Extended natural semantics

JOHN HANNAN[†]

Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark

Abstract

We extend the definition of natural semantics to include simply typed λ-terms, instead of first-order terms, for representing programs, and to include inference rules for the introduction and discharge of hypotheses and eigenvariables. This extension, which we call *extended natural semantics*, affords a higher-level notion of abstract syntax for representing programs and suitable mechanisms for manipulating this syntax. We present several examples of semantic specifications for a simple functional programming language and demonstrate how we achieve simple and elegant manipulations of bound variables in functional programs. All the examples have been implemented and tested in λProlog, a higher-order logic programming language that supports all of the features of extended natural semantics.

Capsule review

The author describes a generalization of Plotkin and Kahn's notation for the specification of programming languages using inference rules. The extension comprises two closely related steps: higher-order abstract syntax and hypothetical and schematic judgments as introduced by Martin–Löf.

Higher-order abstract syntax permits λ-expressions in an underlying language of terms in order to represent variable binding. As the author illustrates, this leads to a significant simplification of many language specifications, since common concepts such as renaming of bound variables or capture-avoiding substitution are now directly supported by the representation language. To fully exploit the added expressive power we also need to generalize the language of inference rules to accomodate schematic judgments (which introduce parameters into deductions) and hypothetical judgments (which permit deductions from hypotheses). The author shows how these two extensions allow very elegant formulations of the operational semantics, type system, and other aspects of mini-ML.

Kahn's original motivation for natural semantics came from the goal of building a meta-environment for programming languages. In this context, specifications should be executable to some extent, and the logic programming language TYPOL implements natural semantic specifications within the Centaur system. Hannan observes that the generalization to extended natural semantics still allows execution of semantic specifications in a suitably enriched logic programming language. All the examples in the paper can consequently also be read as pure λ-Prolog programs which implement mini-ML.

The primary contribution of this paper is to demonstrate how techniques from type theory

† Supported by a grant from the Danish Natural Science Research Council under the DART Project. Author's current address: Department of Computer Science, The Pennsylvania State University, 333 Whitmore Laboratory, University Park, PA 16802, USA. email: hannan@cs.psu.edu.

and higher-order logic programming can be brought to bear on practical problems in the area of programming language specification and meta-environments.

Contents

1 Introduction

Natural semantics provides a simple and elegant means for specifying many aspects of programming language semantics, including evaluation, type checking and compilation (Kahn, 1987). This approach to specifying programming languages has been used successfully in the definition of Standard ML (Milner *et al.*, 1991) and in the construction of an interactive programming system (Borras *et al.*, 1987). While natural semantics specifications of programming languages can be considered high-level, they use very simple data structures for representing programs and other objects, and they use only very simple formulas and inference rules for expressing program properties. This simplicity has the advantage of yielding straightforward and efficient implementations of the specifications, but it also has the obvious disadvantage of sometimes yielding specifications that contain primitive encodings of program properties that obscure some of the logical nature of these properties.

Addressing this disadvantage, we extend natural semantics in two directions. First we consider a more expressive representation of programs by an abstract syntax using simply typed λ-terms. This choice allows us to represent various notions of binding and scoping, typically found in high-level languages, in a purely syntactic manner (rather than relying on side conditions or explicit axiomatic descriptions). Second, we consider a richer logic for specifying program properties. This extension is partially dictated by our choice of abstract syntax as we demonstrate later on. Quantification over higher-order variables and equality of λ-terms modulo $\beta\eta$-equivalence are two of the more important aspects of the logic used in the kind of specifications presented in this paper. We argue that this extension to natural semantics provides a suitable framework for presenting higher-level specifications of program properties.

We can trace two lines of research that have merged in this work: structural operational semantics and higher-order logic for meta-programming. The phrase 'structural operational semantics' is attributed to Plotkin, based on his seminal paper *A Structural Approach to Operational Semantics* (Plotkin, 1981). In that paper he describes evaluation in terms of a one-step reduction relation and his inference rules axiomatize this relation. Natural semantics provides further development of these ideas, and has been used as the basis for an interactive programming system at INRIA (Kahn, 1987). Though using only a small subset of first-order logic, they develop proof systems for a variety of meta-programming tasks including evaluation, type inference and compilation. The work reported in this paper grew directly out of an attempt to extend natural semantics to a setting using a higher-order meta-logic.

One of the earliest applications of a higher-order logic for meta-programming is found in Huet and Lang (1978), in which functional programs are specified as simply typed λ-terms and program transformations are applied via second-order matching. Templates (terms containing free variables) are used to represent a class of functional programs (all those that match the template via some substitution for the free variables). This idea is developed further by including it in the programming language λProlog (Miller and Nadathur, 1987), a language based on a fragment of higher-order intuitionistic logic, where methods richer than template matching alone can be implemented. Closely related to this work is the Isabelle theorem prover (Paulson, 1989), where essentially the same logic is used to implement flexible theorem provers. Using a dependent type theory based on Martin-Löf type theory and following ideas first found in Automath (Bruijn, 1980), LF (Harper *et al.*, to appear) and Elf (Pfenning, 1989) provide similar means for manipulating programs via operational semantics (Burstall and Honsell, 1991). A relationship between LF and our logic is established in Felty and Miller (1990), in which LF signatures are translated into logic programs.

Extended natural semantics, like natural semantics, can be directly implemented in a logic programming language. Unfortunately, Prolog, based on first-order Horn clauses, does not directly support our extensions to natural semantics. In particular, it provides neither a primitive notion of $\beta\eta$-equality for simply typed λ-terms nor an implementation of the natural deduction rules for the introduction and discharge of hypotheses and eigenvariables. The higher-order programming language λProlog does support these features. It provides an implementation of a restricted theorem prover for extended natural semantics and hence a mechanism for interpreting our specifications as programs. Thus, we can experiment with prototype implementations of our semantic specifications coding them directly as λProlog programs and executing these programs.

The remainder of this paper is organized as follows. In section 2 we present the terms of extended natural semantics. We first describe the issues involved in manipulating programs as first-class objects and the idea of abstract syntax. By presenting several representative examples, we illustrate some of the desired features of any abstract syntax used for representing programs. We then present an abstract syntax for an example functional language that supports these desired features. In section 3 we describe a logic that we use to construct propositions and inference rules for specifying operational semantics. We describe how specifications of program properties can be given as either sets of formulas in this logic or as sets of inference rules. In section 4 we present several specifications of meta-programming tasks including type inference and call-by-value evaluation for a simple functional language. In section 5 we summarize our results and discuss directions for future work.

2 The terms of extended natural semantics

Natural semantics uses first-order terms to represent programs and all other data structures associated with specifications. While such a representation is often convenient to manipulate and efficient to implement, it can also be too primitive. For example, specifications may require the use of side conditions to enforce restrictions on the structure of terms.

Before introducing the terms for extended natural semantics we list some desiderata for a good, high-level abstract syntax, that the term structure should support:

1. *Binding and Lexical Scoping:* a programming language can contain a number of binding or scoping constraints involving identifiers. For example, we might have constructs that introduce bindings ranging over type names, constructors, function names and formal parameters. Though a concrete syntax may contain a variety of binding constructs, an abstract syntax could conceivably use just a single construct for specifying all of these. Such uniformity and simplicity would contribute to advantages of manipulating terms in an abstract syntax rather than a concrete syntax.
2. *Handling Substitution:* substitution is a fundamental operation used to specify evaluation and other dynamic operations of functional programming languages. It is commonly written as $b[e/x]$ where b and e are terms and x is a variable. Two issues arise during the process of substitution: (i) only free occurrences of x in b should be replaced with e; (ii) free variables of e should not become bound after substitution into b. Substitution can require a change of bound variable names and also a means for matching occurrences of a variable with its binder. Because all the meta-programming tasks that we consider are invariant under α-conversion and substitution requires changing bound variable names, a suitable theory for reasoning about terms in an abstract syntax should probably equate such terms up to α-equivalence.
3. *Variable Occurrence Restriction:* for several kinds of program analyses, information about occurrences of bound variables in terms provides valuable information for meta-programs. For example, programs that specify explicit

allocation and deallocation of storage rely on notions of 'fresh' variables or unused storage locations. If we assume that storage cells are represented as variables by our program, then the allocation of a new cell requires that the variable chosen to represent this cell must not occur in the current representation of state for the evaluator. Similarly, if some form of garbage collection is provided by the program, then tests to determine that certain cells (variables) are no longer accessible can be accomplished by considering variable restrictions.

We claim that simply typed λ-terms support a representation of programs suitable for meeting these desiderata in a clear and high-level fashion. Below we describe the basis for an abstract syntax using such terms, followed by a larger example.

2.1 A higher-order abstract syntax

We use a representation of programming languages adapted from a standard encoding of the untyped λ-calculus into a simply typed λ-calculus (Scott, 1980; Meyer, 1981). In this encoding every untyped λ-term is represented as a term of some distinguished type, e.g. *tm*, such that all the untyped λ-abstractions are encoded as typed λ-abstractions in which the abstracted variable is of type *tm*. (We call *tm* a meta-level type.) We can extend this idea to build a representation for a programming language in which typed λ-abstractions provide a uniform representation for all binding and scoping constructs of the programming language. Using this approach we can capture all notions of α-convertibility and substitution in a programming language via the operations of α-convertibility and β-reduction in a typed calculus.

The essence of this encoding, momentarily restricted to just the untyped λ-calculus, is as follows. First, we introduce a base type *tm*. Next we introduce two new constants,

$$app : tm \rightarrow tm \rightarrow tm, \qquad abs : (tm \rightarrow tm) \rightarrow tm,$$

which we use to encode untyped applications and λ-abstractions as typed terms. We can then define a simple translation from terms in the untyped calculus to terms (of type *tm*) in the typed calculus including these two constants. For any untyped λ-term e, the encoding of e, $(e)^*$, is defined inductively as:

$$\begin{aligned} (\boldsymbol{x})^* &= x{:}tm \quad \text{for } \boldsymbol{x} \text{ a variable} \\ (e_1\, e_2)^* &= (app\ (e_1)^*\ (e_2)^*) \\ (\boldsymbol{\lambda x}.e)^* &= (abs\ \lambda x{:}tm.(e^*)) \end{aligned}$$

We assume that every untyped variable $\boldsymbol{x}$ maps to a corresponding typed variable $x{:}tm$. Note that the range of this map consists only of normal-form terms of type *tm*. One way of viewing this encoding is that it maps α-equivalent classes of terms to $\alpha\beta\eta$-equivalent classes of terms. As argued above, we often wish to reason about programs up to α-equivalence and we can do this, via this encoding, by considering the $\beta\eta$-normal forms of terms of type *tm*.

