, that we claim is natural and intuitive, of a variety of program analysis tasks. Such descriptions facilitate our understanding of these tasks.

The remainder of the chapter is organized as follows. In Section 24.2 we present some general background material on meta-programming, natural deduction and logic programming. We discuss the relationships between these concepts and motivate some of our decisions. In Section 24.3 we describe the issues involved in manipulating programs as first-class objects. In Section 24.4 we describe a general framework for our proof systems and the methods used to encode functional programs as terms. We outline how program properties can be denoted by propositions in a suitable logic. In Section 24.5 we describe a simple functional language E and then in Sections 24.6 and 24.7 we specify a static semantics (for type inferencing) and a standard evaluation semantics for E.

24.2 Meta-Programming, Natural Deduction and Logic Programming

To help motivate our particular methods we consider several perspectives of meta-programming, both practical and theoretical. Our first point is to distinguish between meta-logic and meta-programming. Meta-programming implicitly implies some programming language is used to manipulate object programs, but it does not stipulate the properties of this meta-language. One could choose languages such as Lisp or Prolog to implement all of the meta-programming tasks that we have in mind. We are, however, concerned with the formal properties of such implementations and so we are concerned with a logic for meta-programming, which we refer to as a meta-logic. Our approach is to first define a logic (including terms, propositional formulas and methods for constructing proofs) suitable for describing manipulations and analyses on simple functional programs. Then we shall describe an implementation for this logic.

Above we described a class of programming tasks operating on programs as data and labeled these as meta-programming. This viewpoint provides one aspect to this work, that of the "high-level" tasks involved and their motivations. Equally important are two orthogonal issues: logical formalism and implementation. Logical formalism refers to the logic upon which our methods or solutions are based. From this viewpoint we expect to provide a formal basis from which we can reason about the tasks involved. We choose a natural deduction system to provide the formal or logical basis of our methods. We also wish to provide practical implementations of the meta-programs and so we must consider the perspective of implementation. We choose a logic programming paradigm to provide an implementation vehicle. The justification for these two choices is given by the intimate relationship among them and meta-programming.

24.2.1 Natural Deduction and Meta-Programming

In a natural deduction theorem prover, one thinks of constructing proofs of propositions according to a prescribed set of inference rules. These propositions are typically defined in some formal logic, e.g., first-order predicate calculus. The inference rules are typically characterized as being either introduction or elimination rules according to whether a logical connective is introduced or eliminated in the conclusion. For our application, the atomic propositions will denote statements about object programs. These statements may either be statements concerning a program property (*"program P is well-typed"*) or concerning an operation on the program (*"program P evaluates to value V"*). The inference rules for our purposes will correspond roughly to the inference rules of Gentzen's *NJ* (natural deduction for intuitionistic logic). The particular choice of terms, propositional formulas and inference rules is largely based on the object language that we consider, though the kind of program property or operation that we consider, also matters.

This connection between natural deduction and meta-programming largely originated with the work on structural operational semantics [25] and later natural semantics [4, 16]. The current work is closely related to the application of the LF (Logical Framework) system to operational semantics [3] and grew out of an effort to extend the methods of natural semantics [11]. While natural semantics provides methods for specifying many meta-programming tasks, we feel that its use of strictly first-order terms and limited types of inference figures makes the formal properties of their meta-program harder to determine. We shall represent programs instead as simply typed λ-terms and shall also extend the underlying reasoning mechanism of natural semantics with two kinds of 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.

24.2.2 Natural Deduction and Logic Programming

The connection between natural deduction and logic programming has already been exploited in the work of natural semantics where the specifications are compiled into Prolog. In general, logic programming languages provide many features that make them suitable implementation languages for natural deduction theorem provers [6]. A brief examination of these features makes this connection obvious. A foremost aspect of computation in logic programming is the search operation. Taking a procedural view of logic programming we can describe the execution of a logic program by describing a search process through a space defined by the program. Search is also an important component of natural deduction theorem proving. The task of constructing a proof can be described as the exploration of a search space. Unification is a second characteristic feature of logic programming. It provides a mechanism for matching two terms. More specifically, we can specify "generic" clauses in a program and then use these clauses in specific instances via unification. A similar mechanism plays an important role in constructing proofs in natural deduction. A natural deduction system can be specified by a set of inference rules. These rules are typically given by rule templates, i.e., ones that contain free variables. Proofs will contain only closed instances of these rules and so some matching or unification process is required to produce the required instances of these rules. Finally, most logic programming languages are constructed from clauses of the general form "$Body \supset Head$" with the intuitive reading "if *Body* is true then *Head* is true." Thus the inference rules used to specify a natural deduction system should have a natural translation into clauses of this form. The head and body of a clause will denote the consequent and the antecedent of an inference rule. This straightforward translation into clauses together with the declarative style of logic programming suggests that using logic programs can provide perspicuous implementations of natural deduction theorem provers.

