From Operational Semantics to Abstract Machines

John Hannan
Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen East, Denmark.
hannan@diku.dk

Dale Miller
Computer and Information Science, University of Pennsylvania,
Philadelphia, PA 19104-6389, USA.
dale@cis.upenn.edu

Received 2 February 1993

We consider the problem of mechanically constructing abstract machines from operational semantics, producing intermediate-level specifications of evaluators guaranteed to be correct with respect to the operational semantics. We construct these machines by repeatedly applying correctness-preserving transformations to operational semantics until the resulting specifications have the form of abstract machines. Though not automatable in general, this approach to constructing machine implementations can be mechanized, providing machine-verified correctness proofs. As examples we present the transformation of specifications for both call-by-name and call-by-value evaluation of the untyped λ-calculus into abstract machines that implement such evaluation strategies. We also present extensions to the call-by-value machine for a language containing constructs for recursion, conditionals, concrete data types, and built-in functions. In all cases, the correctness of the derived abstract machines follows from the (generally transparent) correctness of the initial operational semantic specification and the correctness of the transformations applied.

1. Introduction

The correctness of programming language implementations is an important issue confronting language designers and implementors. Traditionally, such implementations are first built "by hand" and only then proved correct. Unfortunately, the resulting implementations may have little relationship to the language's semantic specifications and so correctness may be difficult to show. For realistic languages, such correctness proofs become unwieldy because of the overwhelming amount of implementation-level information that must be correlated. In an alternative approach, a language implementation is constructed *from* the semantic specification in such a way that the resulting implementation is guaranteed to satisfy correctness requirements: no independent proof being necessary. This approach may provide a mechanical, if not automatic, process allowing the correctness to be machine checked. This latter point is important when considering the sizes of correctness proofs for realistic languages. Of course the choice of semantic specification and target language greatly influences the practicality of this approach.

We use operational semantics to specify the meaning of programs and abstract machines to provide an intermediate representation of the language's implementation. Operational semantics can be presented as inference rules or, equivalently, as formulas in a weak meta-logic permitting quantification at first-order and second-order types. Such semantic specifications provide high-level and clear descriptions of programming languages, often handling numerous syntactic details in a declarative and simple fashion. Abstract machines can be presented as rewrite rules describing single-step operations on the state of a computation. Such specifications provide an intermediate level of representation for many practical implementations of programming languages. The construction of abstract machines has traditionally been performed by hand, with correctness proofs following later. We demonstrate how to take a specification for an evaluator in the high-level, flexible style of operational semantics and derive, through formally justifiable steps, an abstract machine that implements that evaluator. While providing a direct proof of the correctness of derived abstract machines, these derivations also provide guidance for extending such machines and illustrate relationships among various abstract machines.

Because operational semantics specifications can have rich structure, completely automatic methods for constructing abstract machines from them seem feasible only for restricted classes of operational semantics. While identifying these classes and the corresponding procedures for constructing machines is of interest, we focus only on a general strategy for constructing machines and provide two examples of this task. Applying our approach to wider classes of languages and machines than considered here (for example, imperative languages and machines using a continuation-passing style) requires finding new transformations that will most certainly be dependent upon the particular structure of the semantic specification and machine for each case. The examples presented, however, demonstrate the kind of reasoning required for simple functional languages and environment-based abstract machines, and suggest the feasibility of similar transformations for other languages and architectures.

The basic goal of the derivations described in this paper differs slightly from the principal goal of most program translations. Typical translations involve rewriting programs in a source language into a different, target language. Compilation is such a process. Our specifications of both operational semantics and abstract machines sit inside a single meta-logic. Most of our effort in transforming specifications attempts to make various aspects of computation that are *implicit* in the way one specification relies on this meta-logic *explicit* within another specification. In this way, transformed specifications rely less on the meta-logic. For example, one set of transformations changes a specification that uses second-order quantification to a specification that only uses first-order quantification. By continuing this kind of transformation, it is possible to reduce the specification's reliance on the meta-logic to a point where theorem proving using that specification can be done by a simple rewriting of computation states that is modeled well using abstract machines.

This paper is organized as follows. In Section 2 we give a general description of the classes of operational semantics and abstract machines that we consider in later sections.

In Section 3 we present the meta-logic in which we specify operational semantics. This meta-logic provides various high-level features for simple manipulation of bound variables and contexts. In Section 4 we present two high-level and declarative specifications of evaluators which use second-order quantification, and then we transform them into specifications using only first-order quantification. Included in this section is a treatment of the translation from terms represented in a simply typed λ-calculus (with second-order constants) to terms in a language using only first-order constants. In Section 5 we take the first-order specifications developed in Section 4 and apply appropriate transformations to them until we produce systems describing abstract machines. The resulting proof systems are equivalent to the Krivine machine [Cur90] (for the call-by-name case) and a variant of the SECD machine [Lan64] (for the call-by-value case). In Section 6 we describe how the derivations of Sections 4 and 5 can be modified or extended to yield machines for richer languages that include recursion, a conditional and other new language features. Finally, in Section 7 we discuss relations to other work, and in Section 8 we summarize our results.

2. Operational Semantics and Abstract Machines

A recent trend in programming language design has been the use of operational semantics to define the semantics of a programming language. The particular style of operational semantics we consider here was inspired by Martin-Löf [Mar84] and uses sets of inference rules as specifications. Various instances of this approach have been called Proof-Theoretic Semantics, Natural Semantics and Relational Semantics [Han90, Kah87, MTH90]. This style of semantics has proved well suited to specifying evaluation strategies for realistic programming languages. Direct implementations of operational semantics require several general, symbolic techniques that are generally found in theorem provers or logic programming languages.

Abstract machines have been effectively used as intermediate and low-level architectures suitable for supporting serious implementations of a wide variety of programming languages, including imperative, functional, and logic programming languages. Abstract machines are distinguished from operational semantics by having simple and direct algorithmic implementations that can employ efficient data structures. On the other hand, abstract machines are distinguished from lower-level, machine-code implementations because typically the former uses pattern matching for destructing data while the latter explicitly addresses the notions of structure sharing, storage allocation, and register allocation.

One goal of this paper is to connect these two methods of specifying the evaluation of programming languages and, in doing so, to provide means of producing abstract machines from operational semantics. Below we provide an overview of each of these paradigms.

2.1. Operational Semantics

The phrase "structural operational semantics" is attributed to Plotkin, based on his seminal paper "*A Structural Approach to Operational Semantics*" [Plo81]. Although that term has come to describe a few different styles of specifications, in that paper it is used to describe evaluation in terms of a one-step reduction relation and his inference rules axiomatize this relation. Martin-Löf uses an operational semantics for describing type theory [Mar85]. While he does not present the definition as a set of inference rules, his notion of evaluation is both compositional and syntax directed. Milner is perhaps the first to use inference rules to axiomatize evaluation to canonical form in his descriptions of the dynamic semantics of Standard ML, including the latest specification of the full language [MTH90].

The work presented here was initially inspired by the work in *Natural Semantics*, as described in [Kah87]. That work uses a first-order meta-logic with inference rules presented using sequents. We extend this work by using a higher-order (and higher-level) meta-logic employing less explicit, natural deduction-style inference rules. As a reference point we present a simple example of natural semantics, namely type inference for a simple functional language containing function abstraction and application. We begin by defining a first-order abstract syntax for the language. We introduce three constants var: $string \rightarrow term$, lam: $(string \times term) \rightarrow term$, and com: $(term \times term) \rightarrow term$ for constructing variables, λ-abstractions, and applications (combinations), respectively. For example, we represent the familiar combinators $\boldsymbol{I}$, $\boldsymbol{K}$ and $\boldsymbol{S}$ as follows:

Concrete Syntax	*Abstract Syntax*
$\boldsymbol{\lambda x.x}$	lam("x", var("x"))
$\boldsymbol{\lambda x \lambda y.x}$	lam("x", lam("y", var("x")))
$\boldsymbol{\lambda x \lambda y \lambda z.xz(yz)}$	lam("x", lam("y", lam("z", com(com(var("x"),var("z")), com(var("y"),var("z"))))))

To specify type inference we must also represent types as objects and we can do this by introducing a new type ty and the constants tvar: $string \rightarrow ty$ and arrow: $(ty \times ty) \rightarrow ty$. The relationship between a program and its type is denoted by the predicate of the meta-logic hastype of meta-logic type $(((string \times ty)\ list) \times term \times ty) \rightarrow o$. (Following Church [Chu40], we use o to denote the type of meta-logic propositions). The first argument of hastype is a context binding variable names to types. For example, the proposition

hastype(nil, lam("i", var("i")), arrow(tvar("a"), tvar("a")))

relates the combinator $\boldsymbol{I}$ to its type. (List constructors are nil and the infix symbol ::.) We axiomatize such propositions via the set of inference rules in Figure 1. The member predicate is assumed to be defined such that if member$((X,T),\ \Gamma)$ is provable then (X,T) is the leftmost pair in Γ in which the left component is X. If the proposition

$$\frac{\mathsf{member}((X,T),\quad \Gamma)}{\mathsf{hastype}(\Gamma,\quad \mathsf{var}(X),\quad T)} \qquad \frac{\mathsf{hastype}((X,S)::\Gamma,\quad M,\quad T)}{\mathsf{hastype}(\Gamma,\quad \mathsf{lam}(X,M),\quad \mathsf{arrow}(S,T))}$$

$$\frac{\mathsf{hastype}(\Gamma,\quad M,\quad \mathsf{arrow}(S,T)) \qquad \mathsf{hastype}(\Gamma,\quad N,\quad S)}{\mathsf{hastype}(\Gamma,\quad \mathsf{com}(M,N),\quad T)}$$

Fig. 1. Axiomization of type inference

$\mathsf{hastype}(\gamma,\ t,\ \tau)$ is provable using this set of rules then the term (encoded by) t has type (encoded by) τ, given the assumptions in γ for the types of free object-variables of t.

In this setting and in others, the phrase "operational semantics" can be a misnomer, because such semantic specifications may lack a clear operational or algorithmic implementation. The term *deductive semantics* is more accurate, as the meanings of semantic judgments are given by deductions; the task of theorem proving or proof construction is left unspecified. One focus of our paper is the systematic restructuring of specifications so that rather simple algorithms can provide complete theorem provers. Because proofs in the meta-logic of semantic judgments describe the course of a computation, they will play a central role in justifying each restructuring of operational semantics.

2.2. Abstract Machines

Many abstract machines have much in common, and we define here the formal notion of *Abstract Evaluation System* (AES) that captures and abstracts some of this commonality. We assume some familiarity with term rewriting, its terminology and the notion of computation in a rewriting system [HO80]. Recall that a term rewriting system is a pair (Σ, R) such that Σ is a signature and R is a set of directed equations $\{l_i \Rightarrow r_i\}_{i\in I}$ with $l_i, r_i \in T_\Sigma(X)$ and $\mathcal{V}(r_i) \subseteq \mathcal{V}(l_i)$. Here, $T_\Sigma(X)$ denotes the set of first-order terms with constants from the signature Σ and free variables from X, and $\mathcal{V}(t)$ denotes the set of free variables occurring in t. We restrict our attention to first-order systems, i.e., Σ is a first-order signature, though this is not essential.

Definition 1. An *Abstract Evaluation System* is a quadruple (Σ, R, ρ, S) such that the pair $(\Sigma, R \cup \{\rho\})$ is a term rewriting system, ρ is not a member of R, and $S \subseteq R$.

Evaluation in an AES is a sequence of rewriting steps with the following restricted structure. The first rewrite rule must be an instance of the ρ rule. This rule can be understood as "loading" the machine to an initial state given an input expression. The last rewrite step must be an instance of a rule in S: these rules denote the successful termination of the machine and can be understood as "unloading" the machine and producing the answer or final value. All other rules are from R. We also make the following significant restriction to the general notion of term rewriting: all rewriting rules must be applied to a term at its root. This restriction significantly simplifies the computational complexity of applying rewrite rules during evaluation in an AES. A term $t \in T_\Sigma(\emptyset)$ *evaluates* to the term s (with respect to the AES (Σ, R, ρ, S)) if there is a series of rewriting rules satisfying the restrictions above that rewrites t into s.

M	$\Rightarrow$	$\langle nil,\ M,\ nil\rangle$
$\langle E,\ \lambda M,\ X :: S\rangle$	$\Rightarrow$	$\langle X :: E,\ M,\ S\rangle$
$\langle E,\ M \,\hat{}\, N,\ S\rangle$	$\Rightarrow$	$\langle E,\ M,\ \{E, N\} :: S\rangle$
$\langle \{E', M\} :: E,\ 0,\ S\rangle$	$\Rightarrow$	$\langle E',\ M,\ S\rangle$
$\langle X :: E,\ n+1,\ S\rangle$	$\Rightarrow$	$\langle E,\ n,\ S\rangle$
$\langle E,\ \lambda M,\ nil\rangle$	$\Rightarrow$	$\{E, \lambda M\}$

M	$\Rightarrow$	$\langle nil,\ nil,\ M :: nil,\ nil\rangle$
$\langle S,\ E,\ \lambda M :: C,\ D\rangle$	$\Rightarrow$	$\langle \{E, \lambda M\} :: S,\ E,\ C,\ D\rangle$
$\langle S,\ E,\ (M \,\hat{}\, N) :: C,\ D\rangle$	$\Rightarrow$	$\langle S,\ E,\ M :: N :: ap :: C,\ D\rangle$
$\langle S,\ E,\ n :: C,\ D\rangle$	$\Rightarrow$	$\langle nth(n, E) :: S,\ E,\ C,\ D\rangle$
$\langle X :: \{E', \lambda M\} :: S,\ E,\ ap :: C,\ D\rangle$	$\Rightarrow$	$\langle nil,\ X :: E',\ M :: nil,\ (S, E, C) :: D\rangle$
$\langle X :: S,\ E,\ nil,\ (S', E', C') :: D\rangle$	$\Rightarrow$	$\langle X :: S',\ E',\ C',\ D\rangle$
$\langle X :: S,\ E,\ nil,\ nil\rangle$	$\Rightarrow$	X

Fig. 2. The Krivine machine (top) and SECD machine (bottom)

In certain situations, it might be sensible to have a more restricted definition of AES. In particular, allowing only linear rewrite rules (rules whose left-hand sides have no repeated variables) simplifies an implementation of an AES by eliminating the need for a costly runtime equality check (to be certain that the duplicated variables are instantiated to the same term). Similarly, requiring an AES to be deterministic (that is, no two rewrite rules can be used on the same term) is also sensible, especially in the context of modeling evaluation of functional programs. We note that all the examples presented in this paper do, in fact, satisfy both of these additional restrictions.

The SECD machine [Lan64] and Krivine machine [Cur90] are both AESs and variants of these are given in Figure 2. The syntax for λ-terms uses de Bruijn notation with $\hat{}$ (infix) and λ as the constructors for application and abstraction, respectively, and $\{E, M\}$ denotes the closure of term M with environment E. The first rule given for each machine is the "load" rule or ρ of their AES description. The last rule given for each is the "unload" rule. (In each of these cases, the set S is a singleton.) The remaining rules are state transformation rules, each one moving the machine through a computation step.

A state in the Krivine machine is a triple $\langle E, M, S\rangle$ in which E is an environment, M is a single term to be evaluated and S is a stack of arguments. A state in the SECD machine is a quadruple $\langle S, E, C, D\rangle$ in which S is a stack of computed values, E is an environment (here just a list of terms), C is a list of commands (terms to be evaluated) and D is a dump or saved state. The expression $nth(n, E)$, used to access variables in an environment, is treated as a function that returns the $n + 1^{st}$ element of the list E.