A useful property of encoded λ-terms, that we repeatedly exploit, is the following:

Proposition 2.1
Let $(abs\ f)$ and t be the encodings of some terms $\lambda x.e$ and e', respectively. Then $(f\ t) =_{\beta\eta} (e[e'/x])^{*}$.

A proof of this can be found in Hannan (1991).

In this abstract syntax we eliminate the need for α-conversion and encode substitution as normalization. There are, however, other kinds of operations that we wish to perform on programs. In particular, we often wish to examine the structure of programs. To provide such analysis capabilities we consider unification of simply typed λ-terms as described in Huet (1975). Using unification we can examine the structure of terms (representing programs) by unifying such terms with templates, terms containing free variables. The existence of a substitution unifying a term representing a program and such a template can indicate that the program has a certain property (e.g. is tail recursive). Applying the substitution to similar templates can produce new programs with close relationships to the original program. Using unification of simply typed λ-terms to implement program manipulation systems has been proposed by various people. In Huet and Lang (1978), second-order matching (a decidable subcase of λ-term unification) is used to express certain restricted, template program transformations. In Miller and Nadathur (1987) this approach is extended by adding the flexibility of Horn clause programming and richer forms of unification. In Hannan and Miller (1988) we first argued that if the Prolog component of the TYPOL system (Borras *et al.*, 1987) were enriched with higher-order features then logic programming could play a stronger role as a specification language for various kinds of interpreters and compilers.

2.2 An abstract syntax for a fragment of standard ML

To illustrate the use of higher-order abstract syntax (HOAS) to capture notions of scope, binding and substitution, we present an abstract syntax for a fragment of Standard ML, called mini-ML, a core language that we use later in some meta-programming examples. We choose a representative collection of constructs from the language definition and, using the observations made previously about the nature of HOAS, we give a direct representation of these constructs in this syntax. Though other representations (in HOAS) are clearly possible and much of the motivation for our syntax relies on its use in specifying evaluation, we believe that the examples given later illustrate the clarity and elegance of HOAS.

We consider a simple core programming language by introducing some additional constants and constructors to the untyped λ-calculus. Then for each programming language construct we introduce a new constant to a simply typed calculus which is used to build an abstract term representing this construct. 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. Thus the relationships among various binding operations (and the occurrences of bound variables) in our programming language are captured by similar relationships among λ-bound variables in our abstract

c	:	tm			
if	:	$tm \rightarrow tm \rightarrow tm \rightarrow tm$			
$pair$	:	$tm \rightarrow tm \rightarrow tm$	int	:	ty
fst	:	$tm \rightarrow tm$	$bool$	:	ty
snd	:	$tm \rightarrow tm$	$arrow$	:	$ty \rightarrow ty \rightarrow ty$
app	:	$tm \rightarrow tm \rightarrow tm$	$cross$	:	$ty \rightarrow ty \rightarrow ty$
abs	:	$(tm \rightarrow tm) \rightarrow tm$			
let	:	$(tm \rightarrow tm) \rightarrow tm \rightarrow tm$			
fix	:	$(tm \rightarrow tm) \rightarrow tm$			

Fig. 1. Signature for terms and types of mini-ML.

syntax. This uniform treatment of bindings provides a natural specification of many programming language constructs.

We use a slight variant of the language introduced in Clément *et al.* (1986). Let mini-ML be the functional language whose concrete syntax is defined by the following grammar:

$$\begin{aligned} E \quad ::= \quad & \mathrm{C} \mid \mathrm{x} \mid \textbf{if } E \textbf{ then } E \textbf{ else } E \mid \\ & \textbf{pair } E\ E \mid \textbf{fst } E \mid \textbf{snd } E \mid \\ & E\,E \mid \lambda x.E \mid \textbf{let } x = E \textbf{ in } E \mid \textbf{fix x.}E \mid (E) \end{aligned}$$

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. We consider λ, **let**, and **fix** to be binding operations, binding the variable in the corresponding subterms. As an example, consider the following expression that defines the addition function and then applies it to two numbers:

$$\begin{aligned} &\textbf{let add = (fix f.}\boldsymbol{\lambda}\textbf{x.}\boldsymbol{\lambda}\textbf{y.(if (zerop x) then y else (s (f (pred x) y))))} \\ &\qquad\quad \textbf{in (add (s (s z)) (s (s z))).} \end{aligned}$$

This example assumes the primitive constants **z** and **s** for representing natural numbers and the primitive functions **zerop** (test for zero) and **pred** (predecessor). Object-level types are not part of the Mini-ML language but are used to classify expressions (e.g. this example has type **integer**). We postpone any discussion of object-level types until later.

Our abstract syntax for mini-ML is a simple extension of the one given for untyped λ-terms and in the same spirit as Pfenning and Elliot (1988). We begin by giving a signature for some constants which we use to construct terms and types at the object level (see Fig. 1). Notice that the constants *abs*, *let* and *fix* are higher-order, that is, they each require a functional argument of type $tm \rightarrow tm$. Note that object types are given a first-order representation as terms of type ty.

Using the signature of Fig. 1, we can build up λ-terms forming an abstract syntax for mini-ML as follows. We have already introduced the syntax for application and abstraction using the constants *app* and *abs*. We can introduce a number of primitive constants and operators such as those in the example above. For each constant **c** in our object language we introduce a corresponding constant $c{:}tm$ to our abstract syntax. For the **if** statement we introduce the new constant *if* such that if e_1, e_2 and e_3 (each of type *tm*) encode the concrete terms **e1**, **e2** and **e3**, respectively then $(if\ e_1\ e_2\ e_3) : tm$ encodes the concrete term **if e1 then e2 else e3**.

For purposes of abstract syntax, the **let** statement can be viewed as a mechanism for locally introducing an identifier with an associated value to an expression. For example, the following expression

$$\textbf{let x = y + x in (f x) end}$$

introduces the identifier **x** as a variable with value **y + x** to the body of the let expression **(f x)**. A **let** definition is not recursive so, for example, the second (textual) occurrence of **x** refers to some variable that must exist outside of this **let** statement. Furthermore, the introduction of the new identifier **x** effectively hides this previous occurrence in the body of the **let**, i.e. the free occurrence of **x** in the expression **(f x)** refers to this **let**-bound variable. The example above translates to the term

$$(let\ \ (\lambda x.f\ x)\ \ (app\ (app\ plus\ y)\ x)).$$

The scoping of the introduced identifier x is clear from the use of λ-abstraction.

Recursive function definitions provide another binding mechanism that introduce new identifiers. Consider the representation of the addition example given above:

$$\begin{aligned}&(let\ \ \lambda add(app\ \ (app\ \ add\ \ (app\ s\ (app\ s\ z)))\ \ (app\ s\ (app\ s\ z)))\\ &\qquad (fix\ \lambda f(abs\ \lambda x(abs\ \lambda y(if\ \ (app\ zerop\ x)\ \ y\\ &\qquad\qquad\qquad\qquad (app\ s\ (app\ (app\ f\ (app\ pred\ x))\ y))))))).\end{aligned}$$

Note how the four bindings in the concrete syntax (**add**, **f**, **x**, **y**) are translated into explicit λ-abstractions in the abstract syntax.

3 The logic of extended natural semantics

To motivate the logic that we introduce in this section we consider the simple task of axiomatizing the definition for checking that a term $t : tm$ represents a pure (untyped) λ-term, i.e. it is constructed from only the constants *app* and *abs* (and bound variables). We introduce the predicate symbol $pure : tm \to o$. We axiomatize this definition according to the possible structure of t. The case for application is straightforward:

$$\frac{\text{pure } E_1 \qquad \text{pure } E_2}{\text{pure } (app\ E_1\ E_2)}$$

But what about the case when t is a λ-abstraction, represented as a term $(abs\ f)$, or when t is a bound variable? Recall that such an f is a term of type $tm \to tm$ and, furthermore, it is equal to any other term g such that $f =_{\alpha\beta\eta} g$. To determine that $(abs\ f)$ is pure we must 'descend through' the abstraction and check that the only free variable in the body of the abstraction is the bound variable of the abstraction. Alternatively we can check that if occurrences of this bound variable are considered pure then the body of the abstraction must be pure. Just considering the types of terms suggests an inference rule of the form

$$\frac{\mathit{pure}\ (E\ X)}{\mathit{pure}\ (\mathit{abs}\ E)},$$

with some assumption that X is pure, for some term $X : \mathit{tm}$. But the choice of X is crucial because we must ensure that X is pure and that it does not already occur in E. We will provide a logic suitable for accomplishing these goals in a simple and direct manner.

The logic of natural semantics (Kahn, 1987) is insufficient to ensure such conditions above for X without using awkward side conditions. We briefly review this logic to understand its shortcomings. In natural semantics, inference rules have the following form. Each rule contains a numerator and a denominator. The denominator of a rule is a single formula, called the conclusion. The numerator of a rule is an unordered collection of formulas called the premises of the rule. Formulas come in two varieties: *sequents* and *conditions*. The conclusion of a rule is necessarily a sequent. Conditions, occurring in the numerator, generally serve to limit the applicability or instances of a rule by restricting the occurrences of certain variables or requiring that some relation hold between instances of variables, etc. A sequent has two parts, separated by a turnstile ($\vdash$): an *antecedent* (on the left) and a *consequent* (on the right). The consequent is an atomic proposition and the antecedent is just a term that typically represents information about the free (object-level) variables occurring in the consequent. The connection to natural deduction becomes apparent if we consider the antecedents of sequents as representing the set of undischarged hypotheses in a deduction of the consequent.