24.2.3 Meta-Programming and Logic Programming

As discussed above, the characteristic feature of meta-programming is the manipulation of programs as data objects. Thus we require a language that is equipped with terms suitable for representing programs as first-class objects and with mechanisms for analyzing these terms. We will argue that an encoding of programs into simply typed λ-terms provides a convenient representation for manipulating functional programs as objects. A characteristic feature of logic programming is unification of terms and we will make a use of a particular kind unification to provide an appropriate mechanism for analyzing terms representing programs. We will also make a simple use of β-conversion for manipulating λ-terms in useful ways.

A common presentation of meta-programming is in terms of some kind of conditional rules. For example, a program transformation system, in which an input program is manipulated into another program, is typically given by a set of rules. Each rule contains conditions describing the applicability of the rule and these conditions may be in the form of tests or procedures to be performed on the incoming program. Such rules can be naturally encoded as clauses in a logic programming paradigm.

A third point considers again the search paradigm of logic programming. Meta-programming, too, is often reduced to a problem of search. In fact, part of the complexity of many meta-programming tasks is their lack of a simple deterministic algorithm. Rather, a search must be conducted, as in the case of program transformation where many rules may be applicable for a given program.

24.2.4 Natural Deduction and Sequent Systems

In his seminal paper on logical deduction Gentzen presented two intuitionistic proof systems, *LJ* and *NJ* [7]. In the system *LJ*, sequents of the form $\Gamma \longrightarrow A$, where Γ is a sequence of formulas and A is a formula, form the basic propositions. The sequent $\Gamma \longrightarrow A$ denotes the proposition: from the formulas in Γ, the formula A follows. In contrast, the basic proposition in the proof system *NJ* is a single formula: the context by which that formula is proved is left implicit. A common distinction made between the two systems is that the more notationally explicit *LJ* system is more suitable for machine implementation while the less explicit *NJ* is more suitable for human understanding.

A similar distinction between styles of operational semantics can be made. The work on natural semantics is built essentially on a sequent style proof system where sequents are of the form $\Gamma \vdash \mathcal{P}(P)$ in which $\mathcal{P}(P)$ is a predicate over program P and Γ is a set of assumptions about the identifiers (free variables) of P. The work in this paper is built essentially on natural deduction proof systems. In a sense, the difference between the two approaches is insignificant: for the examples presented in this paper a simple translation from one style to the other is possible. Each method does have certain advantages over the other, however. For example, by being more explicit, the sequent style is more closely tied to conventional implementations of functional programming languages. The set Γ often acts as a mapping from identifiers to values and such mappings can be efficiently implemented. While the natural deduction style systems, which rely on introduction and discharge rules, seem further removed from efficient implementations, they are often more easily manipulated in a formal setting. See [10] for an examples where the formal manipulation of a natural deduction style proof system was much more convenient than its corresponding sequential specification.

24.3 Programs as First-Class Objects

At the heart of meta-programming is the issue of representing programs as first-class objects and manipulating these objects. The static structure of programs, however, is inherently complex, with structure arising from a variety of binding constructs and scoping rules. Hence a meta-logic (or meta-programming language) should be suitably equipped with appropriate data structures and operations to facilitate the task of effectively representing these objects. In this section we attempt to characterize an effective representation for the kind of program analyses that we would like to specify in our meta-logic. In what follows we assume the reader to be familiar with the notion of lexical scoping.

If we consider Standard ML [13] as the prototypical functional programming language then we can immediately observe several kinds of binding operations occurring in simple programming examples. The example program in Figure 24.1 contains four different kinds of bindings. The datatype definition introduces two new identifiers: the type name **MYPAIR** and the constructor **pair**. The scope for these two identifiers is assumed to be "global" in that the definition has been added to the top-level environment. Next we have the definition of functions that introduce the identifiers **swap**, **fst** and **snd**, all denoting (function) values. The scope of the identifier **swap** is, like the two above, global, while the scope of both **fst** and **snd** is local to the definition of **swap**. Finally, each of the three function definitions introduces identifiers as formal parameters with a scope local to the associated definition: **p** is local to the definition of **swap**; instances of **x** and **y** are local to the definitions of both **fst** and **snd**.

As this small example plainly illustrates, even a simple programming language can incorporate a number of binding constraints. Here we have four, over type names, constructors, function names and formal parameters. The notion of global scope is really only one of convenience and does not need to be considered as a special case. Note that in each case above what was introduced was simply an identifier, regardless of whether it denotes a value, type, etc. Thus our meta-logic may only need a single mechanism for providing a scoping for identifiers over (a representation of) expressions to capture a wide variety of binding operations in the object language.