Although Landin's original description of the SECD machine used variables names, our use of de Bruijn numerals does not change the essential mechanism of that machine.

3. The Meta-Language

All specifications of evaluation given in this paper, whether high-level or low-level, can be given as formulas or as inference rules within a simple meta-logic.

3.1. Types, Terms, and 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* is 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: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. The *order* of a type is the measure of how deeply function types are nested to the left: primitive types are of order 0 and the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$, in which τ_0 is primitive and $n > 0$, is one greater than the maximum order of the types $\tau_1, \ldots, \tau_n$.

The logical constants $\wedge$ (conjunction) and $\supset$ (implication) have type $o \rightarrow o \rightarrow o$ and the logical constant $\forall_\tau$ (universal quantification of type τ) has type $(\tau \rightarrow o) \rightarrow o$, for every type τ that does not contain o. A *signature* Σ is a finite set of typed, non-logical constants. We often enumerate signatures by listing their members as pairs, written $a\colon \tau$, in which a is a constant of type τ. Although attaching a type in this way is redundant, it makes reading signatures easier. Occurrences of o are restricted 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 $n \geq 0$ and τ_0 is a 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 symbol*. A signature is n^{th}-*order* if all its constants are of order n or less and at least one constant in it is of order n. Only first-order and second-order signatures are used in this paper.

A constant or variable of type τ is a term of type τ. If t is a term of type $\tau \rightarrow \sigma$ and s is a term of type τ, then the application $(t\ s)$ is a term of type σ. Application associates to the left: read the expression $(t_1\ t_2\ t_3)$ as $((t_1\ t_2)\ t_3)$. Finally, if x is a variable of type τ and t is a term of type σ, then the abstraction $\lambda x\ t$ is a term of type $\tau \rightarrow \sigma$. The usual definition of free and bound variable occurrences are assumed as well as the notion of α-conversion. The logical constants $\wedge$ and $\supset$ are written in the familiar infix form. The expression $\forall_\tau(\lambda z\ t)$ is written simply as $\forall_\tau z\ t$. A term of type o is a *formula*. When t and s are λ-terms, the expression $t = s$ denotes the mathematical proposition that t and s are α-convertible.

For variable x and term s of the same type, $t[s/x]$ denotes the operation of substituting s for all free occurrences of x in t, systematically changing bound variable names in t to avoid free variable capture. Besides the relation of α-conversion, terms are also related to other terms by the following rules of β- and η-conversions.

- The term s β-converts to the term s' if s contains a subformula occurrence of the form $(\lambda x\ t)t'$ and s' arises from replacing that subformula occurrence with $t[t'/x]$.

- The term s η-converts to the term s' if s contains a subformula occurrence of the form $\lambda x\ (t\ x)$, in which x is not free in t, and s' arises from replacing that subformula occurrence with t.

The binary relation *conv*, denoting λ-*conversion*, is defined so that t *conv* s if there is a list of terms $t_1, \ldots, t_n$, with $n \geq 1$, t equal to t_1, s equal to t_n, and for $i = 1, \ldots, n-1$, either t_i converts to t_{i+1} or t_{i+1} converts to t_i by α, β, or η conversion. Expressions of the form $\lambda x\ (t\ x)$ are called η-redexes (provided x is not free in t) while expressions of the form $(\lambda x\ t)s$ are called β-redexes. A term is in λ-normal form if it contains no β or η-redexes. Every term can be converted to a λ-normal term, and that normal term is unique up to the name of bound variables. See [HS86] for a more complete discussion of these basic properties of the simply typed λ-calculus.

3.2. Specifications as Formulas

Our specification of evaluators uses a weak extension to Horn clause logic. In particular, let syntactic variables A range over atomic formulas and B range over possibly universally quantified atomic formulas. Formulas used for specifications are closed formulas of the form

$$\forall z_1 \ldots \forall z_m [(B_1 \wedge \ldots \wedge B_n) \supset A] \quad (m, n \geq 0).$$

(If $n = 0$ the implication is not written.) These formulas differ from Horn clauses only in that the B_i's are not necessarily atomic: they may be universally quantified atomic formulas. Occurrences of such universal quantifiers are called *embedded* universal quantifier occurrences. The quantifier occurrences $\forall z_i$ are called *outermost* universal quantifier occurrences.

A *specification* is a pair $\langle \Sigma, \mathcal{P} \rangle$ in which Σ is a signature and $\mathcal{P}$ is a finite, non-empty set of these extended Horn clauses with the following restrictions.

1 The signature Σ is exactly the set of non-logical constants with occurrences in formulas in $\mathcal{P}$. Thus, Σ does not need to be listed separately.

2 If a formula in $\mathcal{P}$ contains an embedded universal quantifier occurrence, then that quantifier occurrence binds a variable of primitive type and Σ is a second-order signature.

3 If Σ is first-order then all of the outermost quantifier occurrences in formulas in $\mathcal{P}$ bind variables of (non-logical) primitive type. In this case, the set $\mathcal{P}$ must contain just first-order Horn clauses; that is, they contain no embedded quantifiers.

Such pairs $\langle \Sigma, \mathcal{P} \rangle$ are used to specify the *extension* to predicates that are members of Σ. In particular, if $p : \tau_1 \to \cdots \to \tau_n \to o$ is a member of Σ, then the tuple $\langle t_1, \ldots, t_n \rangle$ (for closed terms t_i) is in the *extension* of p if and only if the closed formula $(p\ t_1\ \cdots\ t_n)$ is provable from $\mathcal{P}$ and all the constants in the terms $t_1, \ldots, t_n$ are in Σ. Since the formulas permitted in specifications are so weak, classical logic provability here coincides with intuitionistic and minimal logic provability. In each of these cases, equality of terms and formulas is λ-convertibility.

For example, let S be the set $\{i\}$ containing only one primitive type, let Σ_0 be the second-order signature

$$\{abs : (i \to i) \to i,\ app : i \to i \to i,\ eval : i \to i \to o\},$$

and let $\mathcal{P}_0$ be the set containing the two formulas

$$\forall_{i \to i} m\ (eval\ (abs\ m)\ (abs\ m))$$

$$\forall_i p \forall_i q \forall_i v \forall_{i \to i} m\ (eval\ p\ (abs\ m)\ \wedge\ eval\ (m\ q)\ v\ \supset\ eval\ (app\ p\ q)\ v).$$

Untyped λ-terms are represented using the constants *abs* (denoting abstraction) and *app* (denoting application). We discuss this syntax further in Section 4.1. This example has no embedded universal quantifier occurrences but it does have one outermost quantified variable whose type is of order 1. The specification $\langle \Sigma_0, \mathcal{P}_0 \rangle$ is used in Section 4 to give a high-level specification of call-by-name evaluation.

For another example, let $S = \{i, nat, list\}$, and consider the second-order signature

$$\Sigma_1 = \{abs : (i \to i) \to i,\ app : i \to i \to i,\ 0 : nat,\ 1 : nat, + : nat \to nat \to nat,$$
$$nil : list,\ ::\ : i \to list \to list,\ count : list \to i \to nat \to o\}$$

and the set $\mathcal{P}_1$ containing the four formulas

$$\forall_{list} l \forall_i p \forall_i q \forall_{nat} c \forall_{nat} d (count\ l\ p\ c\ \wedge\ count\ l\ q\ d\ \supset\ count\ l\ (app\ p\ q)\ (1 + (c + d)))$$

$$\forall_{list} l \forall_{i \to i} m \forall_i x \forall_{nat} c (\forall_i x (count\ (x :: l)\ (m\ x)\ c)\ \supset\ count\ l\ (abs\ m)\ c)$$

$$\forall_{list} l \forall_i x (count\ (x :: l)\ x\ 0)$$

$$\forall_{list} l \forall_i x \forall_i y \forall_{nat} c (count\ l\ x\ c\ \supset\ count\ (y :: l)\ x\ c).$$

(Here the list constructor :: is written in infix notation.) The pair $\langle \Sigma_1, \mathcal{P}_1 \rangle$ can be used to count the number of occurrences of the constant *app* in a term. In particular, if the atomic formula ($count\ nil\ t\ n$) is provable from $\mathcal{P}_1$ then n is an arithmetic expression equal to the number of occurrences of *app* in t. The functional interpretation of operations such as $+$ are not part of the logic we have described here. For example, if there is a proof of the formula

$$count\ nil\ (abs \lambda x (abs \lambda y (app\ (app\ x\ y)\ (app\ y\ x))))\ n$$

from Σ_1 and $\mathcal{P}_1$ then that proof contains a subproof of the formula

$$count\ (d :: c :: nil)\ (app\ (app\ c\ d)\ (app\ d\ c))\ n,$$

where c and d are eigenvariables of the proof. Thus, eigenvariables can be used to name bound variables when it is important to "descend" through an abstraction. This latter formula is provable if and only if n is the expression $1 + (1 + (0 + 0) + (1 + (0 + 0)))$. Beyond this example, the most sophisticated integer computations we need are those for incrementing positive integers, and for that, special treatment of integers is not necessary. At first glance, it may appear that the first argument of *count* that lists free variables is not needed for this specification. It is only used to determine whether or not the term in the second argument of *count* is a variable. Given the specification of terms that we have picked, it is clear that a term is a variable if and only if it is not an *app* or an *abs*. Specifying such a negative condition, however, is not possible in this meta-logic, and, hence, we are required to replace it with the positive condition of finding the second argument in the first argument.

3.3. Specifications as Proof Systems

Because of the simple structure of formulas in specifications we can describe specifications instead as proof systems by replacing formulas with inference rules. We view a clause of the form

$$\forall z_1 \ldots \forall z_m [B_1 \wedge \ldots \wedge B_n \supset A]$$

as an inference rule of the form

$$\frac{B_1 \ \cdots \ B_n}{A}$$

in which the universal quantifiers $\forall z_1 \ldots \forall z_m$ are implicitly assumed. We use the convention that all capitalized letters occurring in formulas of inference rules are variables implicitly quantified by outermost universal quantifiers. The clauses specifying *count* above can be specified as the following set of inference rules.

$$\frac{count\ L\ P\ C \qquad count\ L\ Q\ D}{count\ L\ (app\ P\ Q)\ (1 + (C + D))} \qquad \frac{\forall_i x(count\ (x :: L)\ (M\ x)\ C)}{count\ L\ (abs\ M)\ C}$$

$$\frac{}{count\ (X :: L)\ X\ 0} \qquad \frac{count\ L\ X\ C}{count\ (Y :: L)\ X\ C}$$

Inference rules of this kind only explicitly mention one logical connective, universal quantification, and occurrences of it are permitted only in the premises of rules. Thus, to complete a proof system we must specify the introduction rule for the universal quantifier. This and the rule for λ-conversion are

$$\frac{B[y/x]}{\forall x\ B}\ \forall\text{-I} \qquad \text{and} \qquad \frac{B}{B'}\ \lambda.$$

These rules have the following provisos: for $\forall$-I, the eigenvariable y must not be in Σ or free in $\forall x\ B$; and for λ, we must have $B\ conv\ B'$. Usual notions of natural deduction [Gen69, Pra65] are assumed. Since there are no implications to introduce, proofs here do not involve the discharging of assumptions. (Adding such occurrences of implication is natural and useful for a wide range of specifications [Han90]; such implications, however, play only a small role in the specification of evaluators and so are not considered here.) A proof system is *first-order* or *second-order* if its equivalent presentation as a specification $\langle \Sigma, \mathcal{P} \rangle$ is such that Σ is first-order or second-order, respectively.

In the rest of this paper, we present specifications as inference rules. It is important to note, however, that proofs in classical (or intuitionistic or minimal) logic using specifications-as-formulas are isomorphic to proofs using specifications-as-proof-systems. The choice of which presentation to use is only one of notational convenience.

3.4. Abstract Evaluation Systems as Proof Systems

While specifications form weak logical systems, they provide high-level mechanisms for the specification of evaluation in object-language functional programs. Many of these mechanisms, such as universal quantification, higher-order types, and λ-conversion, do not correspond to anything that appears in abstract evaluation systems. The following restricted proof systems, however, do correspond to AESs.

Definition 2. A set of first-order inference rules is *AES-defining* if there are two binary predicate symbols p and q (not necessarily distinct) such that

- every axiom is of the form $\dfrac{}{p\ s\ t}$ in which s and t are terms and $\mathcal{V}(t) \subseteq \mathcal{V}(s)$;
- there is a distinguished inference rule of the form $\dfrac{p\ t\ z}{q\ s\ z}$ in which s and t are terms, $\mathcal{V}(t) \subseteq \mathcal{V}(s)$, and z is a variable not free in either s or t; and
- inference rules that are neither axioms nor the distinguished rule are of the form $\dfrac{p\ t\ z}{p\ s\ z}$ in which s and t are terms, $\mathcal{V}(t) \subseteq \mathcal{V}(s)$, and z is a variable that is not free in either s or t.

For every AES-defining proof system $\mathcal{I}$ we can define an AES (Σ, R, ρ, S) (modulo the renaming of free variables) such that the following condition holds: *for all terms s and t, $\mathcal{I} \vdash q\ s\ t$ if and only if there exists some AES-rewriting sequence $s \stackrel{\rho}{\Longrightarrow} s' \stackrel{*}{\Longrightarrow} s'' \stackrel{r}{\Longrightarrow} t$ for some $r \in S$ and some terms s' and s'' ($\stackrel{*}{\Longrightarrow}$ denotes zero or more rewritings using R-rules).* The correspondence between some given $\mathcal{I}$ and (Σ, R, ρ, S) can be described as follows:

- $\mathcal{I}$ contains the axiom $\dfrac{}{p\ s\ t}$ if and only if S contains the rewrite rule $s \Rightarrow t$;
- $\mathcal{I}$ contains the distinguished rule $\dfrac{p\ t\ z}{q\ s\ z}$ if and only if ρ is $s \Rightarrow t$; and
- $\mathcal{I}$ contains the rule $\dfrac{p\ t\ z}{p\ s\ z}$ if and only if R contains $s \Rightarrow t$.

One important characteristic of AES-defining proof systems is the use of an "output" variable in the inference rules. Every inference rule (other than the axioms) is either of the form $\dfrac{p\ t\ z}{p\ s\ z}$ or $\dfrac{p\ t\ z}{q\ s\ z}$ in which z is a variable (implicitly universally quantified). Thus, for every proof in such a proof system there is a single term which is the second argument for every formula occurrence in the proof. Obviously, most specifications of evaluators are not AES-defining proof systems. First-order specifications can fail to satisfy the requirements of AES-defining proof systems in a number of ways. Particular violations that concern us later are proof systems that contain inference rules with

1 multiple formulas in their premises;

2 variables in their premises that do not also occur in their consequents; or

3 the form $\dfrac{p\ t\ y}{p\ s\ z}$ in which y and z are not the same variable or they occur in s or t.

We can systematically convert a proof system containing inference rules with multiple premises to an equivalent (modulo a simple relation between provable formulas) system containing only rules with at most one premise. The latter two cases, however, are more problematic and no single method appears universally applicable.

3.5. Implementations of the Meta-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, then Prolog or the TYPOL language of the CENTAUR system [Kah87] can provide a depth-first interpreters of it. Since the higher-order logic programming language λProlog [NM88] supports higher-order quantification, λ-conversion, and embedded universal quantification, it can be used to give a depth-first implementation of the full meta-logic. Although depth-first interpretation is not generally complete, it is complete for those AES-defining proof systems that are deterministic (at most one rule can be applied at a given point in building a proof in a bottom-up fashion). Because almost all the AES-defining proof systems presented here are deterministic, λProlog can be used to provide correct implementations of them. Since such implementations are also tail-recursive, the result of compiling them should be efficient, iterative programs. By translating the specifications in this paper into λProlog code, we have been able to experiment with them. We found such prototyping and experimentation valuable in understanding the dynamics of various specifications.