While natural semantics can be implemented efficiently, it is not suitable for manipulating programs represented in a higher-order abstract syntax. For example, there is no direct way of completing the definition of *pure* above, without resorting to side-conditions or implicit assumptions that defeat the purpose of the higher-order syntax. Below we consider a more general class of formulas that extends natural semantics and overcomes this problem.

3.1 Formulas

Let S be a finite, non-empty set of non-logical primitive types (sorts) and let o be the one logical primitive type, the type of formulas (o is not a member of S). A *type* will be either o, a member of S, or a functional type $\tau \rightarrow \sigma$ in which both τ and σ are types. The function type constructor associates to the right so $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ and $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ are equivalent as types. The order of a type, $\mathcal{O}(\tau)$, is the measure of how deeply function types are nested to the left in τ and it is defined in the following way:

- $\mathcal{O}(o) = 0$.
- if $\tau \in S$ then $\mathcal{O}(\tau) = 0$;
- if $\tau = \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ for $\tau_0 \in S \cup \{o\}$ then $\mathcal{O}(\tau) = \mathit{max}(\{\mathcal{O}(\tau_i) | i \in 1 \ldots n\}) + 1$

So $\mathcal{O}(\tau) = 1$ exactly when τ is of the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ such that each τ_i $(0 \leqslant i \leqslant n)$ is a primitive type.

The logical constants are $\& : o \rightarrow o \rightarrow o$ (conjunction) and $\Rightarrow : o \rightarrow o \rightarrow o$ (implication) and $\forall_\tau : (\tau \rightarrow o) \rightarrow o$ (universal quantification of type τ) for every type τ that does not contain any occurrences of o. A *signature* is a finite set Σ of typed, non-logical constants. We often enumerate signatures by listing their members as pairs, written $a : \tau$, in which a is a constant of type τ. We restrict occurrences of o in the types of non-logical constants: if a constant c in Σ has type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$, in which $n \geqslant 0$ and τ_0 is primitive type, then the types $\tau_1, \ldots, \tau_n$ may not contain o. If τ_0 is o then c is called a *predicate*. If a constant has a type of order 0 or 1, then we say call that constant *first-order*; otherwise we call it *higher-order*. A signature is *nth-order* if all its constants are of order n or less and at least one constant in it is of order n. In the sequel, we use only first-order and second-order signatures. We have found that second-order signatures are often sufficient for representing the terms of a higher-order syntax and formulas manipulating these terms.

We extend the general notion of typed λ-terms to Σ-terms and Σ-formulas for a signature Σ. If $a : \tau \in \Sigma$ then a is a Σ-term of type τ. If t is a λ-term containing at most constants from Σ and the logical constants $\&$, $\Rightarrow$ and $\forall_\tau$ and typable according to the standard rules for the simply typed calculus then t is a Σ-term. If the type of t is o then t is a Σ-formula. For readability we write the logical constants $\&$ and $\Rightarrow$ in infix form and we write $\forall_\tau x\ t$ for the expression $\forall_\tau(\lambda x\ t)$, often omitting the type when it can be determined from context. Furthermore, we abbreviate a sequence of quantifiers, $\forall x_1 \forall x_2 \cdots \forall x_n\ t$ $(n \geqslant 0)$ as $\forall \bar{x}\ t$.

As the predicate symbols of a signature Σ and logical connectives are just constants added to the simply typed λ-calculus we can define the $\beta\eta$-equivalence relation over formulas as:

- $(p\ s_1\ s_2 \ldots s_n) \cong (p\ t_1\ t_2\ \ldots t_n)$ iff $s_i =_{\beta\eta} t_i$ for $i = 1 \ldots n$;
- $(A_1 \& A_2) \cong (B_1 \& B_2)$ iff $A_1 \cong B_1$ and $A_2 \cong B_2$;
- $(A_1 \Rightarrow A_2) \cong (B_1 \Rightarrow B_2)$ iff $A_1 \cong B_1$ and $A_2 \cong B_2$;
- $(\forall_\tau x\, A) \cong (\forall_\tau y\, B)$ iff $A[z/x] \cong B[z/y]$ for some $z{:}\tau$ not free in A or B.

Thus we can consider a propositional formula as representing an equivalence class of formulas and we typically choose the one in $\beta\eta$-normal form as the canonical representative for each class. Normal forms for formulas must exist since the formulas are just terms in a simply-typed calculus with a few added constants.

To manipulate propositional formulas we use the inference system given in Fig. 2. The first two inference rules are *conjunction introduction* and *conjunction elimination*. The remaining rules treat the introduction and discharge of objects in a proof. To specify the introduction and discharge of assumptions needed to prove hypothetical propositions we use the inference figures ($\Rightarrow I$) (*implication introduction*) and ($\Rightarrow E$) (*implication elimination*). Using implication introduction we can prove $A_1 \Rightarrow A_2$ by first assuming that there is a proof of A_1 and then building 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. Using implication elimination, from proofs of $A_1 \Rightarrow A_2$ and A_1 we can prove A_2.

$$\frac{A_1 \quad A_2}{A_1 \,\&\, A_2}\ (\&I) \qquad\qquad \frac{A_1 \,\&\, A_2}{A_i}\ (\&E)\ i = 1,2$$

$$\begin{array}{c} (A_1) \\ \vdots \\ \dfrac{A_2}{A_1 \Rightarrow A_2} \end{array}\ (\Rightarrow I) \qquad\qquad \frac{A_1 \Rightarrow A_2 \qquad A_1}{A_2}\ (\Rightarrow E)$$

$$\frac{A[y{:}\tau/x]}{\forall_\tau x\, A}\ (\forall I) \qquad\qquad \frac{\forall_\tau x\, A}{A[t{:}\tau/x]}\ (\forall E)$$

Fig. 2. Primitive inference figures.

To specify the introduction and discharge of eigenvariables we use the inference figures ($\forall I$) (*universal introduction*) and ($\forall E$) (*universal elimination*). The introduction rule has the usual proviso that the eigenvariable y does not occur free in A or in any open assumption. To prove a universally quantified formula $\forall_\tau x\, A$ (using ($\forall I$)) we must prove a generic instance of the formula $A[y{:}\tau/x{:}\tau]$ in which y is an eigenvariable that satisfies the proviso. Using the elimination rule we can prove any instance of a (proven) universally quantified formula.

The introduction and elimination rules for implication and universal quantification are not found in natural semantics, but they find quite natural uses in specifying manipulations of terms in our abstract syntax. Implication introduction provides a simple means for introducing local assumptions to a proof. This is a generalization of the typical use of environments in which local information about variables, typically relations between variables and values, is maintained. Using implication, however, provides a direct means of expressing a logical relationship between bound variables and the terms in which they occur. Additionally, we are free to introduce compound propositional formulas as assumptions. Introducing compound formulas of the form $A_0 \Rightarrow A_1$ can be viewed as introducing locally new inference rules. A call-by-name evaluator can be expressed simply by introducing a rule of this form to express the lazy evaluation of arguments. In the next section we present an example which uses such locally scoped inference rules. Note that we have not included disjunction and existential quantification in our logic. We could include them, with some restrictions regarding their occurrence in formulas, but we have found that we do not need them for the kinds of specifications we wish to write.

3.2 Specifications as formulas

A theory or logical specification is given as a pair $(\Sigma, \mathscr{P})$ in which Σ is a signature and $\mathscr{P}$ is a set of closed Σ-formulas representing axioms. We will often leave Σ implicit. A proof constructed from $\mathscr{P}$ will be understood in the standard sense of proofs in natural deduction. If $\mathscr{P}$ names a particular collection of axioms and proposition B is provable from this collection (using the inference rules of Fig. 2) then we write $\mathscr{P} \vdash B$. For more information on natural deduction and its terminology see Gentzen (1969) and Prawitz (1965).

Note that the logic defined here is a particularly weak subset of higher-order intuitionistic logic. For example quantification of the form $(\forall_\tau x\, A)$ is restricted to types τ *not containing* the type o. Hence we have no predicate quantification. Furthermore, for all the examples presented here, the type of τ will be at most second order.

As an example we complete the specification of the *pure* relation that introduced this section. We define a pair $(\Sigma, \mathscr{P})$ as:

$$\begin{aligned}
\Sigma &= \{abs : (tm \to tm) \to tm,\ \ app : tm \to tm \to tm,\ \ pure : tm \to o\} \\
\mathscr{P} &= \{\forall E_1 : tm\, \forall E_2 : tm\, ((pure\, E_1\ \&\ pure\, E_2) \Rightarrow pure\,(app\, E_1\, E_2)), \\
&\qquad \forall E : (tm \to tm)\, ((\forall x : tm(pure\ x \Rightarrow pure\,(E\, x))) \Rightarrow pure\,(abs\, E))\ \}.
\end{aligned}$$

The negative occurrences of universal quantification provides a means for manipulating bound variables of our abstract syntax. Consider the abstract term $(abs\ \lambda x.t)$ and suppose we wish to manipulate or examine the subterm t. Because we equate terms modulo α-equivalence, any examination or manipulation of t must be independent of the bound variable name x. So α-equivalent terms of the syntax must translate or map to an α-equivalence class of some propositional formula (the 'semantics'). We use the rule for universal introduction as a means of introducing a new variable that can be substituted for the bound variable x. If y is this new variable (universally quantified), then (the normal form of) $(\lambda x.t)y$ is the term in which y has been substituted for x in t. As an example, consider the following proof that $\lambda x.x$ is pure:

$$\dfrac{\dfrac{\overline{\forall E:(tm\to tm)\,(\forall x:tm(pure\,x \Rightarrow pure\,(E\,x)) \Rightarrow pure\,(abs\,E))}}{\forall x:tm(pure\,x \Rightarrow pure\,x) \Rightarrow pure\,(abs\,\lambda x.x)}\,\forall E \qquad \dfrac{\dfrac{pure\,y}{pure\,y \Rightarrow pure\,y}\Rightarrow I}{\forall x:tm(pure\,x \Rightarrow pure\,x)}\,\forall I}{pure\,(abs\,\lambda x.x)}\Rightarrow E$$