Consider using Standard ML itself as the meta-language for just this simple example. We could naturally define (at the meta-level) datatypes for representing expressions, declarations and bindings, as was done in the Standard ML of New Jersey compiler [1]. (In this compiler, the datatype declaration for bindings alone contains the union of nine different kinds of bindings!) In this compiler the binding information is managed via an explicit environment that performs binding, lookup and scope management functions. For the particular application of compilation, this treatment of bindings is perhaps optimal as it is efficient and easily implemented. So one conclusion possible from the experience with the Standard ML compiler is that Standard ML provides suitable facilities for representing itself

```
datatype MYPAIR = pair of int * int;

fun swap p =
  let fun fst(pair(x,y)) = x;
      fun snd(pair(x,y)) = y
  in
      pair(snd(p), fst(p))
  end;
```

Figure 24.1: Examples of Bindings in Standard ML

as data objects. Compilation, however, is only one of several meta-programming tasks we would like to consider.

Consider, then, the possible analysis or manipulation facilities that Standard ML could provide. At the primitive level of the language, only simple pattern matching is provided to decompose terms. As a simple example of how this level of sophistication is inadequate for some purposes, consider the following two expressions:

```
fun fst(pair(x,y)) = x;          fun snd(pair(y,x)) = y
```

and suppose we wish to determine if these two expressions denote the same expression. A simple check for syntactic identity would report that these two programs are different. A more sophisticated program could, of course, be written that would check for the alphabetic variants in bound variable names and conclude that these two fragments are equal. Note, however, that the two instances of the identifier `pair` could refer to different constructors (e.g., one defined over pairs of integers and one over pairs of reals), depending on the contexts in which each function was defined. The problem encountered with this example is that equality between λ-terms is typically considered modulo λ-conversion. This notion of equality is a much more complex operation than the simple syntactic equality provided naturally by pattern matching. In particular, using this notion of equality, a λ-term is equal to any alphabetic variant of itself (α-equivalence). Meta-programming equality modulo λ-conversion is a particularly appealing idea: λ-terms provide the essential de-sugared elements of functional programming, dispensing with the need for keeping tract of bound variable names.

To provide further analysis capabilities we consider unification of simply typed λ-terms as described in [14]. If the only method for manipulating λ-terms is via normalization and unification then it is impossible to distinguish between two programs which are equal modulo λ-conversion. Furthermore, unification is a sophisticated mechanism that can be used to probe the structure of programs, respecting congruence classes modulo λ-conversion. The use of λ-terms and λ-term unification to implement program manipulation systems has been proposed by various people. Huet and Lang in [15] employed second-order matching (a decidable subcase of λ-term unification) to express certain restricted, "template" program transformations. Miller and Nadathur in [18] extended their approach by adding to their scheme the flexibility of Horn clause programming and richer forms of unification. In [11] we argued that if the Prolog component of the TYPOL system [2] were enriched with higher-order features, logic programming could play a stronger role as a specification language for various kinds of interpreters and compilers.

With these ideas in mind we shall define an abstract syntax for programs and types of the object language based on the simply typed λ-calculus. We shall represent programs as simply typed terms by introducing an appropriate set of constants to a calculus 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. And for each construct that introduces a binding (of an identifier), we use a λ-abstraction where the abstracted variable denotes the bound identifier and the expression over which it is abstracted defines the scope of the binding. This uniform treatment of bindings provides a natural specification of many programming language constructs. 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 Section 24.5.

While we are only concerned in the current work with the simply typed λ-calculus, richer and more flexible λ-calculi have been proposed as a suitable representation system for programs. For example, Pfenning and Elliot in [23] have extended the simply typed λ-calculus to include simple product types. They also discuss in depth the role of *higher-order abstract syntax*, *i.e.*, the representation of programs as λ-terms, in the construction of flexible and general program manipulation systems. The LF specification language [12] uses a λ-calculus with a strong typing mechanism to specify various components of proof systems: much of this specification language could profitably be used in the context we are concerned with here [3].

Similar advantages of the blend of higher-order unification and logic programming have been exploited in systems that manipulate formulas and proofs of logical systems. Felty and Miller in [6] discuss the use of a higher-order logic programming language to specify and implement theorem provers and proofs systems. Here again, λ-terms and higher-order unification are used to represent and manipulate formulas and proofs. The Isabelle theorem prover of Paulson [22] also makes use of these features to implement flexible theorem provers.

24.4 The Meta-Logic

Having settled on the representation of programs as λ-terms we now define the propositional formulas of our logic and a core of "primitive" inference figures.

These inference figures will correspond to a subset of the inference figures of Gentzen's *NJ* system. For a specific specification (of some meta-programming task), additional inference figures will be added.

Propositions in our meta-logic are either atomic or compound, and are given the type o. The atomic propositions of our meta-logic will be constructed from a finite set of n-ary propositional symbols (typically with $n = 2$), each with a given type. Given a propositional symbol p of type (in curried form) $(\sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \sigma_n \rightarrow o)$ and typed λ-terms $t_1{:}\sigma_1, t_2{:}\sigma_2, \ldots t_n{:}\sigma_n$, then $p(t_1, t_2, \ldots t_n)$ is a proposition of type o. Compound propositions are constructed with the logical connectives $\&{:}(o \rightarrow o \rightarrow o)$, $\Rightarrow{:}(o \rightarrow o \rightarrow o)$ and $\forall{:}((\sigma \rightarrow o) \rightarrow o)$. Here, $\forall$ is polymorphic in the type variable σ. So, for example, if A_1 and A_2 are propositions then so are $(A_1 \& A_2)$, $(A_1 \Rightarrow A_2)$ and $(\forall\ \lambda x.A_2)$. We shall typically write $(\forall\ \lambda x.A)$ as $(\forall x\ A)$.