4. From Explicit to Implicit Abstractions in Terms

This section begins with the specification of call-by-name and call-by-value as proof systems using the approach described in Section 3.3. Our first specifications represent object-level, untyped λ-terms using a meta-level syntax that employs simply typed λ-terms over a second-order signature. This approach makes it possible to map object-level abstractions directly to meta-level abstractions. Since the elegant and declarative meta-level operations for handling explicit abstractions are available, specifications can be clear, concise, and suitably abstracted from numerous implementation details. As we plan to construct abstract machines that use only first-order terms to encode object-level programs, we must translate terms using a second-order signature in which λ-abstractions are available explicitly to terms over a first-order signature in which abstractions are available only implicitly. The task of changing the specification of evaluation to accommodate this change in term-level syntax is not trivial. This section shows how this change can be effected for call-by-name and call-by-value evaluation.

4.1. Specifying Two Evaluators

In this section and the next, we only consider the untyped λ-calculus since it is rich enough to illustrate several problems in the evaluation of functional program. In Section 6 we consider several of those language extensions needed for more realistic functional programming languages, and demonstrate how our methods can be applied to those extensions.

The untyped λ-terms can be encoded into simply typed λ-terms of base type tm using two constants $abs : (tm \rightarrow tm) \rightarrow tm$ and $app : tm \rightarrow tm \rightarrow tm$, for encoding object-level abstraction and application, respectively. (A formal definition of the encoding and its various properties can be found in [Han90].) Some example terms in this syntax are listed below.

Untyped λ-terms	Simply typed λ-terms
$\boldsymbol{\lambda x.x}$	$(abs\ \lambda u.u)$
$\boldsymbol{\lambda y.y}$	$(\lambda z.z)(abs\ \lambda u.u)$
$(\boldsymbol{\lambda x.x})(\boldsymbol{\lambda y.y})$	$(app\ (abs\ \lambda u.u)\ (abs\ \lambda v.v))$

All three untyped λ-terms are β-convertible, but only the first two are α-convertible. Only the first two simply typed λ-terms are β-convertible. By convention, we typically select the unique (up to α-conversion) simply typed λ-term in $\beta\eta$-long normal form to represent the corresponding untyped λ-term term. That is, although $(abs\ \lambda u.u)$ and $(\lambda z.z)(abs\ \lambda v.v)$ are equal meta-level terms and both encode $\boldsymbol{\lambda x.x}$, we only refer to the $\beta\eta$-long normal term (the first one here) as *the* encoding of the given untyped λ-term.

We observe that α-equivalent terms in the untyped λ-calculus map to λ-convertible terms in the simply typed λ-calculus of the meta-logic and a β-redex in the untyped λ-calculus maps to a meta-level term of the form $(app\ (abs\ s)\ t)$, which is not a meta-level β-redex.

To demonstrate some simple computations using this encoding, consider the following two examples. First, to determine if two terms $p, q : tm$ are representations of α-convertible untyped (object-level) λ-terms, we only need to check if the equality $p = q$ is provable in the meta-logic (such equality contains the meta-level conversion rules for α, β, and η).

For the second example, consider specifying object-level substitution. This can be done with using the 3-place predicate $subst\colon tm \rightarrow tm \rightarrow tm \rightarrow o$, which is axiomatized using the simple axiom

$$\forall_{tm\rightarrow tm} A\ \forall_{tm} B\ (subst\ (abs\ A)\ B\ (A\,B)).$$

Thus if $s, t, u : tm$ are encodings of untyped terms $\lambda x.p$, q, r, respectively, and meta-level proposition $(subst\ s\ t\ u)$ is an instance of the above axiom, then, at the object-level, r is α-convertible to the term $p[q/x]$. (See [Han90] for a proof of this correspondence).

The two inference rules in Figure 3 specify call-by-name evaluation and the two inference rules in Figure 4 specify call-by-value evaluation. In both of these cases, values are λ-terms in weak-head normal form. Rather than explicitly use the axiom for $subst$

$$\frac{}{eval\ (abs\ M)\ (abs\ M)} \qquad \frac{eval\ P\ (abs\ M) \qquad eval\ (M\,Q)\ V}{eval\ (app\ P\ Q)\ V}$$

Fig. 3. High-level specification of call-by-name evaluation

$$\frac{}{eval\ (abs\ M)\ (abs\ M)} \qquad \frac{eval\ P\ (abs\ M) \qquad eval\ Q\ R \qquad eval\ (M\,R)\ V}{eval\ (app\ P\ Q)\ V}$$

Fig. 4. High-level specification of call-by-value evaluation

described above, these evaluators use the meta-level expressions $(M\,Q)$ and $(M\,R)$, respectively, to specify substitution via meta-level β-conversion. Alternatively, we could use the axiom for *subst* by expressing the rule for application (in the call-by-name case) as

$$\frac{eval\ P\ (abs\ M) \qquad subst\ (abs\ M)\ Q\ R \qquad eval\ R\ V}{eval\ (app\ P\ Q)\ V}.$$

Notice that these inference rules contain variables of the functional type $tm \rightarrow tm$ and no rule contains explicit mention of bound variables. If we name the rules in Figure 3 by $\mathcal{E}$, then $\mathcal{E} \vdash eval\ s\ t$ if and only if t encodes the weak-head normal form (whnf) obtained by a normal-order reduction of the untyped λ-term encoded by s. A similar statement can be made for the call-by-value inference rules.

These inference rules provide a high-level specification of evaluation in several senses. Below we focus on two of these senses, namely, the use of meta-level β-conversion to automatically perform object-level substitution of terms for bound variables and the use of λ-terms instead of simpler first-order terms in the meta-language.

4.2. Introducing Closures into the Call-by-Name Evaluator

To prepare for a change in the meta-level representation of untyped λ-terms, we introduce three predicates *isabs*, *isapp* and *apply* into the call-by-name evaluator in order to move the actual representation of terms away from the clauses that directly describe evaluation. The resulting inference system is given in Figure 5. Notice that the constants *abs* and *app* do not occur in those inference rules that mention the *eval* predicate. Using elementary facts about unfolding of inferences rules, it is immediate that the *eval*-facts provable from Figure 3 and from Figure 5 are identical. (A brief discussion of unfolding can be found in Appendix A.)

We now introduce the new constant $clo : tm \rightarrow (tm \rightarrow tm) \rightarrow tm$ to "name" object-level β-redexes. All meta-level terms containing *clo* are λ-convertible to terms of the form

$$(clo\ t_1\ \lambda x_1(\ldots(clo\ t_n\ \lambda x_n\, s)\ldots)) \quad (n \geq 0) \qquad (*)$$

in which s is either $(app\ s_1\ s_2)$ for some s_1 and s_2, $(abs\ s')$ for some s', or one of the bound variables $x_1, \ldots, x_n$. Depending on these three cases, we say that $(*)$ is the closure of an application, the closure of an abstraction, or the closure of a variable, respectively.

$$\frac{isapp\ A\ P\ Q \qquad eval\ P\ R \qquad apply\ Q\ R\ S \qquad eval\ S\ V}{eval\ A\ V}$$

$$\frac{isabs\ P}{eval\ P\ P}$$

$$\frac{}{isabs\ (abs\ M)} \qquad \frac{}{isapp\ (app\ P\ Q)\ P\ Q} \qquad \frac{}{apply\ Q\ (abs\ M)\ (M\ Q)}$$

Fig. 5. Addition of some meta-level predicates.

Furthermore, all terms that appear in evaluators we consider have the additional property that the terms $t_1, \ldots, t_n$ are all closed terms and hence contain no occurrences of the bound variables $x_1, \ldots, x_n$. If $n = 0$ then $(*)$ simply denotes s. Terms of the form $(*)$ are abbreviated using the syntax $(clo\ \bar{t}\ \lambda\bar{x}\ s)$.

Let t be a term built from abs, app, and clo as described above. The term t^0 denotes the λ-normal form of the result of replacing every occurrence of clo in t with the term $\lambda u \lambda v\ (v\ u)$. This mapping replaces all named β-redexes with meta-level β-redexes, which are then removed in the process of forming the λ-normal form.

Since the rules for *isabs*, *isapp*, and *apply* in Figure 5 are sensitive to the particular structure of terms, these rules must be modified to handle terms containing *clo*. For example, rules for *isabs* and *isapp* must now determine whether terms are closures of abstractions and applications. Since terms can also be closures of variables, we introduce the *isvar* predicate to test for such terms and to return the term in the closure that corresponds to that variable. The rules listed in Figure 6 address this additional structure.

As with the *count* example in Section 3.2, when reading the inference rules in Figure 6 from the bottom to top, the eigenvariables introduced by universally quantified premises can be seen as naming the bound variables of the λ-abstraction that they instantiate. Consider, for example, proving the formula

$$isvar\ nil\ (clo\ t_1\ \lambda x_1 (\ldots (clo\ t_n\ \lambda x_n\ s) \ldots))\ t.$$

If this atomic formula has a proof, that proof must also contain a subproof of the formula

$$isvar\ ((c_n, t_n) :: \cdots :: (c_1, t_1) :: nil)\ s'\ t$$

where s' is the result of replacing the bound variables $x_1, \ldots, x_n$ in s with the (distinct) eigenvariables $c_1, \ldots, c_n$, respectively. The proof of this latter formula is essentially the computation that determines which of these eigenvariables is equal to s' and that t is the term associated with that eigenvariable.

Let $\vdash_5$ denote provability from the inference rules in Figure 5 and let $\vdash_6$ denote provability from the inference rules in Figure 6. The proof of the following lemma follows from inspection of the inference rules for the meta-level predicates *isabs*, *isapp*, *isvar* and *apply* given in Figure 6.

Lemma 3. Let r be a term of type tm over the constants abs, app and clo.

$$\frac{isapp\ A\ P\ Q \qquad eval\ P\ R \qquad apply\ Q\ R\ S \qquad eval\ S\ V}{eval\ A\ V}$$

$$\frac{isabs\ P}{eval\ P\ P} \qquad \frac{isvar\ nil\ P\ T \qquad eval\ T\ V}{eval\ P\ V}$$

$$\frac{}{isabs\ (abs\ M)} \qquad \frac{\forall x\ (isabs\ (M\ x))}{isabs\ (clo\ T\ M)}$$

$$\frac{}{isapp\ (app\ P\ Q)\ P\ Q} \qquad \frac{\forall x\ (isapp\ (A\ x)\ (M\ x)\ (N\ x))}{isapp\ (clo\ T\ A)\ (clo\ T\ M)\ (clo\ T\ N)}$$

$$\frac{\forall x\ (isvar\ ((x,T) :: E)\ (M\ x)\ V)}{isvar\ E\ (clo\ T\ M)\ V} \qquad \frac{}{isvar\ ((X,T) :: E)\ X\ T} \qquad \frac{isvar\ E\ X\ T}{isvar\ ((Y,S) :: E)\ X\ T}$$

$$\frac{}{apply\ Q\ (abs\ M)\ (clo\ Q\ M)} \qquad \frac{\forall x\ (apply\ Q\ (M\ x)\ (N\ x))}{apply\ Q\ (clo\ T\ M)\ (clo\ T\ N)}$$

Fig. 6. Evaluation with closures represented using $clo : tm \to (tm \to tm) \to tm$.

1. $\vdash_6$ *isabs* r if and only if r is the closure of an abstraction.
2. $\vdash_6$ *isapp* $r\ r_1\ r_2$ for r_1 and r_2 terms of type tm if and only if r is of the closure of an application, that is, it is of the form $(clo\ \bar{t}\ \lambda\bar{x}\ (app\ z_1\ z_2))$ for some terms z_1 and z_2 of type tm, in which case r_1 is $(clo\ \bar{t}\ \lambda\bar{x}\ z_1)$ and r_2 is $(clo\ \bar{t}\ \lambda\bar{x}\ z_2)$. When this relationship holds, $(app\ r_1^0\ r_2^0) = r^0$.
3. $\vdash_6$ *isvar nil* $r\ s$ if and only if r is the closure of a variable, that is, it has the form
$$(clo\ t_1\ \lambda x_1(\ldots(clo\ t_n\ \lambda x_n\ x_i)\ldots)) \quad (n \geq 1)$$
for some i such that $1 \leq i \leq n$. In this case, s is the term t_i. When this relation holds, $s^0 = r^0$.
4. $\vdash_6$ *apply* $t\ r\ s$ if and only if r is the closure of an abstraction, that is, it is of the form $(clo\ \bar{u}\ \lambda\bar{x}\ (abs\ z))$ for some term z of type $tm \to tm$ and s is of the form $(clo\ \bar{u}\ \lambda\bar{x}\ (clo\ t\ z))$. When this relationship holds, there is a term w of type $tm \to tm$ such that $r^0 = (abs\ w)$ and $(w\ t^0)\ conv\ s^0$. In particular, $\vdash_5$ *apply* $t^0\ r^0\ s^0$.

The following two propositions establish the correspondence between the proof systems in Figures 5 and 6.

Proposition 4. Let t and r be two terms of type tm such that $r^0 = t$. If there is a term v such that $\vdash_5$ *eval* $t\ v$, then there exists a term u such that u is the closure of an abstraction, $\vdash_6$ *eval* $r\ u$, and $u^0 = v$.

Proof. Let t and r be two terms of type tm such that $r^0 = t$, and assume that there is a term v such that $\vdash_5$ *eval* $t\ v$. First, consider the case where r is the closure of a variable. By Lemma 3, $\vdash_6$ *isvar nil* $r\ s$ where s is a proper subterm of r and $s^0 = r^0$. Thus, if we can show that there exists a term u such that $\vdash_6$ *eval* $s\ u$ and $u^0 = v$, then we have $\vdash_6$ *eval* $r\ u$ via the proof rule for *eval* which has (*isvar nil* $r\ s$) as a premise. If s is still the closure of a variable, repeat this step again. This repetition terminates in reducing $\vdash_6$-provability of $(eval\ r\ v)$ to $\vdash_6$-provability of $(eval\ r'\ v)$ where r' is a proper subterm of r, $(r')^0 = r^0$, and r' is the closure of an application or abstraction.

Assuming that r is the closure of an application or abstraction, we can proceed by induction on the structure of proofs in Figure 5 of $(eval\ t\ v)$.

For the base case, assume that $(eval\ t\ v)$ follows from the fact that $t = v$ and $\vdash_5$ *isabs* t. Thus r is the closure of a abstraction and by Lemma 3, $\vdash_6$ *isabs* r, so $\vdash_6$ *eval* $r\ r$. Thus, r is the promised value for u.