3.3 Specifications as inference systems

Instead of presenting specifications as sets of formulas we can use an alternative style in which they are given as sets of inference rules, consisting of a collection of atomic propositions denoting axioms and a collection of inference figures, none of which introduce the symbols &, $\Rightarrow$ or $\forall$. The idea is that a formula of the form $\forall \bar{x}(A_1 \Rightarrow A_2)$ with A_2 atomic can be viewed as an inference rule

$$\frac{A_1}{A_2}$$

with the variables $\bar{x}$ implicitly universally quantified (the schema variables of the inference rule). If we restrict our attention to provability of only atomic formulas then any set $\mathscr{P}$ of formulas, constructed from atomic formulas and only the logical connectives &, $\Rightarrow$, and $\forall$, can be translated into an equivalent set $\mathscr{P}'$ in which every formula is of the form $\forall\overline{x}(A_1 \Rightarrow A_2)$ with A_2 atomic. Here equivalence means that both sets define the same theory for *atomic* formulas. This translation is based on the following equivalences (intuitionistically provable) between formulas in which A_1, A_2, A_3 are arbitrary formulas in our logic:

$$\begin{aligned}
A_1 \Rightarrow \forall x A_2 &\equiv \forall x(A_1 \Rightarrow A_2) \quad \text{provided } x \text{ not free in } A_1\\
\forall x(A_1 \& A_2) &\equiv (\forall x A_1) \& (\forall x A_2)\\
A_1 \Rightarrow (A_2 \& A_3) &\equiv (A_1 \Rightarrow A_2) \& (A_1 \Rightarrow A_3)\\
A_1 \Rightarrow (A_2 \Rightarrow A_3) &\equiv (A_1 \& A_2) \Rightarrow A_3
\end{aligned}$$

Given any collection of formulas $\mathscr{P}$, we construct the corresponding inference system as follows. First we translate $\mathscr{P}$ into $\mathscr{P}'$ using the equivalences above as directed (left to right) rewrite rules until none of the rules are applicable. (It is trivial to show that such rewriting is strongly normalizing and confluent.) Next we replace each formula of $\mathscr{P}'$ of the form $A_1 \& A_2 \& \cdots \& A_n$, in which none of the A_i are conjunctions, with the collection $A_1, A_2, \ldots, A_n$. The result is a collection of formulas, each of the form $\forall\overline{x}(A_1 \Rightarrow A_2)$ or $\forall\overline{x}(A_2)$ for some atomic formula A_2. Note that the sequence of universal quantifiers can be empty. Then for each formula $\forall\overline{x}(A_1 \Rightarrow A_2)$ in $\mathscr{P}'$ we construct the inference rule $\dfrac{A_1}{A_2}$ and for each formula $\forall\overline{x}(A_2)$ in $\mathscr{P}'$ we construct the axiom $\dfrac{}{A_2}$. The universal quantifications become implicit. The collection of all such inference rules, together with those in Fig. 2, shall be called $\mathscr{P}^*$. If a formula A is provable from these rules we write $\mathscr{P}^* \vdash A$. If A is atomic then we have $\mathscr{P} \vdash A$ iff $\mathscr{P}^* \vdash A$. Depending on the structure of the formulas in the set $\mathscr{P}$, some of the inference rules from Fig. 2 may never be required to prove atomic formulas from $\mathscr{P}^*$. For example, if $\mathscr{P}$ contains no negatively occurring implications then neither of the implication rules need be included in $\mathscr{P}^*$.

When providing examples of inference figures, we shall drop references to the connective & in premises when they are not within the scope of any other logical connective (except for outermost universal quantifiers). Inference figures of the form

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

Translating the set of formulas for the *pure* relation produces the following inference rules:

$$\frac{\forall x(\mathit{pure}\ x \ \Rightarrow\ \mathit{pure}\ (E\ x))}{\mathit{pure}\ (\mathit{abs}\ E)} \qquad\qquad \frac{\mathit{pure}\ E_1 \qquad \mathit{pure}\ E_2}{\mathit{pure}\ (\mathit{app}\ E_1\ E_1)}$$

in which the capitalized letters denote implicitly universally quantified variables. Using these rules, we can construct the following proof that the term $\lambda x.x$ is pure:

$$\dfrac{\dfrac{\dfrac{\mathit{pure}\ y}{\mathit{pure}\ y \;\Rightarrow\; \mathit{pure}\ y}\,{\Rightarrow}I}{\forall x(\mathit{pure}\ x \;\Rightarrow\; \mathit{pure}\ x)}\,\forall I}{\mathit{pure}\ (\mathit{abs}\ \lambda x(x))}$$

In comparison to the previous proof using the specifications-as-formulas paradigm, this proof does not require the use of elimination rules for $\forall$ and $\Rightarrow$.

The choice between presenting a specification as a set of formulas or a set of inference rules is largely one of notational convenience, as both approaches are equally expressive (for the restricted logic we are considering). In the former case the formulas represent the only axioms and the only inference rules are those from Fig. 2. In the latter case, the set of inference rules contains axioms plus other inference figures, in addition to those of Fig. 2. Note that proofs in classical (or intuitionistic or minimal) logic using specifications-as-formulas are isomorphic to proofs using specifications-as-inference-rules. The isomorphism can be explained approximately as the need to use explicit elimination rules ($\forall E$, $\Rightarrow E$, $\& E$) in the proofs using specifications-as-formulas. In the sequel we present specifications as sets of inference rules, as this graphical representation is more readable.

3.4 An implementation of the logic

Meta-logical specifications can be interpreted as logic programs and the literature on implementing logic programs can be directly applied to provide implementations of specifications. If a specification is first-order and contains no embedded implications, Prolog or the TYPOL language of the CENTAUR system (Kahn, 1987) can provide a depth-first interpreter of it. Because the higher-order logic programming language λProlog (Nadathur and Miller, 1988) supports higher-order quantification, λ-conversion, and embedded universal quantification and implication, it can be used to give a depth-first implementation of the full meta-logic. λProlog can therefore be used to provide implementations of all the specifications described in the following sections. By translating the specifications in this paper into λProlog programs, we have been able to experiment with them. We have found such prototyping and experimentation valuable in understanding the dynamics of various specifications.

Instead of using λProlog syntax to present examples, we continue use the more graphically oriented inference figures. All the examples presented here have been implemented and tested in eLP (Elliott and Pfenning, 1989), an implementation of λProlog.

4 Specifications in extended natural semantics

In this section we present specifications for several meta-programming tasks including type inference and evaluation. We use mini-ML as the object language and use the abstract syntax introduced in section 2. Two of these specifications, type inference and evaluation, describe a static and dynamic semantics for the language, respectively. A third generalizes the dynamic semantics and a fourth defines a translation from our higher-order syntax to a first-order one.

4.1 Type inference

Mini-ML is an implicitly typed language. To establish that a program is well-typed we axiomatize a relation between programs and types. This example follows the one given for mini-ML in Kahn (1987), in which a natural semantics specification of type checking is given. That example uses sequents of the form $\rho \vdash e : \tau$ in which ρ is an environment, e is a program and τ is a type. Environments provide an encoding of hypotheses for typing information for free variables, typically in terms of pairs $(x : \tau)$. To type the term $\lambda x.e$ in some environment ρ we introduce the type assignment $x : \tau_1$ to the environment; then prove the type of e in this new environment to be τ_2; and then finally conclude the type of $\lambda x.e$ to be $\tau_1 \rightarrow \tau_2$. The extension and subsequent use of the environment for typing variables can be explained as a kind of introduction and discharge of assumptions regarding type information. Note, however, that this relies on the use of additional rules in the specification, namely those for extracting type information for variables from the environment. This separation of the introduction and discharge of assumptions in environments and their subsequent use, though a minor point, obscures some of the logical properties of the specification.

Using extended natural semantics we can dispense with the environment and simply axiomatize a relation between programs and types, using open assumptions (possibly to be discharged at some point) to provide type information for free variables. We introduce the predicate symbol *infer* : $tm \rightarrow ty \rightarrow o$ and construct propositions of the form (*infer* e τ) where τ is a λ-term built up from the constants *int*, *bool*, *arrow* and *cross*. The proposition (*infer* e τ) states that the program encoded by e has the type encoded by τ and we write $\mathscr{I} \vdash$ (*infer* e τ) if it is provable using the rules of Fig. 3.

Most of these rules are similar to their natural semantics counterparts. For the constants of mini-ML, we assume some given map $\mathscr{C}$, mapping constants to types, e.g. $\mathscr{C}(<) =$ (*arrow int* (*arrow int bool*)). The first rule of Fig. 3 types the constants using this map. The next four rules specify the typing of the conditionals, pairs and projections. Rule *I*.6 is the typing rule for lambda abstraction and it exploits the rule for universal introduction to introduce an eigenvariable y representing the bound variable and implication introduction to introduce a hypothesis regarding the typing for y. No environment is necessary. Informally we may read the rule as follows. If under the assumption that the bound variable of a λ-abstraction has a type T_1, the body of that abstraction has type T_2, then the λ-abstraction has type (*arrow* T_1 T_2). The universal quantification of x enforces the restriction that the assumption about the type of the bound variable refers only to the occurrences of that bound variable, and no others. This is similar to typical uniqueness conditions placed on variables occurring in environments. Although this is in many ways similar to the environment approach, it avoids the need to access the types associated with

$$\frac{}{\mathit{infer}\ c\ \mathscr{C}(c)} \qquad (I.1)$$

$$\frac{\mathit{infer}\ E_1\ \mathit{bool} \qquad \mathit{infer}\ E_2\ T \qquad \mathit{infer}\ E_3\ T}{\mathit{infer}\ (\mathit{if}\ E_1\ E_2\ E_3)\ T} \qquad (I.2)$$

$$\frac{\mathit{infer}\ E_1\ T_1 \qquad \mathit{infer}\ E_2\ T_2}{\mathit{infer}\ (\mathit{pair}\ E_1\ E_2)\ (\mathit{cross}\ T_1\ T_2)} \qquad (I.3)$$

$$\frac{\mathit{infer}\ E\ (\mathit{cross}\ T_1\ T_2)}{\mathit{infer}\ (\mathit{fst}\ E)\ T_1} \qquad (I.4)$$

$$\frac{\mathit{infer}\ E\ (\mathit{cross}\ T_1\ T_2)}{\mathit{infer}\ (\mathit{snd}\ E)\ T_2} \qquad (I.5)$$

$$\frac{\forall x\ (\mathit{infer}\ x\ T_1\ \Rightarrow\ \mathit{infer}\ (E\ x)\ T_2)}{\mathit{infer}\ (\mathit{abs}\ E)\ (\mathit{arrow}\ T_1\ T_2)} \qquad (I.6)$$

$$\frac{\mathit{infer}\ E_1\ (\mathit{arrow}\ T_1\ T_2) \qquad \mathit{infer}\ E_2\ T_1}{\mathit{infer}\ (\mathit{app}\ E_1\ E_2)\ T_2} \qquad (I.7)$$

$$\frac{\mathit{infer}\ E_2\ T_2 \qquad \mathit{infer}\ (E_1\ E_2)\ T_1}{\mathit{infer}\ (\mathit{let}\ E_1\ E_2)\ T_1} \qquad (I.8)$$

$$\frac{\forall x\ (\mathit{infer}\ x\ T\ \Rightarrow\ \mathit{infer}\ (E\ x)\ T)}{\mathit{infer}\ (\mathit{fix}\ E)\ T} \qquad (I.9)$$

Fig. 3. Type inference system $\mathscr{I}$ for mini-ML.

variables in an environment and the need to make assumptions about the uniqueness of variables in the environment.