To manipulate such propositions, particularly the compound ones, the meta-logic comes equipped with four primitive inference figures, given in Figure 24.2. The first one, $(\beta\eta)$ is applicable when 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. For example an instance of this rule is

$$\frac{type(1,\ \ int)}{type((\lambda x.x\ 1),\ \ int)}$$

where *type* is some non-logical predicate constant.

The second inference figure, $(\&I)$, is called *conjunction introduction.* When using this inference rule to construct proofs 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 inference figure $(\Rightarrow I)$. 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 inference figure labeled $(\forall I)$. 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.* The corresponding discharge or elimination rules are also included in the meta-logic but are not used in any of the examples presented.

These two rules, implication and universal introduction, are not typically found in other presentations of operational semantics. We shall use these two together in the following way. For a particular meta-programming task we shall

$$\frac{A_1}{A_0}\ (\beta\eta) \qquad\qquad \frac{A_1 \quad A_2}{A_1\ \&\ A_2}\ (\&I)$$

$$\frac{\begin{matrix}(A_1)\\ \vdots \\ A_2\end{matrix}}{A_1 \Rightarrow A_2}\ (\Rightarrow I) \qquad\qquad \frac{A[x \mapsto c]}{(\forall x)A}\ (\forall I)$$

Figure 24.2: Primitive Inference Figures

introduce a new inference figure whose premise is of the form

$$\forall c(A_1(c) \ \Rightarrow\ A_2(c))$$

which we can describe operationally as follows:

> Introduce some new constant c, not occurring in A_1, A_2 or any current hypothesis; then assume property A_1 about c. From this assumption attempt to prove property A_2 for c. If such a proof can be found then discharge the assumption $A_1(c)$ and the parameter c.

We shall see how this kind of reasoning provides a powerful and natural way of describing aspects of meta-programming tasks, particularly when dealing with object-level bound variables.

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. Following the convention of specifying proof systems, we interpret these inference figures as schemas, in that the inference figures may contain free variables that get instantiated to specific instances. (We assume all capitalized identifiers denote variables.) Thus we take the universal closure of each inference figure. When providing examples of inference figures later in this chapter, we shall drop references to the connective & in premises. Inference figures of the form

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

Notice that we have not explicitly included the corresponding elimination rules $((\& E)$, $(\Rightarrow E)$ and $(\forall E))$. For all the examples in this paper, the inclusion of these rules is not necessary.

A proof in this language will be understood in the standard sense of proofs in natural deduction. For more information on natural deduction and its terminology (both of which are used in this chapter) see [7, 26].

We shall view the construction of a proof of a proposition as a kind of computation. Rarely shall we be particularly interested in the actual proof constructed, but rather in some instantiation of existentially quantified variables. This is accomplished by starting with propositions containing free variables and assuming that they are existentially quantified. For example, we may have a proposition of the form $type(1, T)$ which can be interpreted as the query "*what is type type of 1?*" Constructing a proof of this proposition should result in the instantiation of the existentially quantified variable T to some ground term, e.g., *int*.

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 *Prop* would be replaced by a proposition $\Gamma \longrightarrow Prop$, in which Γ is used to store assumptions. This is, for example, used in natural semantics to handle contexts. 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*** [19] has (partially) been implemented in the λProlog system [21]. 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 corresponding to the universal closure of inference schemas. Queries to construct proofs of propositions will become goals in λProlog, so the example query above becomes

$$?\!-type(1, T)$$

and a successful computation of this goal results in reporting the answer substitution: $T = int$. Instead of using the λProlog syntax to present example inference rules in later sections, we shall continue to use the more graphically oriented inference figures. All the examples presented here have been implemented and tested in a version of λProlog.

24.5 Abstract Syntax as Lambda Terms

Having defined our meta-logic, including its terms, propositional formulas and inference figures, we now describe how a simple functional programming language can be encoded as terms in our meta-logic. Of course, there may be many possible ways of representing programs as terms, but we want one that will allow us to make full use of the meta-logic. We distinguish between concrete syntax, which may provide a convenient representation for human understanding, and abstract syntax, which contains the essential information needed for program manipulation. In our introduction to handling binding information we have already hinted at what a good abstract representation should include. Here we make these ideas precise by starting with a simple language, namely the pure untyped λ-calculus (λ^u). Later we extend this language to into a more substantial subset of Standard ML. This presentation demonstrates how an abstract syntax for a functional language can be constructed using simply typed lambda terms and how this abstract syntax captures the binding and scoping constructs found in functional programming languages. We take care in making the distinction between terms and types at the object (λ^u) level and terms and types at the meta-level. We refer to the latter as meta-terms and meta-types.