For the inductive case, assume that $(eval\ t\ v)$ follows from

$$\vdash_5 isapp\ t\ t_1\ t_2,\ \vdash_5 eval\ t_1\ (abs\ w),\ \vdash_5 apply\ t_2\ (abs\ w)\ s \text{ and } \vdash_5 eval\ s\ v.$$

Since t is the term $(app\ t_1\ t_2)$, it must be the case that r is the closure of an application. By Lemma 3, $\vdash_6$ *isapp* $r\ r_1\ r_2$ for some r_1 and r_2 such that $r_1^0 = t_1$ and $r_2^0 = t_2$. By the inductive hypothesis, there is a u_1 such that $u_1^0 = (abs\ w)$, u_1 is the closure of an abstraction, and $\vdash_6$ *eval* $r_1\ u_1$. By Lemma 3 there is a term z such that $\vdash_6$ *apply* $r_2\ u_1\ z$ and z^0 *conv* $(w\ r_2^0)$. Since s *conv* $(w\ t_2)$, we have s *conv* z^0. Using the inductive hypothesis again, we conclude that there is a u_2 such that $\vdash_6$ *eval* $z\ u_2$, u_2 is the closure of an abstraction, and $u_2^0 = v$. Thus, from

$$\vdash_6 isapp\ r\ r_1\ r_2,\ \vdash_6 eval\ r_1\ u_1,\ \vdash_6 apply\ r_2\ u_1\ z, \text{ and } \vdash_6 eval\ z\ u_2,$$

we can conclude that $\vdash_6$ *eval* $r\ u_2$, which completes this proof. □

The following proposition establishes a converse to the preceding proposition.

Proposition 5. Let r and u be two closed, λ-terms such that $\vdash_6$ *eval* $r\ u$. Then $\vdash_5$ *eval* $r^0\ u^0$.

Proof. A proof of $(eval\ r\ u)$ using Figure 6 can be converted directly to a proof of $(eval\ r^0\ u^0)$ using Figure 5 by the following three steps. First, remove all proof rules above any formula occurrence whose head symbol is either the predicate *isabs*, *isapp*, or *apply*. Second, repeatedly remove instances of the inference rule for *eval* that has *isvar* in its premise by discarding the proof of its left premise and making the proof of the right premise the proof of the conclusion of that inference rule occurrence. Finally, replace every occurrence of a term t that is the argument of some predicate of some formula occurrence in the proof with the term t^0. It is now an easy exercise to verify that the resulting proof is indeed a $\vdash_5$-proof of $(eval\ r^0\ u^0)$. □

The inference rules in Figure 6 successfully avoid performing substitution of terms for bound variables by making use of a closure-like structure. These rules still depend on using λ-abstraction and universal quantification at the meta-level. We are now in a position to remove this dependency.

A de Bruijn-style representation of terms containing *clo* can be built by introducing the first-order constant *clo*$'$. We treat indices in terms containing *clo*$'$ in such a way that the second argument of *clo*$'$ behaves as an abstraction. Consider the first-order signature

$$\{\hat{}: fotm \rightarrow fotm \rightarrow fotm,\ \lambda: fotm \rightarrow fotm,\ clo': fotm \rightarrow fotm \rightarrow fotm,\ var: nat \rightarrow fotm\},$$

where $\hat{}$ denotes application, λ denotes abstraction (this is an overloading of the λ symbol), *clo*$'$ denotes the closure construction, and *var* maps natural numbers into terms encoding bound variables by their offsets. For the sake of brevity, we often abbreviate the term $(var\ n)$ by n. Here, clo' treats its second argument as an abstraction by modifying offsets of variables in that argument. Following [Bru72], closed terms of type tm can be placed in one-to-one correspondence with those terms of type $fotm$ for which all indices denoting bound variables have an offset to an actual occurrence of a λ or a clo'. Using this first-order signature, the λ-term $(clo\ t_1\ \lambda x_1(\ldots(clo\ t_n\ \lambda x_n\ s)\ldots))$ translates to the first-order term $(clo'\ t'_1\ (\ldots(clo'\ t'_n\ s')\ldots))$ of type $fotm$, where $t'_1, \ldots, t'_n, s'$ are the translations of $t_1, \ldots, t_n, s$, respectively. An index i at the position of s denotes a variable to be replaced by the term t_{n-i}.

Since clo' is being used now to attach a list of terms $t_1, \ldots, t_n$ to the term s, clo' can be replaced with explicit lists; that is, the above term can be represented simply as $\{t'_n :: \cdots :: t'_1 :: nil, s'\}$. Of course, now an index of i at the position of s must be interpreted as referring to the $i+1^{\text{th}}$ member of the associated list of terms and the construction $\{\_,\_\}$ has type $fotm\ list \times fotm \rightarrow fotm$. For example, the term

$$(clo\ (abs\ \lambda x.x)\ \lambda u(clo\ (abs\ \lambda x(abs\lambda y\ y))\ \lambda v(app\ v\ u)))$$

would be translated to the term of type $fotm$ $\{\lambda\lambda 0 :: \lambda 0 :: nil,\ 0\ \hat{}\ 1\}$.

This final representation of syntax has several advantages over the one using clo or clo'. First, it involves first-order constants only. Second, simple pattern matching can determine if a term embedded in a closure is an abstraction, application, or variable index; recursing down a series of clo's is no longer needed. Third, the reversing of closures involved in proving $isvar$ is not needed and the auxiliary list argument to $isvar$ can be dropped. Finally, this syntax also makes it natural to identify a term s that is not a top-level $\{\_,\_\}$ with $\{nil, s\}$ and to identify the doubly nested closure expression $\{\ell, \{t :: nil, s\}\}$ with simply $\{t :: \ell, s\}$. Given this change in the representation of terms, we can easily rewrite our inference rules to those presented in Figure 7. As in the transformation of the rules in Figure 5 and Figure 6, the only rules that change are those involved with the definition of the predicates $isabs$, $isapp$, $isvar$, and $apply$.

If t is a term of type tm built from the constants abs, app, and clo, let t^* be the corresponding de Bruijn-style representation of t of type $fotm$ over the constants $\hat{}$, λ, $\{\_,\_\}$, and var. The proof of the following proposition follows immediately from the motivation of the inference rules in Figure 7 given above.

Proposition 6. Let t and v be two terms of type tm built from the constants abs, app, and clo. Then $\vdash_6 eval\ t\ v$ if and only if $\vdash_7 eval\ t^*\ v^*$.

Assuming that we are only interested in proving $eval$-atoms, the inference rules in Figure 7 can be simplified, using unfolding transformations, to the inference rules $\mathcal{N}_0$, which are displayed in Figure 8. These are further transformed in Section 5.

4.3. Introducing Closures for Call-by-Value

We can follow the approach above to construct a first-order specification of the call-by-value evaluator in Figure 4. Because the essential steps are the same as for the call-by-name case, with most of the rules in each system being the same, we present here only the different inference rules required for call-by-value evaluation.

$$\frac{isapp\ A\ P\ Q \qquad eval\ P\ R \qquad apply\ Q\ R\ S \qquad eval\ S\ V}{eval\ A\ V}$$

$$\frac{isabs\ P}{eval\ P\ P} \qquad \frac{isvar\ P\ T \quad eval\ T\ V}{eval\ P\ V}$$

$$\frac{}{isabs\ \{E, \lambda M\}} \qquad \frac{}{isapp\ \{E, P\,\hat{}\,Q\}\ \{E, P\}\ \{E, Q\}}$$

$$\frac{}{isvar\ \{T :: E, 0\}\ T} \qquad \frac{isvar\ \{E, n\}\ T}{isvar\ \{X :: E, n+1\}\ T}$$

$$\frac{}{apply\ Q\ \{E, \lambda M\}\ \{Q :: E, M\}}$$

Fig. 7. Evaluation using list structures for closures.

$$\frac{}{eval\ \{E, \lambda M\}\ \{E, \lambda M\}} \qquad (\mathcal{N}_0.1)$$

$$\frac{eval\ \{E, M\}\ \{E', \lambda M'\} \qquad eval\ \{\{E, N\} :: E', M'\}\ V}{eval\ \{E, M\,\hat{}\,N\}\ V} \qquad (\mathcal{N}_0.2)$$

$$\frac{eval\ X\ V}{eval\ \{X :: E, 0\}\ V} \qquad (\mathcal{N}_0.3)$$

$$\frac{eval\ \{E, n\}\ V}{eval\ \{X :: E, n+1\}\ V} \qquad (\mathcal{N}_0.4)$$

Fig. 8. The $\mathcal{N}_0$ proof system

The first step in the section above introduces the new predicates *isabs*, *isapp*, and *apply*. The exact same modifications of proof systems can be done for the call-by-value case with the only difference being the rule for application:

$$\frac{isapp\ A\ P\ Q \qquad eval\ P\ R \qquad eval\ Q\ S \qquad apply\ S\ R\ T \qquad eval\ T\ V}{eval\ A\ V}.$$

The next step, which introduces the constant *clo*, follows as before, although the inference rule for evaluating closures of variables is now just

$$\frac{isvar\ nil\ P\ V}{eval\ P\ V},$$

as no further evaluation is required. So the call-by-value version at this stage is just the set of rules in Figure 6, except for this rule and the rule for application, which is as above. The transition to a proof system that employs the first-order signature over type *fotm* follows as before, and all the rules are the same except the ones corresponding to the

$$\frac{}{eval\ \{E, \lambda M\}\ \{E, \lambda M\}} \quad (\mathcal{V}_0.1)$$

$$\frac{eval\ \{E, M\}\ \{E', \lambda M'\} \qquad eval\ \{E, N\}\ R \qquad eval\ \{R :: E', M'\}\ V}{eval\ \{E, (M\ \hat{}\ N)\}\ V} \quad (\mathcal{V}_0.2)$$

$$\frac{}{eval\ \{X :: E, 0\}\ X} \quad (\mathcal{V}_0.3)$$

$$\frac{eval\ \{E, n\}\ V}{eval\ \{X :: E, n+1\}\ V} \quad (\mathcal{V}_0.4)$$

Fig. 9. The $\mathcal{V}_0$ proof system

two rules noted above. Finally, when we unfold these rules to simplify their structure, we obtain the proof system $\mathcal{V}_0$, displayed in Figure 9.

5. Constructing Two Abstract Machines

In this section we transform the two first-order evaluators listed at the end of Section 4 into abstract machines. The transformation steps used in this section vary in generality: some apply to a wide range of inference rules specifying evaluation strategies while some are particular to a given evaluator. In all cases, however, transformations are motivated by two goals. The first goal is to continue making a specification less dependent on the meta-logic in which it is situated. For example, in the previous section, we replaced explicit abstractions using simply typed λ-terms with implicit abstractions using de Bruijn indices and first-order terms. Here we go further by eliminating, for example, any need for general (first-order) unification during proof construction. Our second goal is to commit to certain decisions left implicit or undetermined in a specification so that the resulting specification immediately yields an explicit algorithm.

5.1. Constructing a Call-by-Name Machine

We wish to convert the inference rules of system $\mathcal{N}_0$ (presented at the end of Section 4.2) to an AES-defining set of inference rules to obtain an abstract machine that implements (weak head) normal-order reduction for untyped λ-terms. The proof rules in $\mathcal{N}_0$ fail to be AES-defining for two reasons. First, proofs in $\mathcal{N}_0$ branch when an application is evaluated. Second, the rule for application has two variables, E' and M', in the premise that are not in the conclusion. The first problem is solved by a general technique; the second problem, however, seems to have no general solution. The bulk of this section describes a series of transformations to address this second problem.

We first address the branching structure of proofs in $\mathcal{N}_0$. Rule $\mathcal{N}_0.2$ has two premises. During bottom-up proof construction, proofs for these two premises can be constructed in either order, or even in parallel. The choice is left to the particular implementation of the meta-logic. We can choose a sequential strategy and enforce its use by applying the following transformation. We introduce the new predicate symbol $prove : (o\ list) \rightarrow o$

$$\frac{}{prove\ nil} \qquad (\mathcal{N}_1.1)$$

$$\frac{prove\ G}{prove\ (eval\ \{E, \lambda M\}\ \{E, \lambda M\}) :: G} \qquad (\mathcal{N}_1.2)$$

$$\frac{prove\ (eval\ \{E, M\}\ \{E', \lambda M'\}) :: (eval\ \{\{E, N\} :: E', M'\}\ V) :: G}{prove\ (eval\ \{E, M\ \hat{}\ N\}\ V) :: G} \qquad (\mathcal{N}_1.3)$$

$$\frac{prove\ (eval\ X\ V) :: G}{prove\ (eval\ \{X :: E, 0\}\ V) :: G} \qquad (\mathcal{N}_1.4)$$

$$\frac{prove\ (eval\ \{E, n\}\ V) :: G}{prove\ (eval\ \{X :: E, n+1\}\ V) :: G} \qquad (\mathcal{N}_1.5)$$

Fig. 10. The $\mathcal{N}_1$ proof system

and the atomic axiom (*prove nil*). Each inference rule

$$\frac{A_1 \cdots A_n}{A_0} \quad \text{of } \mathcal{N}_0 \text{ is rewritten as} \quad \frac{prove\ A_1 :: \cdots :: A_n :: G}{prove\ A_0 :: G}$$

for variable G (of type ($o\ list$)) that is not free in any A_i. This kind of transformation is applicable to any proof system in which all inference rules contain only atomic formulas as premises. In the case of $\mathcal{N}_0$ this transformation produces the new system $\mathcal{N}_1$ given in Figure 10. The argument to *prove* represents the collection of formulas that remain to be proved. This list behaves like a stack during bottom-up proof construction, with the first formula in this list always chosen as the next one to prove. Constants such as *prove* of type $(o\ list) \rightarrow o$ are not formally allowed in our meta-logic (argument types are not permitted occurrences of the type o). To be formally correct here is simple: introduce a new primitive type, say b, let *prove* be of type $(b\ list) \rightarrow o$ instead and replace occurrences of *eval* in $\mathcal{N}_1$ with a new constant, say *evl*, of type $tm \rightarrow tm \rightarrow b$. For simplicity we continue to use just the type o.

Lemma 7. For all closed atomic formula $(eval\ \{\ell, s\}\ v)$,

$$\mathcal{N}_0 \vdash (eval\ \{\ell, s\}\ v) \quad \text{if and only if} \quad \mathcal{N}_1 \vdash prove\ (eval\ \{\ell, s\}\ v) :: nil.$$

The proof uses a straightforward induction on the structure of proofs in the two systems.

With $\mathcal{N}_1$ we have solved the problem of having inference rules with multiple premises. We are left to consider the problem of variables occurring in the premise but not in the conclusion of a rule. For the case of $\mathcal{N}_1$ this concerns rule $\mathcal{N}_1.3$. We can characterize this problem more generally, however, in the context of operational semantics. These distinguished variables typically serve as placeholders or temporary names for intermediate results used in computing the final result of the computation specified by the inference rule. This idea is discussed in [How91] where a class of rules called *evaluation rules* is defined. Our concern here is in providing a technique for eliminating explicit reference to these variables.

For the case of $\mathcal{N}_1$ we exploit properties particular to proofs in this system (and subsequent ones as we introduce new proof systems). Unfortunately, these techniques do not extend to general, widely applicable strategies, but they do serve to illustrate the kind of reasoning possible for manipulating proof systems. In devising the steps involved in the following construction we were assisted by knowing the approximate structure of the resulting abstract machine.