Rule $I.7$ is the rule for typing application. The term $(\mathit{arrow}\ T_1\ T_2)$ denotes the type for functions from the type (represented by) T_1 to the type (represented by) T_2. If we can show that E_1 has this type and E_2 has type T_1, then we can conclude that the application of E_1 to E_2, represented as $(\mathit{app}\ E_1\ E_2)$, has type T_2. Rule $I.9$ for fixed points uses the same technique as found in Rule $I.6$. The rule for *let* requires some explanation. Rather than introduce type schemas or polytypes, as done in the Damas-Milner type system, we use substitution to capture the notion of instantiating a term at different types. Rule $I.8$ checks that E_2 has some type, but that type is then ignored. We use β-reduction to substitute E_2 into the abstraction E_1, and then type the resulting expression. If we substitute E_2 into several different places in E_1, we infer a type for each of those instances; the types of different occurrences might be different, due to the contexts in E_1 in which these occurrences appear. Therefore, E_2 could be polymorphic in that its occurrences in E_1 might be at several different types. Note that we need to type E_2 explicitly just in the case that the abstraction is vacuous, i.e. the let-bound variable does not occur in E_1.

We do not need a rule for typing variables because any bound variable occurring in a term is replaced via β-reduction with either (i) a term explicitly typed via an assumption (*abs*, *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 do not consider polymorphic types and the associated rules for type quantification and instantiation. As discussed above, we type *let* expressions in a manner allowing us to avoid the use of explicit polymorphic types.

We can view this proof system as a declarative specification for type checking problems. Given a *closed* proposition of the form (*infer* e τ), finding a proof of this proposition asserts that the type of (the expression denoted by) e is τ. To perform type inference we need an algorithm to which we can provide e as input and which will produce as output τ such that $\mathscr{I} \vdash$ (*infer* e τ). For the simple type inference system here we can directly implement it in a logic programming language such as λProlog. By encoding each of the above inference rules as λProlog clauses and then posing the query ?– *infer* e T, for some closed e and variable T, a proof of some instance of this query can be found (when one exists). 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

$$?\!-\ \mathit{infer}\ \ (\mathit{abs}\ \lambda x.x)\ \ T$$

would have T instantiated to $T' \rightarrow T'$ for some variable T'. We have no explicit rule for quantifying over type variables.

In Hannan (1991) we demonstrate an equivalence between this presentation of typing, with its use of substitution in the *let* rule to achieve a kind of polymorphism, and the Damas-Milner type system (Damas and Milner, 1982) which uses polymorphic types and has rules for type instantiation and generalization. First we show that if we can derive (*infer* e τ) for closed e and τ, then there is an equivalent (modulo representation of terms and types) typing derivation in the Damas-Milner system. Second, if in the Damas-Milner system an expression e can by given some polymorphic type $\forall t_1(\cdots(\forall t_n(\tau)))$, then we can derive (*infer* e τ).

4.2 *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 (Mitchell and Harper, 1988). This relation concerns *monotypes* and *polytypes*. Monotypes are just simple types, i.e. those considered in the previous example, which we represent as terms of type *ty*. Polytypes are monotypes which may have a prefix of universal quantifiers,

$$\frac{}{subsume\ T\ T} \qquad \frac{\forall x{:}ty\ (subsume\ T_1\ (T_2\, x))}{subsume\ T_1\ (forall\ T_2)} \qquad \frac{subsume\ (T_1\, X)\ T_2}{subsume\ (forall\ T_1)\ T_2}$$

Fig. 4. Subsumes relation for polytypes.

such as $\forall t_1 \forall t_2 (t_1 \rightarrow t_2)$, in which the quantification ranges over monotypes. We introduce a new type *poly* and new constants *mono* : *ty*→*poly*, *forall* : (*ty*→*poly*)→*poly* for representing polytypes. All monotypes are polytypes and *mono* provides a way of coercing a monotype into a polytype. The constant *forall* is second-order and used to represent the ∀-quantification of polytypes. In the following discussion, the greek letters τ and σ represent monotypes and polytypes, respectively. To define the subsumes relation we need the following auxiliary definition.

Definition 4.1

τ is an instance of polytype $(forall\ \lambda t_1(\ldots(forall\ \lambda t_n(\tau'))\ldots))$ if there exists some substitution $\mathcal{S}$ 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 4.2 (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 $\forall t.t$ 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 specification of subsumes differently.

In our meta-language we can construct a simple proof system for the subsumes relation and it is given in Fig. 4. We use the binary predicate symbol *subsume* : *poly*→*poly*→*o*. The first clause states the obvious: any polytype subsumes itself. The second clause produces a canonical instance of (*forall* T_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 (*forall* T_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 monotype.

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 *subsume* as subset. The first clause states that every set (type) is a subset of (subsumes) itself. The second clause states that if a type is a subset of all members of a family of types, then it is a subset of the intersection of that family. The third clauses similarly states that if some member of a family is a subset of a given type, then the intersection of that family is a subset of that type.

4.3 Weak evaluation

We now present an evaluator for mini-ML, using a style very similar to the one for type inference. This evaluator reduces terms using the call-by-value strategy as described, for example, in Plotkin (1975). The natural semantics approach uses a judgment of the form $\Gamma \vdash e \Rightarrow v$, with the environment Γ providing the values for the free variables of e. We avoid the use of environments by specifying rules that explicitly substitute values for variables in programs. Higher-order syntax allows this to be done concisely.

We introduce a new predicate symbol $eval : tm \rightarrow tm \rightarrow o$. Propositions in this system are of the form $(eval\ e\ v)$ in which e and v are expressions in mini-ML and v represents the result of evaluating e. Proofs of these propositions are constructed from the proof system CBV given in Fig. 5. If a proposition $(eval\ e\ v)$ is provable in this system we write CBV $\vdash (eval\ e\ v)$. The first rule specifies that constants just evaluate to themselves. The next two rules treat the *if* expression in a natural way: Given an expression $(if\ e_1\ e_2\ e_3)$, e_1 must evaluate to *true* or *false* for a proof to be found. If $(eval\ e_1\ true)$ is provable then rule *E*2 applies; if $(eval\ e_1\ false)$ is provable then rule *E*3 applies; otherwise no proof of an *if* expression is possible. The next three rules handle the evaluation of pairs and projections. Rule *E*7 states that an abstraction evaluates to itself. In the rule for application, *E*8, we exploit Proposition 2.1 and the equality of terms up to $\beta\eta$-convertibility: given a term $(app\ s\ t)$, if for some f, $(eval\ s\ (abs\ f))$ is provable and for some v_2, $(eval\ t\ v_2)$ is provable, then the β-normal form of $(f\ v_2)$ represents the abstract term in which v_2 has been substituted for the (outermost) bound variable of $(abs\ f)$. In other words, the use of β-reduction correctly captures the notion of substitution used in function application. Similar comments apply to our rule for *let*, *E*9, in which the schema variable E_1 is a functional argument whose instance is applied to the term V_2, again making use of β-reduction to perform substitution. In the rule for recursion, *E*10, we unfold the definition of a *fix* expression. Recall that the *fix* constant is essentially the Y combinator and so this rule expresses the equation $(Y\,f) = f(Y\,f)$. The result of β-converting the expression $(f\ (fix\ f))$ substitutes the recursive function, namely $(fix\ f)$, within the body of the definition, given by f. In each of the rules manipulating bound variables static scoping is ensured 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.

To obtain a call-by-name evaluator we need only change the rules *E*8 and *E*9 to the following:

$$\frac{eval\ E_1\ (abs\ E) \qquad eval\ (E\ E_2)\ V}{eval\ (app\ E_1\ E_2)\ V}$$

$$\frac{eval\ (E_1\ E_2)\ V}{eval\ (let\ E_1\ E_2)\ V}.$$

No other change is necessary.

$$\frac{}{eval\ c\ c} \qquad (E1)$$

$$\frac{eval\ E_1\ true \qquad eval\ E_2\ V}{eval\ (if\ E_1\ E_2\ E_3)\ V} \qquad (E2)$$

$$\frac{eval\ E_1\ false \qquad eval\ E_3\ V}{eval\ (if\ E_1\ E_2\ E_3)\ V} \qquad (E3)$$

$$\frac{eval\ E_1\ V_1 \qquad eval\ E_2\ V_2}{eval\ (pair\ E_1\ E_2)\ (pair\ V_1\ V_2)} \qquad (E4)$$

$$\frac{eval\ E\ (pair\ V_1\ V_2)}{eval\ (fst\ E)\ V_1} \qquad (E5)$$

$$\frac{eval\ E\ (pair\ V_1\ V_2)}{eval\ (snd\ E)\ V_2} \qquad (E6)$$

$$eval\ (abs\ E)\ (abs\ E) \qquad (E7)$$

$$\frac{eval\ E_1\ (abs\ E) \qquad eval\ E_2\ V_2 \qquad eval\ (E\ V_2)\ V}{eval\ (app\ E_1\ E_2)\ V} \qquad (E8)$$

$$\frac{eval\ E_2\ V_2 \qquad eval\ (E\ V_2)\ V}{eval\ (let\ E\ E_2)\ V} \qquad (E9)$$

$$\frac{eval\ (E\ (fix\ E))\ V}{eval\ (fix\ E)\ V} \qquad (E10)$$

Fig. 5. Call-by-value evaluator CBV for mini-ML.