Suppose that the concrete syntax for λ^u is given by the following grammar:

$$U \ ::= \ x \ | \ \lambda x.U \ | \ (U \ U).$$

(We must be careful here because we shall overload the use of the symbol λ.) We introduce a new meta-type *tm* for representing (at the meta-level) terms of λ^u. Now there is a standard way of encoding untyped terms into the simply typed λ-calculus ($\lambda^{\rightarrow}$) and this is described in [17]. The idea is to introduce two new constants into the typed calculus ($\lambda^{\rightarrow}$):

$$\begin{aligned} \Psi &: (tm \rightarrow tm) \rightarrow tm \\ \Phi &: tm \rightarrow (tm \rightarrow tm) \end{aligned}$$

We can then define a simple mapping $(\cdot)^* : \lambda^u \longrightarrow \lambda^{\rightarrow}$ as follows:

Definition 1 ($(\cdot)^*$). *For any $M \in \lambda^u$ let $(M)^*$ be*

$$\begin{aligned} (x)^* &= x^*{:}tm \quad \text{for } x \text{ a variable.} \\ (MN)^* &= \Phi M^* N^* \\ (\lambda x.M)^* &= \Psi(\lambda x^*{:}tm.M^*) \end{aligned}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
C & : & tm & & & \\
if & : & tm \to tm \to tm \to tm & int & : & tp \\
@ & : & tm \to (tm \to tm) & bool & : & tp \\
lamb & : & (tm \to tm) \to tm & \to & : & tp \to tp \to tp \\
let & : & (tm \to tm) \to tm \to tm & & & \\
fix & : & (tm \to tm) \to tm & & &
\end{array}$$

Figure 24.3: Signature for Terms and Types of E

We assume that $(\cdot)^*$ defines a bijective mapping of untyped variables to typed variables (of type tm). For a more complete discussion of this encoding, including soundness and completeness results, see [8]. As an example of this encoding, consider the untyped term $\lambda x \lambda y(xy)$. Via this encoding its corresponding typed term is $\Psi(\lambda x(\Psi(\lambda y(\Phi\, x\, y))))$. Note that this is a term of type tm. Thus we have a simple way of representing any pure untyped λ-term as a simply typed term of uniform type. This is important because in the task of type inference, to be discussed in the next section, we must be able to handle terms that are both well-typed and untypable.

We now consider a slightly larger programming language. Let E be the functional language whose concrete syntax is defined by the following grammar:

$$\begin{array}{rl} E \; ::= & \mathbf{C} \mid \mathbf{x} \mid \mathbf{if}\ E\ \mathbf{then}\ E\ \mathbf{else}\ E \mid (E\,E) \mid \\ & \boldsymbol{\lambda}\mathbf{x}.E \mid \mathbf{let}\ \mathbf{x} = E\ \mathbf{in}\ E \mid \mathbf{fix}\ \mathbf{x}.E \end{array}$$

Here, x ranges over variables and **C** ranges over primitive constants, typically including the integers and booleans and a set of primitive operations to manipulate them.

To define our abstract syntax for E we follow an approach similar to the above one for $\lambda^u$and in the same spirit as [23]. We begin by giving a signature for some meta-terms that we use to construct terms and types at the object level. (See Figure 24.3.) Notice that the constants *lamb*, *let* and *fix* are higher-order, that is, they each require a functional argument of type $tm \to tm$. In the examples that follow M will be used as a higher-order (meta-)variable of this meta-type. '$\to$' is the function space constructors for tp. We have overloaded the symbol '$\to$', using it at both the object and meta levels; its use, however, should always be clear from context. The object types we consider are only monotypes (in the sense of [20] as we do allow type variables). In the next section we present a separate discussion of manipulating polytypes.

Using the signature of Figure 24.3 we can build up λ-terms forming an abstract syntax for E as follows. For constants and variables in the concrete syntax we just introduce associated constants and variables of type tm to the abstract syntax.

For the **if** statement we introduce the new constant *if* such that given three terms $e_1, e_2, e_3 : tm$, then $(if\ e_1\ e_2\ e_3) : tm$. Application is made explicit with the infix operator '@' so that $e_1@e_2$ represents the expression denoted by the term e_1 applied to e_2. For lambda abstraction we introduce the constructor *lamb* that takes a meta-level abstraction of the form $\lambda x.e$, in which x and e are of meta-type *tm*, and produces a term of type *tm*. For example the concrete syntax for lambda abstraction is $\boldsymbol{\lambda}\mathbf{x}.\boldsymbol{E}$ and its abstract syntax is $(lamb\ \lambda x.e)$ (for e the λ-term corresponding to E). Similar to *lamb*, the *let* construct uses a meta-term M of the form $\lambda x.e$ to represent the binding of an identifier. Thus the concrete syntax **let x** $=$ E_1 **in** E_2 is given by the abstract term $(let\ \lambda x.e_2\ e_1)$ in which e_1 and e_2 are the (abstract) terms denoting the expressions E_1 and E_2, respectively. To represent the recursive **fix** construct we introduce the *fix* constant which again uses an explicit abstraction to capture the binding. An example of this construction is given below.