A useful manipulation of proof systems involves introducing partial instantiations of an inference rule. Partial instances of a rule are constructed by substituting (possibly open) terms for some of the schema variables of the rule, with the free variables of these terms becoming schema variables of the new rule. (A schema variable is one that is implicitly universally quantified.) Clearly we can always add (to a proof system) instances of a rule already in the system. Furthermore, we can replace a rule by a set of its partial instances if every instance of the original rule is also an instance of one of the partial instances. We apply this idea to rule $\mathcal{N}_1.2$. The variable G in this rule is of type ($o\ list$) and because there are only two list constructors (nil and $::$), all possible instances of G are also instances of either nil or $A :: G'$ for some instances of A and G'. Thus, we can replace rule $\mathcal{N}_1.2$ by the two inference rules:

$$\frac{prove\ nil}{prove\ (eval\ \ \{L, \lambda M\}\ \ \{L, \lambda M\}) :: nil} \qquad (\mathcal{N}_1.2a)$$

$$\frac{prove\ A :: G'}{prove\ (eval\ \ \{L, \lambda M\}\ \ \{L, \lambda M\}) :: A :: G'}. \qquad (\mathcal{N}_1.2b)$$

Notice that the premise of $\mathcal{N}_1.2a$ is always trivially provable. In fact, we can unfold the rules $\mathcal{N}_1.1$ and $\mathcal{N}_1.2a$ to produce the axiom

$$\frac{}{prove\ (eval\ \ \{L, \lambda M\}\ \ \{L, \lambda M\}) :: nil}. \qquad (\mathcal{N}_1.2a')$$

Further, notice that rules $\mathcal{N}_1.2b, 3, 4, 5$ all have premises whose list arguments are non-nil and hence instances of $\mathcal{N}_1.1$ cannot occur immediately above instances of these rules. If we use rule $\mathcal{N}_1.2a'$ then we no longer need rule $\mathcal{N}_1.1$ for non-trivial proofs. Taking rules $\mathcal{N}_1.2a', 2b, 3, 4, 5$ (relabeling them as shown) yields the $\mathcal{N}_2$ proof system displayed in Figure 11.

Lemma 8. For all s, t, $\mathcal{N}_1 \vdash (prove\ (eval\ s\ t) :: nil)$ if and only if $\mathcal{N}_2 \vdash (prove\ (eval\ s\ t) :: nil)$.

The proof of this Lemma is a straightforward induction on the size of $\mathcal{N}_1$ and $\mathcal{N}_2$ proofs and uses the reasoning outlined above.

Note that $\mathcal{N}_1$ and $\mathcal{N}_2$ are not precisely equivalent since $\mathcal{N}_1 \vdash (prove\ nil)$ but $\mathcal{N}_2 \not\vdash (prove\ nil)$. Since we are only interested in atomic *prove* statements that contain a non-empty list of formulas, this discrepancy is irrelevant.

The premise of rule $\mathcal{N}_2.3$ contains two occurrences of each of the two variables L' and M' which do not occur in the conclusion. The following proposition describes a redundancy within these proof rules that allows us to remove one occurrence of each of L' and M'.

$$\frac{}{prove\ (eval\ \{L, \lambda M\}\ \{L, \lambda M\}) :: nil} \qquad (\mathcal{N}_2.1)$$

$$\frac{prove\ A :: G}{prove\ (eval\ \{L, \lambda M\}\ \{L, \lambda M\}) :: A :: G} \qquad (\mathcal{N}_2.2)$$

$$\frac{prove\ (eval\ \{L, M\}\ \{L', \lambda M'\}) :: (eval\, \{\{L, N\} :: L', M'\}\ V) :: G}{prove\ (eval\, \{L, M \mathbin{\hat{}} N\}\ V) :: G} \qquad (\mathcal{N}_2.3)$$

$$\frac{prove\ (eval\ X\ V) :: G}{prove\ (eval\ \{X :: L, 0\}\ V) :: G} \qquad (\mathcal{N}_2.4)$$

$$\frac{prove\ (eval\ \{L, n\}\ V) :: G}{prove\ (eval\ \{X :: L, n+1\}\ V) :: G} \qquad (\mathcal{N}_2.5)$$

Fig. 11. The $\mathcal{N}_2$ proof system

Proposition 9. Let a be a formula, let Π be an $\mathcal{N}_2$ proof of $(prove\ a :: nil)$ and let $(prove\ a_1 :: \cdots :: a_n :: nil)$, for $n \geq 2$, be a formula occurring in Π. Then for all $i = 1, \ldots, n-1$, a_i has the form $(eval\ s\ \{\ell, \lambda t\})$ and a_{i+1} has the form $(eval\ \{s' :: \ell, t\}\ v)$ for some terms s, s', ℓ, t and v.

Proof. Assume that the proposition does not hold for some formula a and proof Π. Let $prove\ a_1 :: \cdots :: a_n :: nil$ be the formula in Π closest to the root that does not have the desired form. Since $n \neq 1$, this atom is the premise of some inference rule. That inference rule must be $\mathcal{N}_2.3$ since the conclusion of any other inference rule applied to that formula would not have the desired form. If the inference rule was $\mathcal{N}_2.3$, then the first two atoms, a_1 and a_2, do have the required form. Hence, some pair in $a_2 :: \cdots :: a_n :: nil$ must not have the required form, and again the conclusion of this inference rule does not have the require form, contradicting the assumption. □

Thus, every instance of the inference rule $\mathcal{N}_2.2$ in an $\mathcal{N}_2$ proof of $(prove\ a :: nil)$ is also an instance of

$$\frac{prove\ (eval\, \{\{L', M'\} :: L, M\}\ V) :: G}{prove\ (eval\ \{L, \lambda M\}\ \{L, \lambda M\}) :: (eval\, \{\{L', M'\} :: L, M\}\ V) :: G}.$$

This inference rule could therefore replace $\mathcal{N}_2.2$. The structural information of Proposition 9 can be used in a more interesting fashion: we can introduce a new predicate symbol $eval' : fotm \to fotm \to o$ and modify $\mathcal{N}_2$ to get the proof system $\mathcal{N}_3$ so that atomic formulas of the form

$$\begin{aligned} prove \quad & (eval\ c_1\ \{\ell_1, \lambda t_1\}) :: (eval\ \{c_2 :: \ell_1, t_1\}\ \{\ell_2, \lambda t_2\}) :: \\ & \cdots :: (eval\ \{c_n :: \ell_{n-1}, t_{n-1}\}\ \{\ell_n, \lambda t_n\}) :: nil \end{aligned}$$

in $\mathcal{N}_2$ are replaced by formulas of the form

$$prove\ (eval\ c_1\ \{\ell_1, \lambda t_1\}) :: (eval'\ c_2\ \{\ell_2, \lambda t_2\}) :: \cdots :: (eval'\ c_n\ \{\ell_n, \lambda t_n\}) :: nil$$

in $\mathcal{N}_3$ proofs. Here, a new predicate $eval'$ is used to show that a variant of $eval$ is intended: while the proposition $(eval\ s\ t)$ states that s evaluates to t, the proposition

$$\frac{}{prove\ (eval\ \{L, \lambda M\}\ \{L, \lambda M\}) :: nil} \qquad (\mathcal{N}_3.1)$$

$$\frac{prove\ (eval\{\{L_1, M_1\} :: L, M\}\ V) :: G}{prove\ (eval\ \{L, \lambda M\}\ \{L, \lambda M\}) :: (eval'\{L_1, M_1\}\ V) :: G} \qquad (\mathcal{N}_3.2)$$

$$\frac{prove\ (eval\ \{L, M\}\ \{L', \lambda M'\}) :: (eval'\ \{L, N\}\ V) :: G}{prove\ (eval\ \{L, M\ \hat{}\ N\}\ V) :: G} \qquad (\mathcal{N}_3.3)$$

$$\frac{prove\ (eval\ X\ V) :: G}{prove\ (eval\ \{X :: L, 0\}\ V) :: G} \qquad (\mathcal{N}_3.4)$$

$$\frac{prove\ (eval\ \{L, n\}\ V) :: G}{prove\ (eval\ \{X :: L, n+1\}\ V) :: G} \qquad (\mathcal{N}_3.5)$$

Fig. 12. The $\mathcal{N}_3$ proof system

$(eval'\ s\ t)$ occurring in the context

$$(eval\ s'\ \{\ell, \lambda t'\}) :: (eval'\ s\ t) :: \cdots :: nil$$

states that $\{s :: \ell, t'\}$ evaluates to t. The proof system $\mathcal{N}_3$ is displayed in Figure 12.

Lemma 10. For all s, t, $\mathcal{N}_2 \vdash (prove\ (eval\ s\ t)::nil)$ if and only if $\mathcal{N}_3 \vdash (prove\ (eval\ s\ t):: nil)$.

The proof follows from Proposition 9 and the discussion above. Note that the distinction between the predicate symbols *eval* and *eval′* is semantic. Syntactically, however, we can use just one symbol instead of the two if we restrict our attention to formulas of the form

$$prove\ (eval\ c_1\ \{\ell_1, \lambda t_1\}) :: (eval'\ c_2\ \{\ell_2, \lambda t_2\}) :: \cdots :: (eval'\ c_n\ \{\ell_n, \lambda t_n\}) :: nil$$

(for $n \geq 1$) because only the rules $\mathcal{N}_3.2$ and $\mathcal{N}_3.3$ manipulate *eval′* predicates and always as the second element in a list of formulas.

The next transformation is a simple reorganization of data structures that exploits the isomorphism between a list-of-pairs and a pair-of-lists. The constants *eval* and *eval′* can be viewed as pairing constructor (pairing a term with a value). The predicate *prove* takes a list of such pairs. An equivalent formulation uses the binary predicate $prove_1 : (fotm\ list) \rightarrow (fotm\ list) \rightarrow o$ that takes a pair of lists. The explicit pairing constructors can be eliminated in this way. This is done in the proof system $\mathcal{N}_4$ displayed in Figure 13.

Note that for any formula occurring in an $\mathcal{N}_4$ proof we can easily construct the corresponding formula occurrence in an $\mathcal{N}_3$ proof: use *eval* to pair together the first elements in the lists, and use *eval′* to pair together all the remaining elements. Given the motivation for $\mathcal{N}_4$ above and the fact that the syntactic distinction between the constructors *eval* and *eval′* is not relevant, the following is immediate.

Lemma 11. For all s, t, $\mathcal{N}_3 \vdash (prove\ \ (eval\ s\ t) :: nil)$ if and only if $\mathcal{N}_4 \vdash (prove_1\ \ s :: nil\ \ t :: nil)$.

Recall that an AES-defining system must have an output variable as part of the atomic

$$\frac{}{prove_1 \quad \{L, \lambda M\} :: nil \quad \{L, \lambda M\} :: nil} \tag{$\mathcal{N}_4.1$}$$

$$\frac{prove_1 \quad \{\{L_1, M_1\} :: L, M\} :: S \quad V :: T}{prove_1 \quad \{L, \lambda M\} :: \{L_1, M_1\} :: S \quad \{L, \lambda M\} :: V :: T} \tag{$\mathcal{N}_4.2$}$$

$$\frac{prove_1 \quad \{L, M\} :: \{L, N\} :: S \quad \{L', \lambda M'\} :: V :: T}{prove_1 \quad \{L, M \mathbin{\hat{}} N\} :: S \quad V :: T} \tag{$\mathcal{N}_4.3$}$$

$$\frac{prove_1 \quad X :: S \quad V :: T}{prove_1 \quad \{X :: L, 0\} :: S \quad V :: T} \tag{$\mathcal{N}_4.4$}$$

$$\frac{prove_1 \quad \{L, n\} :: S \quad V :: T}{prove_1 \quad \{X :: L, n+1\} :: S \quad V :: T} \tag{$\mathcal{N}_4.5$}$$

Fig. 13. The $\mathcal{N}_4$ proof system

$$\frac{}{prove_2 \quad \{L, \lambda M\} :: nil \quad \{L, \lambda M\} :: nil \quad \{L, \lambda M\}} \tag{$\mathcal{N}_5.1$}$$

$$\frac{prove_2 \quad \{\{L_1, M_1\} :: L, M\} :: S \quad V :: T \quad Z}{prove_2 \quad \{L, \lambda M\} :: \{L_1, M_1\} :: S \quad \{L, \lambda M\} :: V :: T \quad Z} \tag{$\mathcal{N}_5.2$}$$

$$\frac{prove_2 \quad \{L, M\} :: \{L, N\} :: S \quad \{L', \lambda M'\} :: V :: T \quad Z}{prove_2 \quad \{L, M \mathbin{\hat{}} N\} :: S \quad V :: T \quad Z} \tag{$\mathcal{N}_5.3$}$$

$$\frac{prove_2 \quad X :: S \quad V :: T \quad Z}{prove_2 \quad \{X :: L, 0\} :: S \quad V :: T \quad Z} \tag{$\mathcal{N}_5.4$}$$

$$\frac{prove_2 \quad \{L, n\} :: S \quad V :: T \quad Z}{prove_2 \quad \{X :: L, n+1\} :: S \quad V :: T \quad Z} \tag{$\mathcal{N}_5.5$}$$

Fig. 14. The $\mathcal{N}_5$ proof system

formula being proved. This variable is set by the axioms and is preserved by all the other inference rules. Towards such a structure for our rules, we state the following trivial proposition.

Proposition 12. For any terms s, t, the leaf of an $\mathcal{N}_4$ proof of $(prove_1 \;\; s :: nil \;\; t :: nil)$ must be an occurrence of the formula $(prove_1 \;\; t :: nil \;\; t :: nil)$.

We exploit this property of proofs by introducing a third argument to the $prove_1$ predicate, modifying the one $\mathcal{N}_4$ axiom to identify this new argument with the final value t from the second argument $t :: nil$ and modifying the remaining inference rules to preserve the value of this argument. The resulting proof system, called $\mathcal{N}_5$ and displayed in Figure 14, replaces $prove_1$ with the new predicate $prove_2 : (fotm\ list) \to (fotm\ list) \to fotm \to o$ of three arguments.

Lemma 13. For all s, t, $\mathcal{N}_4 \vdash (prove_1 \;\; s :: nil \;\; t :: nil)$ if and only if $\mathcal{N}_5 \vdash (prove_2 \;\; s :: nil \;\; t :: nil \;\; t)$.

The proof follows from Proposition 12.

$$\frac{}{prove_3\ \{L, \lambda M\} :: nil\ \{L, \lambda M\}} \qquad (\mathcal{N}_6.1)$$

$$\frac{prove_3\ \{\{L', N\} :: L, M\} :: S\ Z}{prove_3\ \{L, \lambda M\} :: \{L', N\} :: S\ Z} \qquad (\mathcal{N}_6.2)$$

$$\frac{prove_3\ \{L, M\} :: \{L, N\} :: S\ Z}{prove_3\ \{L, M\ \hat{}\ N\} :: S\ Z} \qquad (\mathcal{N}_6.3)$$

$$\frac{prove_3\ X :: S\ Z}{prove_3\ \{X :: L, 0\} :: S\ Z} \qquad (\mathcal{N}_6.4)$$

$$\frac{prove_3\ \{L, n\} :: S\ Z}{prove_3\ \{X :: L, n+1\} :: S\ Z} \qquad (\mathcal{N}_6.5)$$

Fig. 15. The $\mathcal{N}_6$ proof system

Now, $\mathcal{N}_5$ is not an AES-defining system only because in rule $\mathcal{N}_5.3$ the variables L' and M' occur in the premise but not in the conclusion. Consider simplifying the inference rules of $\mathcal{N}_5$ by replacing each inference rule of the form

$$\frac{(prove_2\ t_1'\ t_2'\ t_3')}{(prove_2\ t_1\ t_2\ t_3)} \quad \text{with} \quad \frac{(prove_3\ t_1'\ t_3')}{(prove_3\ t_1\ t_3)}$$

where $prove_3 : (fotm\ list) \rightarrow fotm \rightarrow o$ is a new binary predicate. This kind of generalization is complementary to the instantiation transformations that we applied earlier. The resulting system, called $\mathcal{N}_6$, is displayed in Figure 15.