The set rules of Fig. 5 can be directly translated into a λProlog program and this program provides a simple interpreter for mini-ML. The rules are simple enough that the simple depth-first search strategy employed by a λProlog implementation is sufficient. Given some closed expression e, we can evaluate e by posing the query ?– *eval* e V, where V is a variable, resulting in an instantiation v for V exactly when *eval* e v is provable using these rules. Searching for a proof in a bottom-up fashion may involve situations in which several inference rules are applicable. For example, the proposition (*eval* (*if* e_1 e_2 e_3) V) matches the conclusion of two inference rules, and the use of at most one of these can eventually lead to a proof (assuming we cannot prove both (*eval* e_1 *true*) and (*eval* e_1 *false*)). The correct rule to apply in this case can only be determined after obtaining a proof of either $(eval\ e_1\ true)$ or $(eval\ e_1\ false)$.

4.4 Normal-form evaluation

The definition of evaluation given in the previous section prohibits the evaluation of terms inside of a bound variable, e.g. a λ-abstraction is already a value. This is a typical strategy for a programming language semantics in which a function's body is not evaluated until the function has been applied to the appropriate number of arguments. In particular, only closed programs are ever considered. If free variables do occur in terms they are identified with terms via a substitution or environment. However, other kinds of operations on programs may require a more flexible manipulation of programs.

Suppose now that we wish to define a more liberal evaluation strategy, one that allows evaluation under λ-abstractions. In particular, evaluation of λ-terms to $\beta\eta$-normal form requires such a strategy. The notion of descending into an abstraction to perform reductions can also be viewed as *mixed evaluation* since evaluation must be done not only on closed terms of mini-ML but also on terms containing abstracted variables. These are often treated as *symbolic* values. Thus, computations on real and symbolic values must be mixed together.

Two problems immediately arise when we consider evaluation inside the scope of a λ-abstraction. The first is that of correctly evaluating an expression containing an abstracted variable: our evaluator (from section 4.3) currently only treats expressions whose top-level symbol is a constant declared in the abstract syntax of mini-ML. A reasonable approach to extending evaluation to variables is allow bound variables to evaluate to themselves. The challenge here is to see how a proof system might support such a treatment of abstracted variables, particularly given that we cannot directly access the name of a bound variable in our abstract syntax. The second problem of evaluating within an abstraction is that generally the result of such rewriting will not be a proper value, that is, a value computable by our standard evaluator CBV. Consider, for example, the expression

$$(abs\ \lambda x(if\ \ x\ \ e_1\ \ e_2))$$

for some e_1 and e_2. Obviously, only with $v = (abs\ \ \lambda x(if\ \ x\ \ e_1\ \ e_2))$ can we prove

$$(eval\ \ (abs\ \ \lambda x(if\ \ x\ \ e_1\ \ e_2))\ \ v)$$

from CBV, even if we can prove $(eval\ e_1\ v_1)$ and $(eval\ e_2\ v_2)$, for some values v_1, v_2. Furthermore, for no value v' can we prove $(eval\ \ (if\ \ x\ \ e_1\ \ e_2)\ \ v')$ under the assumption that x a bound variable. Intuitively, we might want $v' = (if\ \ x\ \ v_1\ \ v_2)$. Thus, we must consider a more general notion of values than that defined by the rules of CBV. As this example suggests, an *if* expression should be interpreted in one of three different ways: Fig. 5 provides two ways and the third way relates it to another *if* expression in which its arguments may have been evaluated. This last observation reveals a cost in doing a more liberal rewriting of terms: evaluation may become more non-deterministic.

To obtain the new specification of evaluation for mini-ML we add new rules to those for the standard evaluator of section 4.3. As the discussed above, we should include the following inference figure in our new evaluator:

$$\frac{eval\ E_1\ E_1' \qquad eval\ E_2\ E_2' \qquad eval\ E_3\ E_3'}{eval\ \ (if\ E_1\ E_2\ E_3)\ \ (if\ E_1'\ E_2'\ E_3')}.$$

Thus, an *if*-expression can evaluate to another *if*-expression if their corresponding arguments are in the evaluation relation. Similar proof rules for some other constructs can likewise be defined. The case for λ-abstraction is different, however. We claim that the following inference rule specifies a natural evaluation strategy for object-level λ-abstractions:

$$\frac{\forall x\ (eval\ x\ x\ \Rightarrow\ (eval\ (E\ x)\ (E'\ x)))}{eval\ (abs\ E)\ (abs\ E')}.$$

This rule for the evaluation of (*abs E*) can be read operationally as follows: if under the assumption that x evaluates to itself, in which x is a new constant added to the signature for mini-ML terms, we can show that $(E\ x)$ evaluates to $(E'\ x)$, then we can conclude that (*abs E*) evaluates to (*abs E′*). This inference rule can be interpreted as replacing the abstracted variable of (*abs E*) with an eigenvariable that will name that bound variable. This variable, say x, is also assumed to evaluate to itself. A value is then sought for the expression $(E\ \ x)$. This value may contain free occurrences of x. The abstraction E' is then the result of abstracting all occurrences of x from this value. The condition that x does not occur in instances of particular terms is enforced via the conditions for universal introduction.

We might be tempted to write this rule as

$$\frac{\forall x\,\forall y(eval\ x\ y\ \Rightarrow\ (eval\ (E\ x)\ (E'\ y)))}{eval\ (abs\ E)\ (abs\ E')},$$

which seems to be a stronger statement about evaluation. Unfortunately, this rule is not adequate for our call-by-value semantics in which an evaluated term (in function applications) can be evaluated. Before a function is applied its argument is evaluated to a value and then this value (after being substituted for the bound variable of the function) may itself be evaluated. The above rule provides a way for evaluating x (to y) but no way to then evaluate y.

Fig. 6 contains the inference rules that are needed to extend the proof system for standard evaluation into a proof system for this new kind of evaluation. Let NF be the set of inference rules consisting of the ones from Fig. 6 and the ones for standard evaluation (Fig. 5). We refer to proofs constructed in this system as NF-proofs.

Since we have constructed this proof system by extending the one for standard evaluation, we have that for all e, v, CBV $\vdash$ $(eval\ e\ v)$ implies NF $\vdash$ $(eval\ e\ v)$. The converse is not true, however, and for a given e there may now be many e' such that NF $\vdash$ $(eval\ e\ e')$. Also, notice that NF-evaluation is reflexive, that is, for all terms $e \in$ mini-ML, NF $\vdash$ $(eval\ e\ e)$. In Hannan and Miller (1989) we further explore the relationship between our original weak evaluation system and this new one. An important property discussed there is the following result.

$$\frac{eval\ E_1\ E_1' \qquad eval\ E_2\ E_2' \qquad eval\ E_3\ E_3'}{eval\ \ (if\ E_1\ E_2\ E_3)\ \ (if\ E_1'\ E_2'\ E_3')} \tag{M.1}$$

$$\frac{\forall x\ (eval\ x\ x\ \Rightarrow\ (eval\ (E\ x)\ (E'\ x)))}{eval\ \ (abs\ E)\ \ (abs\ E')} \tag{M.2}$$

$$\frac{eval\ E_1\ E_1' \qquad eval\ E_2\ E_2'}{eval\ \ (app\ E_1\ E_2)\ \ (app\ E_1'\ E_2')} \tag{M.3}$$

$$\frac{eval\ E_2\ E_2' \qquad \forall x\ (eval\ x\ x\ \Rightarrow\ (eval\ (E\ x)\ (E'\ x)))}{eval\ \ (let\ E\ E_2)\ \ (let\ E'\ E_2')} \tag{M.4}$$

$$\frac{\forall x\ (eval\ x\ x\ \Rightarrow\ (eval\ (E\ x)\ (E'\ x)))}{eval\ \ (fix\ E)\ \ (fix\ E')} \tag{M.5}$$

Fig. 6. New inference rules for NF-evaluation.

Lemma 4.3
If NF ⊢ (*eval* (*abs f*) (*abs g*)) *and* NF ⊢ (*eval s t*), *then* NF ⊢ (*eval* (*f s*) (*g t*)).

The proof follows straightforwardly from results regarding hereditary Harrop formulas and uniform proofs Hannan and Miller (1991). See Hannan and Miller (1989) for an outline of the proof. Proving statements similar to this lemma in weaker meta-logics is typically a much more difficult task and often requires explicit proofs by induction.

Now let us consider an example of NF-evaluation. Let A be an abbreviation of the addition function given by the term

$$(fix\ \lambda f(abs\ \lambda x(abs\ \lambda y(if\ \ (app\ zerop\ x)\ \ y$$
$$(app\ s\ (app\ (app\ f\ \ (app\ pred\ \ x))\ \ y)))))).$$

Now suppose we try to show that there exists some v such that

CBV ⊢ (*eval* (*app A* (*app s* (*app s z*))) *v*).

It is not hard to see that the only possible value for v is

$$(abs\ \lambda y(if\ \ (app\ zerop\ (app\ s\ (app\ s\ z)))\ \ y$$
$$(app\ s\ (app\ (app\ A\ (app\ pred\ \ (app\ s\ (app\ s\ z))))\ \ y)))).$$

Now consider showing NF ⊢ (*eval* (*app A* (*app s* (*app s z*))) *v*) for some *v*. The additional rules of the NF proof system provide a means for further simplification of this expression. Using NF we can have the same value for *v* as obtained with CBV. However, another possible instance of *v* is (*abs* λ*y*(*app s* (*app s y*))).