Throughout most of this chapter we will avoid discussing primitive operations such as $+$, $-$, *etc.* They are, of course, important to have in the full language but including them here is neither difficult nor illuminating. We shall typically assume, for the sake of examples, that we at least have some basic set of list operations. In the following and subsequent examples we systematically drop the apply "@" operator in order to make examples more readable.

Consider the following expression that defines the append function and then applies it to two lists.

let app = (fix f.λk.λl.(if empty(k) then l else cons(hd(k) f(tl(k) l))))
in (app [1] [2]).

The corresponding term in the abstract syntax is

$$(let\ \ \lambda app.(app\ \ (cons\ 1\ nil)\ \ (cons\ 2\ nil))$$
$$(fix\ \lambda f(lamb\ \lambda k(lamb\ \lambda l(if\ (empty\,k)\ l\ (cons\ (hd\ k)\ (f\ \ (tl\,k)\ \ l))))))).$$

Note how the four bindings in the concrete syntax (**app**, **f**, **k**, **l**) are translated into explicit λ-abstractions in the abstract syntax.

Before presenting some example specifications we recall the distinction we made earlier between natural deduction and sequent style systems. Now that our abstract syntax has been defined further comment concerning the difference between our method and typical approaches to natural or operational semantics is appropriate. This distinction concerns the treatment of identifiers. The typical approach to programs analysis 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, as suggested earlier, 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 [24]. There two different evaluation functions were defined: the awkward *Eval* function defined in terms of closures and the simpler *eval* defined 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.

In the following two sections we present some meta-programming examples using our abstract syntax.

24.6 Static Semantics

Static semantics refers to a class of program analyses that provide information about programs based on their static structure (i.e., not considering their behavior during some form of evaluation). One common example of a static semantics is type inferencing. An example of this kind of analysis is given below. Other kinds of static analysis include type checking, certain kinds of flow analysis and possibly complexity analysis.

24.6.1 Type Inference

We introduced the language E as an implicitly typed functional language, in the same vein as Standard ML. Thus an important static operation on E programs is type inference. More specifically, we only wish to admit programs generated by the given grammer for E that are "well-typed." By this we mean that a type can be given to the program according to some laws. The idea of using inference rules to specify type inference is not new. Most recently Tofte has given a thorough treatment of polymorphic type inference in an operational semantics style [28].

$$c \xrightarrow{ty} \mathcal{C}(c) \qquad \frac{e_1 \xrightarrow{ty} bool \qquad e_2 \xrightarrow{ty} \tau \qquad e_3 \xrightarrow{ty} \tau}{(\mathit{if}\ e_1\ e_2\ e_3) \xrightarrow{ty} \tau} \qquad (1,2)$$

$$\frac{(\forall c)\ (c \xrightarrow{ty} \tau_1 \ \Rightarrow\ (M\ c) \xrightarrow{ty} \tau_2)}{(\mathit{lamb}\ M) \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)} \qquad \frac{e_1 \xrightarrow{ty} (\tau_1 \rightarrow \tau_2) \qquad e_2 \xrightarrow{ty} \tau_1}{(e_1 @ e_2) \xrightarrow{ty} \tau_2} \qquad (3,4)$$

$$\frac{e_2 \xrightarrow{ty} \tau_2 \qquad (M\ e_2) \xrightarrow{ty} \tau_1}{(\mathit{let}\ M\ e_2) \xrightarrow{ty} \tau_1} \qquad \frac{(\forall c)\ (c \xrightarrow{ty} \tau \ \Rightarrow\ (M\ c) \xrightarrow{ty} \tau)}{(\mathit{fix}\ M) \xrightarrow{ty} \tau} \qquad (5,6)$$

Figure 24.4: Type Inference for E

The specification for type inference in E is given in Figure 24.4. We introduce the infix propositional symbol $\xrightarrow{ty} : tm \rightarrow tp \rightarrow o$ and construct propositions of the form $e \xrightarrow{ty} \tau$ where τ is a λ-term built up from the constants *int*, *bool*, etc. and $\rightarrow$. The proposition $e \xrightarrow{ty} \tau$, in which e is a closed term denoting the abstract syntax of functional program E and τ is a closed term denoting the abstract syntax of a type, states that e has type τ. We assume that we have a fixed map $\mathcal{C}$ from the abstract constants to types, such that for each base constant c of the abstract syntax, $\mathcal{C}(c) = \tau$. Clause 1 of the specification types the constants using the map $\mathcal{C}$. The next clause 2 gives the typing for the conditional statement. Clause 3 is the typing rule for lambda abstraction and it is a bit different from the usual typing rule using environments [28]. In the environment approach, typing the 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 $(\mathit{lamb}\ M)$ we first pick a new constant c and assume it has type τ_1 (i.e., we introduce the assumption $c \xrightarrow{ty} \tau_1$). 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 4 is the usual typing rule for application. Clause 6 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 5 requires some explanation. The more standard implementation of type inference for *let* first infers a 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 5 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 use of β-reduction 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.