Observe that all schema variables free in the premise of any $\mathcal{N}_6$ rule are also free in the conclusion of that rule. Furthermore, observe that for every $\mathcal{N}_5$ inference rule, the schema variables occurring free in the second argument of $prove_2$ occur nowhere else in the premise. Thus this second argument is, in some sense, independent of the other arguments in premise formulas. Observe also that in any $\mathcal{N}_5$-provable formula the first and second arguments are always lists of the same length and the second argument is always of the form $\{\ell_1, \lambda s_1\} :: \{\ell_2, \lambda s_2\} :: \cdots :: nil$. Thus for any $\mathcal{N}_5$-provable formula, if the first and third arguments of this formula match the corresponding arguments in the premise of an inference rule, then the second argument must also match its corresponding argument in the premise of the rule. That is, the second argument supplies no constraint on the construction of $\mathcal{N}_5$ proofs and so can be safely discarded. This fact is formalized in the proof of the following proposition.

Lemma 14. For all s, t, $\mathcal{N}_5 \vdash (prove_2\ s{::}nil\ t{::}nil\ t)$ if and only if $\mathcal{N}_6 \vdash (prove_3\ s{::}nil\ t)$.

Proof. The proof in the forward direction is immediate: given a proof in $\mathcal{N}_5$ simply delete the second argument of all atomic formulas and change $prove_2$ to $prove_3$ throughout. We prove the reverse direction by proving the slightly more general statement: for all $n \geq 0$, and terms $s, s_1, \ldots, s_n, t$, if $\mathcal{N}_6 \vdash (prove_3\ \ s_n :: \ldots :: s_1 :: s :: nil\ \ t)$ then there exists terms $t_1, \ldots, t_n$ such that $\mathcal{N}_5 \vdash (prove_2\ \ s_n :: \ldots :: s_1 :: s :: nil\ \ t_n :: \cdots :: t_1 :: t :: nil\ \ t)$. The proof proceeds by induction on the height h of an $\mathcal{N}_6$ proof Π.

Base: For $h = 1$, Π must be of the form

$$\frac{}{(prove_3\ \ \{\ell, \lambda s\} :: nil\ \ \{\ell, \lambda s\})}$$

and so $n = 0$. The corresponding $\mathcal{N}_5$ proof is simply

$$\frac{}{(prove_2\ \ \{\ell, \lambda s\} :: nil\ \ \{\ell, \lambda s\} :: nil\ \ \{\ell, \lambda s\})}.$$

Step: Assume the statement holds for proofs of height $h \geq 1$ and let Π be an $\mathcal{N}_6$ proof of $(prove_3\ \ s_n :: \ldots :: s_1 :: s :: nil\ \ t)$ with height $h+1$. We show that there exists an $\mathcal{N}_5$ proof Δ of $(prove_2\ \ s_n :: \ldots :: s_1 :: s :: nil\ \ \ell\ \ t)$ for some $\ell = t_n :: \cdots :: t_1 :: t :: nil$.

Let $(prove_3\ \ s'_m :: \ldots :: s'_1 :: s' :: nil\ \ t)$ be the premise of the last inference of Π. By the inductive hypothesis, there exist terms $t_1, \ldots, t_m$ such that $(prove_2\ \ s'_m :: \ldots :: s'_1 :: s' :: nil\ \ t_m :: \cdots :: t_1 :: t :: nil\ \ t)$ has an $\mathcal{N}_5$ proof Δ'. We proceed now by cases according to the last inference rule of Π.

1. If the last rule is $\mathcal{N}_6.4$ ($\mathcal{N}_6.5$) then $n = m$, and let $\ell = t_m :: \cdots :: t_1 :: t :: nil$. The required proof Δ is then built from Δ' and $\mathcal{N}_5.4$ ($\mathcal{N}_5.5$).
2. If the last rule is $\mathcal{N}_6.2$ then $n = m+1$ and $s'_m = \{\{\ell_1, m_1\} :: \ell_2, m_2\}$ for some ℓ_1, m_1, ℓ_2, m_2. Let $\ell = \{\ell_2, \lambda m_2\} :: t_m :: \cdots :: t_1 :: t :: nil$. Again the required proof Δ is constructed from Δ' and $\mathcal{N}_5.2$.
3. If the last rule is $\mathcal{N}_6.3$ then $m = n+1$ and $m > 0$. Let $\ell = t_{m-1} :: \cdots :: t_1 :: t :: nil$. The proof Δ is constructed from Δ' and $\mathcal{N}_5.3$.

It is trivial to verify that in each case the ℓ we construct is of the form $t_n :: \cdots :: t_1 :: t :: nil$ for some $t_1, \ldots, t_n$. Letting $n = 0$ yields the required proof for the lemma. □

We now can state the main result of this section.

Theorem 15. For all s, t, $\mathcal{N}_0 \vdash (eval\ s\ t)$ if and only if $\mathcal{N}_6 \vdash (prove_3\ s :: nil\ t)$.

The proof follows from linking together all the lemmas in this section.

Notice that $\mathcal{N}_6$ is simply an AES-defining proof system for call-by-name evaluation if we add to it the load rule

$$\frac{prove_3\ \ \{nil, M\} :: nil\ \ Z}{cbn\ M\ Z}.$$

The abstract machine encoded by this proof system is given in Figure 16. Its state consists of a list (or stack) of closures $\{E, M\} :: S$. Compare this with the description in Figure 2 in which the state consists of the triple $\langle E, M, S \rangle$.

5.2. Constructing a Call-by-Value Machine

In this section we construct a machine for call-by-value evaluation using a strategy similar to the one applied in the previous section. Starting with the first-order specification $\mathcal{V}_0$ derived in Section 4.3, we construct the proof system $\mathcal{V}_1$ (displayed in Figure 17) in which inference rules involving atomic formulas are replaced by inference rules involving stacks of such formulas.

$$\begin{array}{|rcr|}
\hline
M & \Rightarrow & \{nil, M\} :: nil \\
\hline
\{E, M\,\hat{}\,N\} :: S & \Rightarrow & \{E, M\} :: \{E, N\} :: S \\
\{E, \lambda M\} :: \{E', N\} :: S & \Rightarrow & \{\{E', N\} :: E, M\} :: S \\
\{X :: E, 0\} :: S & \Rightarrow & X :: S \\
\{X :: E, n+1\} :: S & \Rightarrow & \{E, n\} :: S \\
\hline
\{E, \lambda M\} :: nil & \Rightarrow & \{E, \lambda M\} \\
\hline
\end{array}$$

Fig. 16. A call-by-name abstract machine

$$\frac{}{prove\ nil} \qquad (\mathcal{V}_1.1)$$

$$\frac{prove\ G}{prove\ (eval\ \{E, \lambda M\}\ \{E, \lambda M\}) :: G} \qquad (\mathcal{V}_1.2)$$

$$\frac{\begin{array}{l} prove\ (eval\ \{E, M\}\ \{E', \lambda M'\}) :: (eval\ \{E, N\}\ R) \\ \qquad\qquad\qquad\qquad :: (eval\ \{R :: E', M'\}\ V) :: G \end{array}}{prove\ (eval\ \{E, M\,\hat{}\,N\}\ V) :: G} \qquad (\mathcal{V}_1.3)$$

$$\frac{prove\ G}{prove\ (eval\ \{X :: E, 0\}\ X) :: G} \qquad (\mathcal{V}_1.4)$$

$$\frac{prove\ (eval\ \{E, n\}\ V) :: G}{prove\ (eval\ \{X :: E, n+1\}\ V) :: G} \qquad (\mathcal{V}_1.5)$$

Fig. 17. The $\mathcal{V}_1$ proof system

Lemma 16. For all terms ℓ, s and v,

$$\mathcal{V}_0 \vdash (eval\ \{\ell, s\}\ v) \quad \text{if and only if} \quad \mathcal{V}_1 \vdash (prove\ (eval\ \{\ell, s\}\ v) :: nil).$$

The proof is similar to the proof of Lemma 7.

As with the call-by-name case we are again faced with the task of eliminating variables that occur only in the premise of a rule. The problematic variable occurrences in this case are E' and M' in rule $\mathcal{V}_1.3$. There seems to be no simple generalization to Proposition 9 that reveals any redundancy in the inference rules of $\mathcal{V}_1$. Thus, we appeal to a more general transformation (described in more detail in [Han90]) for introducing value stacks. We give here just an outline of this transformation and its applicability to operational semantics.

The formula $(eval\ p\ v)$ represents a relation between an input program p and an output value v. We can formalize this notion of input and output by considering the following general form of an inference rule.

$$\frac{prove\ (eval\ p_1\ v_1) :: (eval\ p_2\ v_2) :: \cdots :: (eval\ p_n\ v_n) :: G}{prove\ (eval\ p\ v) :: G} \qquad (*)$$

The use of explicit lists of formulas enforces a left-to-right order in which these formulas are solved during bottom-up proof construction. Assume that during proof construction p is a closed term (known) and v is a variable (unknown) yet to be instantiated with a closed term. To ensure that a program p_i depends only on known information we restrict occurrences of free variables as follows: every free variable of p_i must either occur in p or in some v_j for some $j < i$. Thus an instance of program p_i can be determined from the program p (which we assume is given as input) and the results v_j of previous computations. We also assume that every variable occurring in v occurs in some v_i $(1 \leq i \leq n)$. For a similar characterization of input and output within evaluation rules, see [How91].

If such variable restrictions hold for all inference rules in a proof system then we can reformulate the inference rules into a more functional, rather than relational, style. Instead of a binary predicate $eval : fotm \rightarrow fotm \rightarrow o$ taking a program and a value as arguments we use a unary predicate $eval_1 : fotm \rightarrow instr$ taking just a program as an argument and producing an instruction (using the new type *instr*). The unary predicate $prove : (o\ list) \rightarrow o$ is replaced by a ternary predicate $prove_1 : (instr\ list) \rightarrow (fotm\ list) \rightarrow fotm \rightarrow o$. The first argument is a list of instructions, replacing the list of predicates. The second argument is a *value stack* representing the values produced by instructions that have already been evaluated. The third argument represents the result of the original program (occurring at the root of the proof tree). Now the basic idea of the transformation we employ is that the value or result of a program is pushed onto the value stack during bottom-up proof construction. So if $(prove\ (eval\ p\ v) :: nil)$ is provable then a proof of $(prove_1\ \ (eval_1\ p) :: \ell\ \ s\ \ v_0)$ must have a subproof $(prove_1\ \ \ell\ \ v :: s\ \ v_0)$. This transformation directly addresses the elimination of meta-level variables occurring in the premise, but not conclusion, of inference rules. These variables provide access to intermediate results, and this is the same service the stack provides. The full details of this transformation are presented in [Han90].

Using this transformation we can construct the new proof system $\mathcal{V}_2$ (given in Figure 18) from $\mathcal{V}_1$. This step introduces a new constant $\phi : instr$.

Lemma 17. For all terms p and v, $\mathcal{V}_1 \vdash (prove\ \ (eval\ \ p\ \ v) :: nil)$ if and only if $\mathcal{V}_2 \vdash (prove_1\ (eval_1\ p) :: nil\ \ nil\ \ v)$.

The proof follows immediately from Theorem 7.11 of [Han90].

In any $\mathcal{V}_2$-provable proposition $(prove_1\ \ell\ s\ z)$, the list ℓ consists only of $eval_1$ or ϕ instructions. As $eval_1$ and ϕ are the only constructors for *instr*, we can replace the terms of type *instr* with ones of type *fotm*. We first replace $prove_1$ with a new predicate $prove_2 : (fotm\ list) \rightarrow (fotm\ list) \rightarrow fotm \rightarrow o$; then replace terms $(eval_1\ p)$ with just p; and replace $\phi : instr$ with a new constant $\psi : fotm$. Performing this transformation yields $\mathcal{V}_3$, displayed in Figure 19.

Lemma 18. For all terms p and v, $\mathcal{V}_2 \vdash (prove_1\ \ (eval_1\ \ p) :: nil\ nil\ v)$ if and only if $\mathcal{V}_3 \vdash (prove_2\ p :: nil\ nil\ v)$.

The proof is a trivial induction on the height of proof trees.

For any $\mathcal{V}_3$-provable proposition $(prove_2\ \ell\ s\ v)$, ℓ is a list in which each element is either a closure $\{e, t\}$ or the term ψ. We can view this as a list of only closures if

$$\frac{}{prove_1 \;\; nil \;\; V :: nil \;\; V} \qquad (\mathcal{V}_2.1)$$

$$\frac{prove_1 \;\; C \;\; \{E, \lambda M\} :: S \;\; V}{prove_1 \;\; (eval_1 \;\; \{E, \lambda M\}) :: C \;\; S \;\; V} \qquad (\mathcal{V}_2.2)$$

$$\frac{prove_1 \;\; (eval_1 \;\; \{E, M\}) :: (eval_1 \;\; \{E, N\}) :: \phi :: C \;\; S \;\; V}{prove_1 \;\; (eval_1 \;\; \{E, M \,\hat{}\, N\}) :: C \;\; S \;\; V} \qquad (\mathcal{V}_2.3)$$

$$\frac{prove_1 \;\; C \; X :: S \;\; V}{prove_1 \;\; (eval_1 \;\; \{X :: E, 0\}) :: C \;\; S \;\; V} \qquad (\mathcal{V}_2.4)$$

$$\frac{prove_1 \;\; (eval_1 \;\; \{E, n\}) :: C \;\; S \;\; V}{prove_1 \;\; (eval_1 \;\; \{X :: E, n+1\}) :: C \;\; S \;\; V} \qquad (\mathcal{V}_2.5)$$

$$\frac{prove_1 \; (eval_1 \;\; \{R :: E, M\}) :: C \;\; S \;\; V}{prove_1 \;\; \phi :: C \;\; R :: \{E, \lambda M\} :: S \;\; V} \qquad (\mathcal{V}_2.6)$$

Fig. 18. The $\mathcal{V}_2$ proof system

$$\frac{}{prove_2 \; nil \; V :: nil \; V} \qquad (\mathcal{V}_3.1)$$

$$\frac{prove_2 \; C \;\; \{E, \lambda M\} :: S \; V}{prove_2 \;\; \{E, \lambda M\} :: C \; S \; V} \qquad (\mathcal{V}_3.2)$$

$$\frac{prove_2 \;\; \{E, M\} :: \{E, N\} :: \psi :: C \; S \; V}{prove_2 \;\; \{E, M \,\hat{}\, N\} :: C \; S \; V} \qquad (\mathcal{V}_3.3)$$

$$\frac{prove_2 \; C \; X :: S \; V}{prove_2 \;\; \{X :: E, 0\} :: C \; S \; V} \qquad (\mathcal{V}_3.4)$$

$$\frac{prove_2 \;\; \{E, n\} :: C \; S \; V}{prove_2 \;\; \{X :: E, n+1\} :: C \; S \; V} \qquad (\mathcal{V}_3.5)$$

$$\frac{prove_2 \;\; \{R :: E, M\} :: C \; S \; V}{prove_2 \; \psi :: C \;\; R :: \{E, \lambda M\} :: S \; V} \qquad (\mathcal{V}_3.6)$$

Fig. 19. The $\mathcal{V}_3$ proof system

we introduce a new symbol $\psi' : fotm$ and replace ψ with a dummy closure $\{nil, \psi'\}$. We can again replace such a list with a pair of lists by introducing a new predicate $prove_3 : ((fotm\ list)\ list) \rightarrow (fotm\ list) \rightarrow (fotm\ list) \rightarrow fotm \rightarrow o$. Now instead of having a list of pairs (the first argument of $prove_2$), we have a pair of lists (the first two arguments of $prove_3$). We are thus led to the proof system $\mathcal{V}_4$ displayed in Figure 20.

Lemma 19. $\mathcal{V}_3 \vdash (prove_2 \;\; \{\ell, m\} :: nil \;\; nil \;\; v)$ if and only if $\mathcal{V}_4 \vdash (prove_3 \;\; \ell :: nil \;\; m :: nil \;\; nil \;\; v)$.