As noted above, one cost of introducing the additional rules for NF-evaluation is an increase in the number of possible values for an expression. Thus controlling the application of these rules in a practical way is an important issue. We elide this point, choosing to focus only on the declarative aspects of NF-evaluation.

We can specify evaluation of applications in a slightly different manner, using a technique that corresponds closely to the use of environments in natural semantics. Instead of substituting (via β-reduction) the value of the argument for the bound variable as done in rule $E8$, we can introduce an assumption specifying that this variable evaluates to the given value. This idea is similar to the one used to specify type checking for λ-abstraction. We can construct a new specification, NF′, by replacing rule $E8$ of NF with

$$\frac{eval\ E_1\ (abs\ E) \qquad eval\ E_2\ V_2 \qquad \forall x\,(eval\ x\ V_2 \Rightarrow eval\ (E\,x)\ V)}{eval\ (app\ E_1\ E_2)\ V}.$$

Using this rule to evaluate an application proceeds as follows. We evaluate E_1 to an abstraction ($abs\ E$) and E_2 to a value V_2, just as in rule $E8$. Then we evaluate the body of the abstraction under the assumption that the bound variable x evaluates to V_2. The equivalence of NF and NF′ is straightforward to show. The advantage of using this new rule instead of $E8$ becomes evident when we consider the implementation of these rules in a language like λProlog. This new rule can be implemented in the language L_λ (Miller, 1991), which employs a simpler form of β-conversion and whose implementation requires only a decidable subclass of higher-order unification. The rule $E8$ cannot be specified directly in L_λ, but only a language like λProlog, with general β-conversion. We can introduce similar rules for *let* and *fix* expressions, such that the entire specification for evaluation can be given in L_λ. In general, full β-conversion (required by the implementation of specifications) can be systematically replaced by the simpler form of β-conversion found in L_λ.

Note that if we were to use this rule in place of $E8$ for the weak form of evaluation, a problem occurs. There may be some occurrences of x in the term $(E\,x)$ that we never attempt to evaluate (because they occur within the scope of a λ-abstraction). Hence we can never replace these occurrences with the evaluated term V_2. Furthermore, the resulting value we do get, containing occurrences of x, cannot match the variable V because any instance of V cannot contain any instances of x (due to the universal quantification of x).

4.5 Translation to a first-order syntax

All of the previous examples make only limited use of implication in the antecedents of rules. In particular, every instance of $A \Rightarrow B$ in the antecedent of a rule is such that both A and B are atomic formulas, and furthermore, such that A is an atomic formula expressing a property of an eigenvariable. This parallels the use of antecedents in the sequents of natural semantics where they consist of assumptions about variables. This also parallels the limited use of implication (instead of sequents) in the operational semantics of Burstall and Honsell (1991). We now consider an interesting use of implication in the antecedents of rules in which we use formulas of the form $(A \Rightarrow B) \Rightarrow C$. Such a construct has no parallel in natural semantics and offers a more sophisticated level of reasoning than is possible in natural semantics. Our example is the translation from a higher-order to a first-order syntax. While such a translation can be defined without using such nested implications we believe that the specification given below illustrates the elegance of this approach.

Manipulating programs in a higher-order abstract syntax has its attractive features, but eventually we might want to translate programs into a lower-level syntax, often for reasons of efficiency. We can think of translating along two different lines: (1) from a higher-order to a first-order version of the same syntax (language) or (2) from one language to another, with possibly both represented in a higher-order syntax. We consider the former problem in this section. We introduce a simple first-order syntax for mini-ML, and then describe a translation between the two syntaxes.

We choose a first-order abstract syntax using de Bruijn indices to represent variables. We introduce new types tm^o and nat and the following set of constructors:

0	:	nat	fst^o	:	$tm^o \rightarrow tm^o$
$\hat{}$	:	$nat \rightarrow nat$	snd^o	:	$tm^o \rightarrow tm^o$
c^o	:	tm^o	app^o	:	$tm^o \rightarrow tm^o \rightarrow tm^o$
var	:	$nat \rightarrow tm^o$	abs^o	:	$tm^o \rightarrow tm^o$
if^o	:	$tm^o \rightarrow tm^o \rightarrow tm^o \rightarrow tm^o$	let^o	:	$tm^o \rightarrow tm^o \rightarrow tm^o$
$pair^o$	:	$tm^o \rightarrow tm^o \rightarrow tm^o$	fix^o	:	$tm^o \rightarrow tm^o$

We introduce the constants 0 and $\hat{}$ to construct natural numbers. For every primitive constant $c : tm$ of the higher-order syntax, we introduce a constant $c^o : tm^o$. We introduce the constant *var* to construct de Bruijn indices from natural numbers. As an example of a program in this syntax consider the following representation of the addition function:

$$\begin{array}{l}
(let^o\ \ (app^o\ (app^o\ (var\ (\hat{}0))\ \ (app^o\ s^o\ (app^o\ s^o\ z^o)))\ \ (app^o\ s^o\ (app^o\ s^o\ z^o)))\\
\qquad (fix^o\ (abs^o\ (abs^o\ (if^o\ \ (app^o\ zerop^o\ (var\ (\hat{}(\hat{}0))))\ (var\ (\hat{}0))\\
\qquad\qquad\qquad\qquad\qquad\quad (s^o\ (app^o\ \ (app^o\ \ (var\ (\hat{}(\hat{}(\hat{}0))))\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (app^o\ pred^o\ (var\ (\hat{}(\hat{}0)))))\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (var\ (\hat{}0)))))))))).
\end{array}$$

Compare this term with the one from section 2.

For the translation between the higher-order and first-order syntaxes we introduce the predicate symbol $trans : nat \rightarrow tm \rightarrow tm^o \rightarrow o$. The first argument to this predicate represents the *depth*, given as a natural number, at which the current expression occurs. The depth represents the number of abstractions encountered along the path from the root of the term to the current subterm. Abstractions occur in λ-abstractions (*abs* or abs^o), let-expressions (first argument to *let* or let^o) and fix-expressions (*fix* or fix^o). For example, in the term above, the subterm $(app^o\ s^o\ (app^o\ s^o\ z^o))$ occurs at depth 1 (inside the let^o abstraction) and the subterm $(app^o\ zerop^o\ (var\ (\hat{}(\hat{}0))))$ occurs at depth 3 (inside the fix^o and two abs^o abstractions). The entire term occurs at depth 0. We use this depth to determine the appropriate de Bruijn indices for representing variables in the first-order syntax. Recall that in de Bruijn syntax a variable occurrence is represented by a natural number which is the number

$$\frac{}{trans\ N\ c\ c^o} \tag{T.1}$$

$$\frac{trans\ N\ E_1\ F_1 \qquad trans\ N\ E_2\ F_2 \qquad trans\ N\ E_3\ F_3}{trans\ \ N\ \ (if\ E_1\ E_2\ E_3)\ \ (if^o\ F_1\ F_2\ F_3)} \tag{T.2}$$

$$\frac{trans\ N\ E_1\ F_1 \qquad trans\ N\ E_2\ F_2}{trans\ \ N\ \ (pair\ E_1\ E_2)\ \ (pair^o\ F_1\ F_2)} \tag{T.3}$$

$$\frac{trans\ N\ E\ F}{trans\ \ N\ \ (fst\ E)\ \ (fst^o\ F)} \tag{T.4}$$

$$\frac{trans\ N\ E\ F}{trans\ \ N\ \ (snd\ E)\ \ (snd^o\ F)} \tag{T.5}$$

$$\frac{trans\ N\ E_1\ F_1 \qquad trans\ N\ E_2\ F_2}{trans\ \ N\ \ (app\ E_1\ E_2)\ \ (app^o\ F_1\ F_2)} \tag{T.6}$$

$$\frac{\begin{array}{r}\forall x\,(\forall N_1 \forall N_2 (minus\ N_1\ N\ N_2\ \Rightarrow\ trans\ N_1\ x\ (var\ N_2))\ \Rightarrow \\ trans\ \ ({}^\wedge N)\ \ (E\,x)\ \ F)\end{array}}{trans\ \ N\ \ (abs\ E)\ \ (abs^o\ F)} \tag{T.7}$$

$$\frac{\begin{array}{l}trans\ N\ E_1\ F_1 \\ \forall x\,(\forall N_1 \forall N_2 (minus\ N_1\ N\ N_2\ \Rightarrow\ trans\ N_1\ x\ (var\ N_2))\ \Rightarrow \\ \hfill trans\ \ ({}^\wedge N)\ \ (E\,x)\ \ F)\end{array}}{trans\ \ N\ \ (let\ E\ E_1)\ \ (let^o\ F\ F_1)} \tag{T.8}$$

$$\frac{\begin{array}{r}\forall x\,(\forall N_1 \forall N_2 (minus\ N_1\ N\ N_2\ \Rightarrow\ trans\ N_1\ x\ (var\ N_2))\ \Rightarrow \\ trans\ \ ({}^\wedge N)\ \ (E\,x)\ \ F)\end{array}}{trans\ \ N\ \ (fix\ E)\ \ (fix^o\ F)} \tag{T.9}$$

Fig. 7. Translation from higher-order to first-order syntax.

of abstractions encountered along the path from this occurrence to the defining λ-abstraction for this variable.

The complete specification of *trans* is given in Fig. 7. We assume that subtraction has been axiomatized using the predicate *minus* : $nat \to nat \to nat \to o$ as follows:

$$\frac{}{minus\ N\ 0\ N} \qquad\qquad \frac{minus\ N_1\ N_2\ N_3}{minus\ ({}^\wedge N_1)\ ({}^\wedge N_2)\ N_3}$$

The proposition $(minus\ n_1\ n_2\ n_3)$ is provable iff n_1, n_2, n_3 are representations of natural numbers n1, n2, and n3, respectively, n1 $\geqslant$ n2, and n1 - n2 = n3.

The rules for translating λ-abstractions, let-expressions and fix-expressions require some explanation, as these rules contain some uses of our meta-logic that we have not encountered previously. First, note that we have no explicit rules for translating variables in the two syntaxes. This definition of *trans* works only for closed expressions. We still, however, must handle the translation of bound variables. We explain the rule for translating λ-abstractions, with similar explanations following for let- and fix-expressions.