We can view this proof system as a declarative specification for type checking problems. Given a *closed* proposition of the form $e \xrightarrow{ty} \tau$, finding a proof of this proposition asserts that the type of (the expression denoted by) e is τ. Of course we would like to have type inference algorithm to which we supply the open proposition $e \xrightarrow{ty} T$. Numerous works have shown that type inference can be accomplished by unification. We apply these ideas by exploiting our logic programming implementation for our meta-logic that comes equipped with unification. Thus by posing the query ?$-$ $e \xrightarrow{ty} T$, for some closed e, to the logic program corresponding to this specification, unification resolves all the type constraints imposed by the inference rules. Note that the resulting answer substitution θ may not be ground, i.e., $\theta(T)$ may contain free type variables. For example the result of the query

$$?- \ (lamb\ \lambda x.x) \xrightarrow{ty} T$$

would have T instantiated to $t \rightarrow t$ for some type variable t. We have no explicit rule for quantifying over type variables but we may implicitly assume the expression to denote the type $\forall t.t \rightarrow t$.

24.6.2 The Subsumes Relation for Polytypes

As a second example of using our meta-language to manipulate ML-like types, we present a proof system for the subsumes relation on polytypes [20]. 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 $\rightarrow$) 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 provide an auxiliary definition.

DEFINITION 2 (Instance of a Polytype). *τ is an instance of polytype (forall $\lambda t_1(\ldots(forall\ \lambda t_n(\tau'))\ldots)$) if there exists some 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 3 (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 24.5. The first clause states the obvious: any polytype subsumes itself. The second 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.

24.7 Dynamic Semantics

Dynamic semantics refers to a class of program analyses that provide information about programs based on a dynamic behavior, i.e., some set of evaluation

$$\sigma \sqsubseteq \sigma \qquad \frac{(\forall c)\ \sigma_1 \sqsubseteq (M\ c)}{\sigma_1 \sqsubseteq (\mathit{forall}\ M)} \qquad \frac{(M\ x) \sqsubseteq \sigma_2}{(\mathit{forall}\ M) \sqsubseteq \sigma_2}$$

Figure 24.5: Subsumes Relation for Polytypes

rules is assumed and the behavior of programs under these rules is considered. In this section we present a standard evaluation semantics that provides a declarative specification for an E interpreter. Other, non-standard semantics, including strictness analysis and mixed evaluation are also possible [9].

We would like to specify the evaluation of expressions in E, based on a simple interpreter for the language. (We say standard here to distinguish from a non-standard semantics.) 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 $e \longrightarrow \alpha$ in which e is an expression of the object language and α is the result of evaluating e. Informally, the terms appearing to the left of $\longrightarrow$ denote expressions and the terms appearing to the right are the "values" or meanings of the expressions. 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 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 E, using the same abstract syntax as given in Section 24.5. As with the type inference specification we introduce a new infix propositional symbol $\xrightarrow{se} : tm \rightarrow tm \rightarrow o$. Propositions in our system are of the form $e \xrightarrow{se} \alpha$ in which e and α are expressions in E and α is the result of "evaluating" e. Proofs of these propositions are constructed from the proof system given in Figure 24.6. The first rule treats the constants of the language as just evaluating to themselves. The next two rules treat the *if* expression in a natural way: the conditional part, e_1 must evaluate to *true* or *false* for a proof to be found. Rule (3) states that an abstraction evaluates to itself. In the rule for application (4), meta-level β-reduction correctly captures the notion of function application (with a call-by-value semantics). Similar comments apply to our rule for *let* (5). In the rule for recursion (6) we introduce a fixed point operator with its intuitive operational semantics (i.e., unfolding). This again makes explicit use of meta-level β-reduction as the meta-term 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, given by M. Static scoping is ensured with this specification 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

$$c \xrightarrow{se} c \tag{1}$$

$$\frac{e_1 \xrightarrow{se} \mathit{true} \qquad e_2 \xrightarrow{se} \alpha}{(\mathit{if}\ e_1\ e_2\ e_3) \xrightarrow{se} \alpha} \qquad \frac{e_1 \xrightarrow{se} \mathit{false} \qquad e_3 \xrightarrow{se} \alpha}{(\mathit{if}\ e_1\ e_2\ e_3) \xrightarrow{se} \alpha} \tag{2a, 2b}$$

$$(\mathit{lamb}\ M) \xrightarrow{se} (\mathit{lamb}\ M) \tag{3}$$

$$\frac{e_1 \xrightarrow{se} (\mathit{lamb}\ M) \qquad e_2 \xrightarrow{se} \alpha_2 \qquad (M\ \alpha_2) \xrightarrow{se} \alpha}{(e_1 @ e_2) \xrightarrow{se} \alpha} \tag{4}$$

$$\frac{e_2 \xrightarrow{se} \alpha_2 \qquad (M\ \alpha_2) \xrightarrow{se} \alpha}{(\mathit{let}\ M\ e_2) \xrightarrow{se} \alpha} \qquad \frac{(M\ (\mathit{fix}\ M)) \xrightarrow{se} \alpha}{(\mathit{fix}\ M) \xrightarrow{se} \alpha} \tag{5, 6}$$

Figure 24.6: Standard Evaluation Semantics for E

abstraction.