The proof is a trivial induction on the height of proof trees.

The dummy environment associated with the constant ψ' can be eliminated since it contains no information and it only appears on top of the environment stack E when ψ'

$$\frac{}{prove_3\ \ nil\ \ nil\ \ V :: nil\ \ V} \qquad (\mathcal{V}_4.1)$$

$$\frac{prove_3\ \ L\ \ C\ \ \{E, \lambda M\} :: S\ \ V}{prove_3\ \ E :: L\ \ \lambda M :: C\ \ S\ \ V} \qquad (\mathcal{V}_4.2)$$

$$\frac{prove_3\ \ E :: E :: nil :: L\ \ M :: N :: \psi' :: C\ \ S\ \ V}{prove_3\ \ E :: L\ \ (M\,\hat{}\,N) :: C\ \ S\ \ V} \qquad (\mathcal{V}_4.3)$$

$$\frac{prove_3\ \ L\ \ C\ \ X :: S\ \ V}{prove_3\ \ (X :: E) :: L\ \ 0 :: C\ \ S\ \ V} \qquad (\mathcal{V}_4.4)$$

$$\frac{prove_3\ \ E :: L\ \ n :: C\ \ S\ \ V}{prove_3\ \ (X :: E) :: L\ \ n+1 :: C\ \ S\ \ V} \qquad (\mathcal{V}_4.5)$$

$$\frac{prove_3\ \ (R :: E) :: L\ \ M :: C\ \ S\ \ V}{prove_3\ \ nil :: L\ \ \psi' :: C\ \ R :: \{E, \lambda M\} :: S\ \ V} \qquad (\mathcal{V}_4.6)$$

Fig. 20. The $\mathcal{V}_4$ proof system

is on top of the term stack C. Rules $\mathcal{V}_4.3$ and $\mathcal{V}_4.6$ can be replaced by the following two rules.

$$\frac{prove_3\ \ E :: E :: L\ \ M :: N :: \psi' :: C\ \ S\ \ V}{prove_3\ \ E :: L\ \ M\,\hat{}\,N :: C\ \ S\ \ V} \qquad (\mathcal{V}_4.3')$$

$$\frac{prove_3\ \ (R :: E) :: L\ \ M :: C\ \ S\ \ V}{prove_3\ \ L\ \ \psi' :: C\ \ R :: \{E, \lambda M\} :: S\ \ V} \qquad (\mathcal{V}_4.6')$$

If we combine the first three arguments of $prove_3$ into a tuple (making $prove_3$ a binary predicate) then the resulting proof system is AES-defining. Figure 21 contains the corresponding abstract evaluation system: the CLS machine. (We have switched the order of the first two arguments.) It can be viewed as a dumpless SECD machine. As in the previous section we can introduce an appropriate rule to load the machine. This can be derived by considering how the original binary predicate $(eval\ P\ V)$ has evolved into the four-place predicate $(prove_3\ L\ C\ S\ V)$. The first argument of $prove_3$, a stack of environments, corresponds to the environment (E) and dump (D) of the SECD. The second argument of $prove_3$ corresponds to the code stack (C) of the SECD. The third argument of $prove_3$ acts as the argument stack (S) of the SECD. The constant ψ' in our machine corresponds to the instruction ap (or @) found in the description of the SECD machine [Lan64]. This simplified machine avoids using an explicit dump in favor of maintaining a stack of environments such that the i^{th} term on the code stack is evaluated with respect to the i^{th} environment on the E stack (ignoring the occurrences of ψ' which are not evaluated with respect to an environment). While the original SECD machine stores the entire state on the dump, the only required information is the old environment and the code associated with it. Comparing this machine with the SECD machine given in Figure 2, we note a one-to-one correspondence between the rules in the two machines, except for the two rules that manipulate the dump of the SECD.

M	$\Rightarrow$	$\langle M :: nil,\quad nil :: nil,\quad nil\rangle$
$\langle \lambda M :: C,\quad E :: L,\quad S\rangle$	$\Rightarrow$	$\langle C,\quad L,\quad \{E, \lambda M\} :: S\rangle$
$\langle (M \mathbin{\hat{}} N) :: C,\quad E :: L,\quad S\rangle$	$\Rightarrow$	$\langle M :: N :: \psi' :: C,\quad E :: E :: L,\quad S\rangle$
$\langle 0 :: C,\quad (X :: E) :: L,\quad S\rangle$	$\Rightarrow$	$\langle C,\quad L,\quad X :: S\rangle$
$\langle n+1 :: C,\quad (X :: E) :: L,\quad S\rangle$	$\Rightarrow$	$\langle n :: C,\quad E :: L,\quad S\rangle$
$\langle \psi' :: C,\quad L,\quad R :: \{E, \lambda M\} :: S\rangle$	$\Rightarrow$	$\langle M :: C,\quad (R :: E) :: L,\quad S\rangle$
$\langle nil,\quad nil,\quad V :: S\rangle$	$\Rightarrow$	V

Fig. 21. The CLS machine

There is a close relationship between our CLS machine and the Categorical Abstract Machine (CAM) [CCM87]. The latter can be viewed as a compiled version of the former. Both utilize a stack of environments and a stack of result values, though in the CAM these two stacks have been merged into one structure. The codes for the two machines are different. The CLS machine operates on a list of λ-terms (plus the constant ψ') while the CAM operates on a list of instructions for manipulating its stack. These instructions do, however, correspond closely to operations performed on terms in the CLS, and recent work has shown this correspondence by defining the compilation of λ-terms into CAM instructions from the definition of the CLS machine [Han91].

6. Language and Machine Extensions

In this section we consider some extensions to our simple, object-level programming language, making it more realistic. We demonstrate how these new features can be specified via our high-level inference rules, and we outline how the transformations of the previous section apply to these new features.

6.1. The Second-Order Lambda Calculus

The object-language we have considered so far is just the untyped λ-calculus. Considering the simply typed calculus as the object-language adds nothing new to the theory of evaluation, since evaluation, in this case, can be viewed as an untyped operation. We can just erase the types of a simply typed term and apply untyped reduction to these terms.

A more interesting case involves the second-order lambda calculus, or System F [Gir86], in which types play a more important role during reduction. We begin by defining the syntax for this calculus.

The types of F are given by the following grammar, where α ranges over type variables.

$$T ::= \alpha \mid T \rightarrow T \mid \forall\alpha.T.$$

The terms of F are defined inductively as follows:

1 for any type σ, we have a denumerable set of variables of type σ, and these are all terms of type σ;
2 if t is a term of type τ and x^σ is a variable of type σ, then $\lambda x^\sigma.t$ is a term of type $\sigma \rightarrow \tau$;
3 if t and u are terms of types $\sigma \rightarrow \tau$ and σ, respectively, then $(t\ u)$ is a term of type τ;
4 if t is a term of type σ and α is a type variable, such that for all free variables x^τ in t, α does not occur free in τ, then $\Lambda\alpha.t$ is a term of type $\forall\alpha.\sigma$; and
5 if t is a term of type $\Lambda\alpha.\sigma$ and τ is a type, then $t\{\tau\}$ is a term of type $\sigma[\tau/\alpha]$.

Reduction in System F can be characterized by two rules:

$$(\lambda x^\alpha.t)u \Longrightarrow t[u/x^\alpha] \qquad \text{and} \qquad (\Lambda\alpha.t)\{\tau\} \Longrightarrow t[\tau/\alpha],$$

with the first just being the typed version of β-reduction for term application. Our goal is to axiomatize normal-order reduction (to weak head normal form) as we did with the untyped calculus. We start by defining an abstract syntax for types and terms of System F. We assume two (meta-level) syntactic sorts, ty and tm, to denote object-level types and terms, respectively, of System F. The following signature will be used to present the abstract syntax of such terms and types.

$$\begin{array}{ll}
\Rightarrow : ty \rightarrow ty \rightarrow ty & app : tm \rightarrow tm \rightarrow tm \\
pi : (ty \rightarrow ty) \rightarrow ty & abs : ty \rightarrow (tm \rightarrow tm) \rightarrow tm \\
 & tapp : tm \rightarrow ty \rightarrow tm \\
 & tabs : (ty \rightarrow tm) \rightarrow tm.
\end{array}$$

Types are constructed using the two constants $\Rightarrow$ (infix) and pi, for example, $(pi\ \lambda\alpha(\alpha \Rightarrow \alpha))$. Terms are constructed using the remaining constants. Term application is denoted as app; term abstraction is denoted as abs, where its first argument is the type of the abstracted variable. Type application is denoted by $tapp$ and type abstraction is denoted by $tabs$. For example, we can represent the term $\Lambda\beta((\Lambda\alpha(\lambda x^\alpha.x))\{\beta \rightarrow \beta\})$ by the term

$$(tabs\ \lambda b(tapp\ (tabs\ \lambda a(abs\ a\ \lambda x.x))\ (b \Rightarrow b))).$$

As we did in Section 4, we can axiomatize a reduction relation by introducing a predicate symbol $eval : tm \rightarrow tm \rightarrow o$. The four rules in Figure 22 axiomatize normal-order reduction to weak head normal form. Notice how meta-level β-reduction provides object-level substitution of both terms and types.

Applying techniques analogous to those of Section 4, we can introduce a first-order syntax for this calculus, using de Bruijn-style notation. This syntax is given by the following signature

$$\begin{array}{lll}
\Rightarrow : foty \rightarrow foty \rightarrow foty & \hat{}\ : (fotm \times fotm) \rightarrow fotm & \text{(infix)} \\
\forall : foty \rightarrow foty & \lambda : (foty \times fotm) \rightarrow fotm & \\
tvar : nat \rightarrow foty & \hat{}\hat{}\ : (fotm \times foty) \rightarrow fotm & \text{(infix)} \\
 & \Lambda : fotm \rightarrow fotm & \\
 & var : nat \rightarrow fotm &
\end{array}$$

(The overloading of the type for $\Rightarrow$ should not cause any confusion). Type application is written $t\ \hat{}\hat{}\ \tau$ for term t and type τ. Type abstraction is written Λt for term t. The

$$\frac{}{eval\ (abs\ T\ M)\ (abs\ T\ M)} \qquad \frac{}{eval\ (tabs\ H)\ (tabs\ H)}$$

$$\frac{eval\ P\ (abs\ T\ M) \qquad eval\ (M\, Q)\ V}{eval\ (app\ P\ Q)\ V}$$

$$\frac{eval\ P\ (tabs\ H) \qquad eval\ (H\, T)\ V}{eval\ (tapp\ P\ T)\ V}$$

Fig. 22. Call-by-name evaluation for System F

$$\frac{}{eval\ \{E, \lambda(T, M)\}\ \{E, \lambda(T, M)\}} \qquad \frac{}{eval\ \{E, \Lambda H\}\ \{E, \Lambda H\}}$$

$$\frac{eval\ \{E, P\}\ \{E', \lambda(T, M)\} \qquad eval\ \{\{E, Q\} :: E', M\}\ V}{eval\ \{E, P \,\hat{}\, Q\}\ V}$$

$$\frac{eval\ \{E, P\}\ \{E', \Lambda H\} \qquad eval\ \{\{\!|E, T|\!\} :: E', H\}\ V}{eval\ \{E, P \,\hat{}\hat{}\, T\}\ V}$$

$$\frac{eval\ X\ V}{eval\ \{X :: E, 0\}\ V} \qquad \frac{eval\ \{E, n\}\ V}{eval\ \{X :: E, n+1\}\ V}$$

Fig. 23. A first-order specification of call-by-name evaluation for System F

non-negative integers are de Bruijn indices and are coerced into types *foty* and *fotm* by *tvar* and *var*, respectively. For brevity, we simply write n for both $(tvar\ n)$ and $(var\ n)$. Here, λ, Λ, and $\forall$ act as nameless binders and an occurrence of a (type or term) variable is given by an index that refers to the number of λ's, Λ's, and $\forall$'s to its (the variable's) binding occurrence. For example, the term $\Lambda\alpha.\lambda x^{\alpha}.x$ becomes $\Lambda\lambda(1, 0)$. The 1 refers to the type variable bound by Λ and the 0 refers to the term variable bound by λ. The abstract syntax of the term $\Lambda\beta((\Lambda\alpha(\lambda x^{\alpha}.x))\{\beta \rightarrow \beta\})$ over this signature is $\Lambda((\Lambda(\lambda\ (1, 0)) \,\hat{}\hat{}\, (0 \Rightarrow 0)))$.

Translating the evaluation specification above into one using this new syntax, following the approach of Section 4, yields the proof system in Figure 23. We have introduced a new form of closure, $\{\!|E, T|\!\}$, denoting the closure of type T with environment E, as types may now contain variables that are bound in the environment, just as with terms.

Finally, by following the steps of Section 5.1, we can produce an abstract machine corresponding to these rules and it is given in Figure 24.

Evaluation here is not sufficiently different than in the untyped case, since types are never evaluated and once they are moved into a closure, they are never retrieved. This example is included here largely to show that the transformation techniques of the previous two sections can be applied equally well in this typed situation.

$$
\begin{array}{rcl}
M & \Rightarrow & \{nil, M\} :: nil \\
\hline
\{E, M \,\hat{}\, N\} :: S & \Rightarrow & \{E, M\} :: \{E, N\} :: S \\
\{E, \lambda M\} :: \{E', N\} :: S & \Rightarrow & \{\{E', N\} :: E, M\} :: S \\
\{X :: E, 0\} :: S & \Rightarrow & X :: S \\
\{X :: E, n+1\} :: S & \Rightarrow & \{E, n\} :: S \\
\{E, M \,\hat{}\,\hat{}\, T\} :: S & \Rightarrow & \{E, M\} :: \{\!|E, T|\!\} :: S \\
\{E, \Lambda M\} :: \{\!|E', T|\!\} :: S & \Rightarrow & \{\{\!|E', T|\!\} :: E, M\} :: S \\
\hline
\{E, \lambda M\} :: nil & \Rightarrow & \{E, \lambda M\} \\
\{E, \Lambda M\} :: nil & \Rightarrow & \{E, \Lambda M\}
\end{array}
$$

Fig. 24. A machine for System F

6.2. Recursive Expressions

Recursion can be introduced into our object-language by introducing a constant that acts like the $\boldsymbol{Y}$ combinator. In particular, let the constant $fix : (tm \rightarrow tm) \rightarrow tm$ be added to the second-order signature used in the beginning of Section 4. Evaluation of recursive expressions is given simply by the rule

$$\frac{eval\ (M\,(fix\ M))\ V}{eval\ (fix\ M)\ V}.$$

Operationally, this rule performs an unwinding of the recursive definition by using meta-level β-reduction to do substitution. Those transformations used in Section 4 that replaced a second-order signature with a first-order signature can be applied also to this rule. The second-order constant fix must be replaced with a first-order constant $\mu : fotm \rightarrow fotm$ that acts as a binder just like the constant λ. The corresponding transformation of the above evaluation rule, which can be added to both the call-by-name proof system $\mathcal{N}_0$ and the call-by-value proof system $\mathcal{V}_0$, is the following rule.

$$\frac{eval\ \{\{E, \mu M\} :: E, M\}\ V}{eval\ \{E, \mu M\}\ V}.$$

Notice that this rule does not violate any of the conditions for AES-defining proof systems. The sequence of transformations applied to proof systems in Section 5 can be applied to both the proof systems $\mathcal{N}_0$ and $\mathcal{V}_0$ extended with this rule for recursion. In the call-by-name case, we would need to simply add the rewrite rule

$$\langle \{E, \mu M\} :: S \rangle \quad \Rightarrow \quad \langle \{\{E, \mu M\} :: E,\ M\} :: S \rangle$$

to the machine in Figure 16.