The rule *T*.7 has a similar structure to the rule *M*.2: to descend through a λ-abstraction we use a $\forall$-judgment to introduce the quantified variable x which represents the bound variable of the abstraction. We also introduce an assumption (or hypothesis) about x. Consider an instance of rule *T*.7 (in which the implicitly universally quantified variables N, E, F have been instantiated by some closed terms n, e, f:

$$\frac{\forall x\,(\forall N_1 \forall N_2 (\mathit{minus}\ N_1\ n\ N_2\ \Rightarrow\ \mathit{trans}\ N_1\ x\ (\mathit{var}\ N_2))\ \Rightarrow\ \mathit{trans}\ (\hat{\ }n)\ (e\,x)\ f)}{\mathit{trans}\ n\ (\mathit{abs}\ e)\ (\mathit{abs}^o\ f)}$$

To construct a proof of the premise to this rule we first use the ($\forall I$) rule to replace the variable x with an eigenvariable y (requiring the usual freeness condition as stipulated by the rule for $\forall$-Introduction) and then we construct a proof of $(\mathit{trans}\ (\hat{\ }n)\ (e\,y)\ f)$ using the hypothesis

$$\forall N_1 \forall N_2 (\mathit{minus}\ N_1\ n\ N_2\ \Rightarrow\ \mathit{trans}\ N_1\ y\ (\mathit{var}\ N_2)).$$

If we write this hypothesis in the more graphical notation of inference rules (the implication occurs positively in this rule), we get

$$\frac{\mathit{minus}\ N_1\ n\ N_2}{\mathit{trans}\ N_1\ y\ (\mathit{var}\ N_2)},$$

in which N_1 and N_2 are implicitly universally quantified. We can view this hypothesis as a rule for translating the eigenvariable y (representing the bound variable of the λ-abstraction): *y translates to de Bruijn index* ($\mathit{var}\ N_2$) *such that N_2 is equal to the difference between the current depth N_1 and the depth n at which the λ-abstraction occurs*. This computation of indices is exactly what is expected for de Bruijn notation of λ-terms. Instances of this hypothesis can be obtained and used by first applying the ($\forall E$) rule twice, instantiating N_1 and N_2 with appropriate terms. For example, to prove $(\mathit{trans}\ n_1\ y\ n_2)$ for some particular terms n_1, n_2 we could start with the following deduction:

$$\cfrac{\cfrac{\forall N_1 \forall N_2 (\mathit{minus}\ N_1\ n\ N_2\ \Rightarrow\ \mathit{trans}\ N_1\ y\ (\mathit{var}\ N_2))}{\forall N_2 (\mathit{minus}\ n_1\ n\ N_2\ \Rightarrow\ \mathit{trans}\ n_1\ y\ (\mathit{var}\ N_2))}}{\mathit{minus}\ n_1\ n\ n_2\ \Rightarrow\ \mathit{trans}\ n_1\ y\ (\mathit{var}\ n_2)}$$

From this deduction and a deduction for $(\mathit{minus}\ n_1\ n\ n_2)$, we can prove (using implication elimination) $(\mathit{trans}\ n_1\ y\ (\mathit{var}\ n_2))$. Note that for each λ-abstraction encountered during the translation of an expression, a new assumption is made for translating the bound variable of the abstraction.

As an example of this specification consider the following formula which is provable using the rules of Fig. 7 (for readability we use integers 0, 1, 2, 3 instead of their actual representation):

$$\begin{aligned}(\mathit{trans}\ \ 0\ \ &(\mathit{abs}\ \lambda x(\mathit{abs}\ \lambda y(\mathit{app}\ x\ (\mathit{abs}\ \lambda z\ (\mathit{app}\ (\mathit{app}\ x\ z)\ y)))))\\ &(\mathit{abs}^o\ (\mathit{abs}^o\ (\mathit{app}^o\ (\mathit{var}\ 2)\ (\mathit{abs}^o\ (\mathit{app}^o\ \ (\mathit{app}^o\ (\mathit{var}\ 3)\ (\mathit{var}\ 1))\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{var}\ 2)))))).\end{aligned}$$

This formula relates two representations for the λ-term $\lambda x \lambda y(x\, \lambda z(x\, z\, y))$. Note how the two occurrences of the bound variable x are represented as $(\mathit{var}\ 2)$ and $(\mathit{var}\ 3)$ in the first-order representation. To translate these two occurrences of x the proof uses two instances of the hypothesis

$$\forall N_1 \forall N_2(\mathit{minus}\ N_1\ n\ N_2 \ \Rightarrow\ \mathit{trans}\ N_1\ x\ (\mathit{var}\ N_2))$$

modulo renaming of the bound variable x with an eigenvariable.

5 Conclusion

We have extended natural semantics to include a richer notion of syntax and appropriate mechanisms for describing semantics at a reasonably high level. An important aspect of this work is the use of a higher-order abstract syntax for representing functional programs. By uniformly encoding various binding and scoping constructs of a language via λ-abstraction in an abstract syntax and providing an equality theory over terms in the syntax that includes α, β and η-conversion, we can specify syntactically many of the 'non context-free' aspects of a programming language such as scoping of identifiers. Not all such aspects can be enforced by our abstract syntax, however. For instance, the linearity restriction imposed on variables occurring in pattern expressions in Standard ML cannot naturally be expressed in this syntax without explicitly axiomatizing the notion of linearity. But this is no worse than traditional first-order representations. Similarly, a higher-order abstract syntax can be unsuitable for language definitions in which programs are not generally equivalent up to a renaming of variables. For example, dynamically scoped variables, available in Lisp, could not easily be described using this syntax. The binding of a variable in this language is determined dynamically by considering the most recently (temporally) defined variable of the same name. Higher-order syntax seems best suited to statically scoped languages.

The ability to use hypothetical reasoning (via implications) provides a way of eliminating explicit environments in semantic specifications. While the two formulations (sequents with explicit open assumptions and traditional natural deduction with implicit open assumptions) are equivalent, reasoning about the latter seems easier as the relationship between assumption and conclusion is directly represented in the logic (as an implication), while with natural semantics and sequents, the relationship is encoded and relies on the proper use of environments by other rules. This difference becomes significant when dealing with proofs as objects, such as in the LF logical framework (Harper *et al.*, to appear) and the Elf programming language (Pfenning, 1991). In these settings a proof of $A \Rightarrow B$ is given as a function which, when applied to a term representing a proof of A, produces a proof of

B. Reasoning about operational semantics by manipulating proofs in this setting provides a means for mechanically verifying properties of the semantics. A general treatment of this subject can be found in Michaylov and Pfenning (1992). A specific example considering compiler verification can be found in Hannan and Pfenning (1992).

Acknowledgments

I am grateful to Dale Miller for his detailed comments and suggestions for improving this paper. I wish to thank the members of the TOPPS group at the University of Copenhagen for providing a stimulating environment during the preparation of this paper.

References

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. and Pascual, V. 1987. CENTAUR: the system. Technical Report 777, INRIA, (December).

Bruijn, N. 1980. A survey of the project AUTOMATH. In Hindley, R. and Seldin, J., editors, *To H. B. Curry: Essays on Combinatory Logic*, Academic Press, 589–606.

Burstall, R. and Honsell, F. 1991. A natural deduction treatment of operational semantics. In Huet, G. and Plotkin, G., editors, *Logical Frameworks*, Cambridge University Press, 89–111.

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

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

Elliott, C. and Pfenning, F. 1989. eLP, a Common Lisp Implementation of λProlog.

Felty, A. and Miller, D. 1990. Encoding a dependent-type λ-calculus in a logic programming language. In Stickel, M. E., editor, *Proceedings of the Tenth International Conference on Automated Deduction*, Springer-Verlag LNCS Vol. 449, 221–235.

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

Hannan, J. and Miller, D. 1988. Enriching a meta-language with higher-order features. Technical Report MS-CIS-88-45, University of Pennsylvania (June).

Hannan, J. and Miller, D. 1989. Deriving mixed evaluation from standard evaluation for a simple functional language. In van de Snepscheut, J. L. A., editor, *Mathematics of Program Construction*, Springer-Verlag LNCS Vol. 375, 239–255.

Hannan, J. and Pfenning, F. 1992. Compiler verification in LF. In Scedrov, A., editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, IEEE Press, 407–418.

Hannan, J. 1991. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, (Available as MS-CIS-91-09.)

Harper, R., Honsell, F. and Plotkin, G. To appear. A framework for defining logics. *Journal of the ACM*. (A preliminary version appeared in *Symposium on Logic in Computer Science*, June 1987, 194–204.)

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

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

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

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

Michaylov, S. and Pfenning, F. 1992. Natural semantics and some of its meta-theory in Elf. In Hallnäs, L., editor, *Extensions of Logic Programming*, Springer-Verlag LNCS 596, 299–344. (A preliminary version is available as Technical Report MPI–I–91–211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.)

Miller, D. and Nadathur, G. 1987. A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Fourth Symposium on Logic Programming*, IEEE Press, 379–388.

Miller, D., Nadathur, G., Pfenning, F. and Scedrov, A. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, **51**:125–157.

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

Milner, R., Tofte, M. and Harper, R., 1991. *The Definition of Standard ML*. MIT Press.

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

Nadathur, G. and Miller, D. 1988. An overview of λProlog. In Bowen, K. and Kowalski, R., editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 810–827.

Paulson, L. 1989. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, **5**:363–397.

Pfenning, F. and Elliott, C. 1988. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 199–208. SIGPLAN Notices, Vol. 23, Number 7.

Pfenning, F. 1989. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, IEEE Press, 313–322.

Pfenning, F. 1991. Logic programming in the LF logical framework. In Huet, G. and Plotkin, G., editors, *Logical Frameworks*, Cambridge University Press, 149–181.

Plotkin, G. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159.

Plotkin, G. 1981. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark (September).

Prawitz, D., 1965. *Natural Deduction*. Almqvist & Wiksell, Uppsala.

Scott, D. 1980. Relating theories of the λ-calculus. In Hindley, R. and Seldin, J., editors, *To H. B. Curry: Essays on Combinatory Logic*, Academic Press, 403–450.