The values implicitly defined by this specification (i.e., the set of terms that can appear to the right of $\xrightarrow{se}$) are just the set of constants, lambda abstractions and primitive constructors. In general, the set of values may not always be a subset of the language (as is the case in [16]). Now given some closed expression e we can think of evaluating e by finding some value α such that $e \xrightarrow{se} \alpha$ is provable. We assume some non-deterministic search procedure is used to find such an α and construct such a proof.

24.8 Summary

We have presented a meta-logic for the analysis and manipulation of functional programs. Using a higher-order, intuitionistic meta-logic we encoded axioms and inference rules as clauses in this logic. The expressive power of this logic provides us the ability to specify, in a natural and formal setting, a variety of program manipulation tasks (e.g., type inferencing, evaluation and compilation) as proof systems. This formal setting distinguishes our approach, as a meta-logic, from more ad hoc methods of meta-programming. Though not discussed here, existing methods from proof theory, as pertain to natural deduction, often provide a natural basis for performing meta-theoretic analyses of these proof systems.

We presented several examples to support our claim that this meta-logic permits a high-level and elegant specification of program manipulations. From the perspective of program specifications, we argued that the proof rules provided in this meta-logic were more perspicuous than, for example, a first-order logic, and we did not need to introduce any non-logical meta-level operations to implement all the examples considered. An important aspect of practical meta-programming systems is the compiling of the specifications (e.g., inference rules) into efficient programs. Although we see no reason to believe that the specifications given here could not be implemented efficiently, it seems probable that such compiling will be more involved than it is for compiling specifications written in a first-order meta-logic.

The ability of a meta-language to "scale-up" to richer languages is also important. The language that we considered was only a simple one providing a small subset of Standard ML. In particular, we did not include datatype definitions, exceptions, modules, etc. We have, however, used our meta-logic to represent and manipulate an enriched language containing datatype definitions (both concrete and abstract). The binding and scoping of the datatype constructors is handled in a manner similar to our treatment of bound variables: New (higher-order) constants are added to the abstract syntax for constructing terms denoting the introduction of data constructors and type names to expressions. For abstract datatypes, the concealment of the structure of the datatype is naturally handled using meta-level abstractions in λ-terms. Additional binding and scoping facilities exist in Standard ML (e.g., modules) and future work will explore the applicability of our methods to these.

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.

The first author is supported in part by a fellowship from the Corporate Research and Architecture Group, Digital Equipment Corporation, Maynard, MA USA. Both authors are supported in part by grants NSF CCR-87-05596, ONR N00014-88-K-0633, and DARPA N00014-85-K-0018.

References

[1] A. Appel and D. MacQueen. A standard ML compiler. In G. Kahn, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, Springer-Verlag LNCS, Vol. 274, 1987.

[2] P. Borras *et. al.* *CENTAUR: the System.* Technical Report 777, INRIA, December 1987.

[3] R. Burstall and Furio Honsell. A natural deduction treatment of operational semantics. In *Foundations of Software Technology and Theoretical Computer Science*, pages 250–269, Springer-Verlag LNCS, Vol. 338, 1988.

[4] 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.

[5] D. Clément, J. Despeyroux, L. Hascoët, and G. Kahn. *Natural Semantics on the Computer*. Research Report 416, INRIA, June 1985.

[6] 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.

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

[8] J. Hannan. Notes on embedding the untyped λ-calculus in a simply typed λ-calculus. 1988. (Unpublished notes).

[9] J. Hannan. Proof theoretic methods for analysis of functional programs. December 1988. Dissertation Proposal, University of Pennsylvania, Technical Report MS-CIS-89-07.

[10] J. Hannan and D. Miller. Deriving mixed evaluation from standard evaluation for a simple functional language. In *Proceedings of the International Conference on Mathmatics of Program Construction*, Springer-Verlag, 1989. (to appear).

[11] J. Hannan and D. Miller. *Enriching a Meta-Language with Higher-Order Features*. Technical Report MS-CIS-88-45, University of Pennsylvania, June 1988.

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

[13] R. Harper, R. Milner, and M Tofte. *The Semantics of Standard ML, Version 2*. Technical Report, Edinburgh University, 1988.

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

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

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

[19] 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.

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

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

[22] L. Paulson. The foundation of a generic theorem prover. (To appear in the *Journal of Automated Reasoning*).

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

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

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

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

[27] T. Reps. *Generating Language-Based Environments.* MIT Press, 1985.

[28] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1987.