If we perform the series of transformations used for call-by-value evaluation, we need to add to proof system $\mathcal{V}_4$ the inference rule

$$\frac{prove_3\ (\{E, \mu M\} :: E) :: L\quad M :: C\quad S\quad Z}{prove_3\ E :: L\quad \mu M :: C\quad S\quad Z},$$

which translates to the rewrite rule

$$\langle \mu M :: C,\ E :: L,\ S\rangle \quad \Rightarrow \quad \langle M :: C,\ (\{E, \mu M\} :: E) :: L,\ S\rangle.$$

This rewrite rule can be added to the specification of the CLS machine in Figure 21.

6.3. Conditional Expressions

Introducing a conditional expression to our object-language provides a challenge requiring a new transformation. Using the constants $if : tm \to tm \to tm \to tm$, $true : tm$, and $false : tm$ we can construct terms representing conditional expressions. Evaluation of such expressions is specified via the two inference rules

$$\frac{eval\ P\ true \qquad eval\ Q\ V}{eval\ (if\ P\ Q\ R)\ V} \quad \text{and} \quad \frac{eval\ P\ false \qquad eval\ R\ V}{eval\ (if\ P\ Q\ R)\ V}.$$

The transformations to abstract syntax that uses a first-order signature are straightforward and yield the two rules

$$\frac{eval\ \{E, P\}\ true \qquad eval\ \{E, Q\}\ V}{eval\ \{E, (if\ P\ Q\ R)\}\ V}$$

$$\frac{eval\ \{E, P\}\ false \qquad eval\ \{E, R\}\ V}{eval\ \{E, (if\ P\ Q\ R)\}\ V}.$$

(Here, we assume that the types of if, $true$, and $false$ now involve the primitive type $fotm$ instead of tm.) Let $\mathcal{V}_0'$ be $\mathcal{V}_0$ extended with these two new rules (we only consider the more difficult call-by-value situation here). Transforming $\mathcal{V}_0$ into $\mathcal{V}_1$ required introducing the $prove$ predicate and an explicit list of formulas. Following this transformation here yields $\mathcal{V}_1'$, which is just $\mathcal{V}_1$ extended with the rules

$$\frac{prove\ (eval\ \{E, P\}\ true) :: (eval\ \{E, Q\}\ V) :: G}{prove\ (eval\ \{E, (if\ P\ Q\ R)\}\ V) :: G}$$

$$\frac{prove\ (eval\ \{E, P\}\ false) :: (eval\ \{E, R\}\ V) :: G}{prove\ (eval\ \{E, (if\ P\ Q\ R)\}\ V) :: G}.$$

Paralleling the transformation from $\mathcal{V}_1$ to $\mathcal{V}_2$ yields $\mathcal{V}_2'$, which is $\mathcal{V}_2$ plus the following four rules:

$$\frac{prove_1\ \ (eval_1\ \{E, P\}) :: (\phi_1\ E\ Q) :: C\ \ S\ \ Z}{prove_1\ \ (eval_1\ \{E, (if\ P\ Q\ R)\}) :: C\ \ S\ \ Z}$$

$$\frac{prove_1\ \ (eval_1\ \{E, P\}) :: (\phi_2\ E\ R) :: C\ \ S\ \ Z}{prove_1\ \ (eval_1\ \{E, (if\ P\ Q\ R)\}) :: C\ \ S\ \ Z}$$

$$\frac{prove_1\ \ (eval_1\ \{E, Q\}) :: C\ \ S\ \ Z}{prove_1\ \ (\phi_1\ E\ Q) :: C\ \ true :: S\ \ Z} \qquad \frac{prove_1\ \ (eval_1\ \{E, R\}) :: C\ \ S\ \ Z}{prove_1\ \ (\phi_2\ E\ R) :: C\ \ false :: S\ \ Z}.$$

Notice that the evaluation of a conditional uses the same stack that is used for functional application. The final two transformations of $\mathcal{V}_2$ to $\mathcal{V}_4$ were simplifications of data structures, and these two transformations apply directly to the rules above, yielding the following four rules that must be added to $\mathcal{V}_4$ to derive $\mathcal{V}_4'$:

$$\frac{prove_3\ \ E :: E :: L\ \ P :: (\psi_1\ Q) :: C\ \ S\ \ Z}{prove_3\ \ E :: L\ \ (if\ P\ Q\ R) :: C\ \ S\ \ Z}$$

$$\frac{prove_3\ \ E :: E :: L\ \ P :: (\psi_2\ R) :: C\ \ S\ \ Z}{prove_3\ \ E :: L\ \ (if\ P\ Q\ R) :: C\ \ S\ \ Z}$$

$$\frac{prove_3\ \ E :: L\ \ Q :: C\ \ S\ \ Z}{prove_3\ \ E :: L\ \ (\psi_1\ Q) :: C\ true :: S\ Z} \qquad \frac{prove_3\ \ E :: L\ \ R :: C\ \ S\ \ Z}{prove_3\ \ E :: L\ \ (\psi_2\ R) :: C\ false :: S\ Z}.$$

Notice that $(\phi_1\ E\ Q)$ and $(\phi_2\ E\ R)$ have been replaced by just $(\psi_1\ Q)$ and $(\psi_2\ R)$ in this transformation since the environment E appears at the front of the list in the first argument of $prove_2$ in those inference rules that "introduce" ϕ_1 and ϕ_2.

The proof system $\mathcal{V}_4'$ is an AES-defining proof system, and the abstract machine it encodes is formed by adding the following rewrite rules to those in Figure 21.

$$\begin{array}{llrlll}
\langle E :: L, & (if\ P\ Q\ R) :: C, & S\rangle & \Rightarrow & \langle E :: E :: L, & P :: (\psi_1\ Q) :: C,\ \ S\rangle \\
\langle E :: L, & (if\ P\ Q\ R) :: C, & S\rangle & \Rightarrow & \langle E :: E :: L, & P :: (\psi_2\ R) :: C,\ \ S\rangle \\
\langle E :: L, & (\psi_1\ Q) :: C, & true :: S\rangle & \Rightarrow & \langle\ \ E :: L, & Q :: C,\ \ S\rangle \\
\langle E :: L, & (\psi_2\ R) :: C, & false :: S\rangle & \Rightarrow & \langle\ \ E :: L, & R :: C,\ \ S\rangle
\end{array}$$

This abstract machine is not as simple as we would like it to be because the first two rewrites share the same left-hand side. This means that rewriting would need to perform backtracking in order to do evaluation. Appendix B contains a transformation that can be used to "factor" the common parts of these two evaluation rules. Applying this factoring transformation to the four inference rules that are added to $\mathcal{V}_4$ yields the following three rules:

$$\frac{prove_3\ \ E :: E :: L\ \ P :: (choice\ (\psi_1\ Q) :: nil\ \ (\psi_2\ R) :: nil) :: C\ S\ Z}{prove_3\ \ E :: L\ (if\ P\ Q\ R) :: C\ \ S\ \ Z}$$

$$\frac{prove_3\ \ L\ \ C_1 \mathbin{@} C\ \ S\ \ Z}{prove_3\ \ L\ \ (choice\ C_1\ C_2) :: C\ \ S\ \ Z} \qquad \frac{prove_3\ \ L\ \ C_2 \mathbin{@} C\ \ S\ \ Z}{prove_3\ \ L\ \ (choice\ C_1\ C_2) :: C\ \ S\ \ Z}.$$

Here, the infix symbol @ is used to denote list concatenation.

Because the only rule that mentions *choice* in the premise is the first rule above, the last two rules can be replaced by the following instantiations (making the obvious simplification of $(\alpha :: nil) \mathbin{@} C$ to $\alpha :: C$):

$$\frac{prove_3\ \ L\ \ (\psi_1\ Q) :: C\ \ S\ \ Z}{prove_3\ \ L\ \ (choice\ (\psi_1\ Q) :: nil\ (\psi_2\ R) :: nil) :: C\ \ S\ \ Z}$$

$$\frac{prove_3\ \ L\ \ (\psi_2\ R) :: C\ \ S\ \ Z}{prove_3\ \ L\ \ (choice\ (\psi_1\ Q) :: nil\ (\psi_2\ R) :: nil) :: C\ \ S\ \ Z}.$$

We can now unfold these two rules with the two containing ψ_1 and ψ_2 in their conclusions.

This yields the following two rules:

$$\frac{prove_3 \quad E :: L \quad Q :: C \quad S \quad Z}{prove_3 \quad E :: L \quad (choice\ (\psi_1\ Q) :: nil\ (\psi_2\ R) :: nil) :: C\ true :: S\ Z}$$

$$\frac{prove_3 \quad E :: L \quad R :: C \quad S \quad Z}{prove_3 \quad E :: L \quad (choice\ (\psi_1\ Q) :: nil\ (\psi_2\ R) :: nil) :: C\ false :: S\ Z}.$$

These two rules plus the if introduction rule above provide a suitable definition for the conditional expression. The resulting proof system is AES-defining and rewriting will not require any backtracking. The corresponding abstract machine is given by adding the following rewrites:

$$\begin{array}{l} \langle E :: L, \quad (if\ P\ Q\ R) :: C, \quad S\rangle \quad \Rightarrow \\ \qquad\qquad\qquad \langle E :: E :: L, \quad P :: (choice\ (\psi_1\ Q) :: nil \quad (\psi_2\ R) :: nil) :: C, \quad S\rangle \end{array}$$

$$\begin{array}{l} \langle E :: L,\ (choice\ (\psi_1\ Q) :: nil \quad (\psi_2\ R) :: nil) :: C, \quad true :: S\rangle \ \Rightarrow\ \langle E :: L,\ Q :: C,\ S\rangle \\ \langle E :: L,\ (choice\ (\psi_1\ Q) :: nil \quad (\psi_2\ R) :: nil) :: C,\ false :: S\rangle \ \Rightarrow\ \langle E :: L,\ R :: C,\ S\rangle \end{array}$$

We can simplify these rules by combining some constants: since *choice* expressions are always of the form $(choice\ (\psi_1\ Q) :: nil\ \ (\psi_2\ R) :: nil)$, we can introduce the "abbreviated" form $(choice'\ Q\ R)$, modifying the rules appropriately.

6.4. Pairs

We can introduce strict pairs to our object-language by introducing a new signature item $pair : tm \rightarrow tm \rightarrow tm$ for constructing pairs of terms. In the highest level evaluator, the inference rule for evaluating a pair is

$$\frac{eval\ P_1\ V_1 \qquad eval\ P_2\ V_2}{eval\ (pair\ P_1\ P_2)\ (pair\ V_1\ V_2)}.$$

In the first-order setting using closures, the rule for evaluating a pair is

$$\frac{eval\ \{E, P_1\}\ V_1 \qquad eval\ \{E, P_2\}\ V_2}{eval\ \{E, (pair\ P_1\ P_2)\}\ (pair\ V_1\ V_2)}$$

and this can be added to $\mathcal{V}_0$. (Again, we assume that the type of *pair* is now $fotm \rightarrow fotm \rightarrow fotm$.)

Now, following the derivations in Section 5.2, we eliminate branching rules, producing the rule

$$\frac{prove \quad (eval\ \{E, P_1\}\ V_1) :: (eval\ \{E, P_2\}\ V_2) :: G}{prove \quad (eval\ \{E, (pair\ P_1\ P_2)\}\ (pair\ V_1\ V_2)) :: G}.$$

To accommodate the next step, that of introducing an value stack, we first split the above rule into the following two rules:

$$\frac{prove\ (eval\ \{E, P_1\}\ V_1) :: (eval\ \{E, P_2\}\ V_2) :: (mkpair\ V_1\ V_2\ V) :: G}{prove\ (eval\ \{E, (pair\ P_1\ P_2)\}\ V) :: G}$$

$$\frac{prove\ G}{prove\ (mkpair\ V_1\ V_2\ (pair\ V_1\ V_2)) :: G}.$$

These two rules essentially split the act of producing the values V_1 and V_2 from the act of constructing the pair of these two values. Using these two rules we can introduce a value stack, producing the following new rules:

$$\frac{prove_1\ \ (eval_1\ \ \{E, P_1\}) :: (eval_1\ \ \{E, P_2\}) :: \phi_3 :: C\ \ S\ \ Z}{prove_1\ \ (eval_1\ \ \{E, (pair\ P_1\ P_2)\}\ ) :: C\ \ S\ \ Z}$$

$$\frac{prove_1\ \ (mkpair\ V_1\ V_2) :: C\ \ S\ \ Z}{prove_1\ \ \phi_3 :: C\ V_2 :: V_1 :: S\ Z}$$

$$\frac{prove_1\ \ C\ (pair\ V_1\ V_1) :: S\ Z}{prove_1\ \ (mkpair\ V_1\ V_2\ ) :: C\ \ S\ \ Z}.$$

Folding the latter two rules yields

$$\frac{prove_1\ \ C\ (pair\ V_1\ V_1) :: S\ Z}{prove_1\ \ \phi_3 :: C\ V_2 :: V_1 :: S\ Z}.$$

The remaining transformation steps are straightforward, producing the following rules for the CLS machine:

$$\begin{array}{lcl} \langle (pair\ P_1\ P_2) :: C,\quad E :: L,\quad S\rangle & \Rightarrow & \langle P_1 :: P_2 :: \psi_3 :: C,\quad E :: E :: L,\quad S\rangle \\ \langle \psi_3 :: C,\quad L,\quad V_2 :: V_1 :: S\rangle & \Rightarrow & \langle C,\quad L,\quad (pair\ V_1\ V_2) :: S\rangle. \end{array}$$

(The switch from $\phi_3 : o$ to $\psi_3 : fotm$ reflects the change in type of the instruction list C.)

6.5. Primitive Constants and Constructors

We can introduce constants and data constructors in a variety of ways. We present a method in which they are treated similarly. All constants are assumed to be in normal form, and so they evaluate to themselves. Data constructors are assumed strict and their arguments are encoded as pairs.

For constants, we introduce the new signature item $const : string \rightarrow fotm$, and we represent each constant `c` of our object-language by the term $(const\ c)$. Then we specify evaluation by the single rule

$$eval\ (const\ c)\ (const\ c)$$

and this translates to the machine rule

$$\langle (const\ c) :: C,\ E :: L,\ S\rangle \Rightarrow \langle C,\ L,\ (const\ c) :: S\rangle.$$

For data constructors, we introduce the new signature item $constr$, and we represent each constructor `c(t1,...,tn)` of our object-language by the term $(constr\ c\ args)$ in which $args$ is the encoding of the tuple $(t_1, \ldots, t_n)$. Then we specify evaluation by the single rule

$$\frac{eval\ A\ B}{eval\ (constr\ K\ A)\ (constr\ K\ B)}$$

and this translates (via a process similar to that for pairs) to the machine rules

$$\begin{array}{lcl}\langle (constr\ K\ A) :: C,\ E :: L,\ S\rangle & \Rightarrow & \langle A :: (mkconstr\ K) :: C,\ L,\ S\rangle \\ \langle (mkconstr\ K) :: C,\ E :: L,\ B :: S\rangle & \Rightarrow & \langle C,\ L,\ (constr\ K\ B) :: S\rangle.\end{array}$$

6.6. Primitive Functions

Primitive functions can also be added in a straightforward way. We first assume that all primitive functions are strict and unary. In our first-order abstract syntax we encode primitive operators by introducing the constructor *prim* : *string* → *fotm* in which the operation is denoted by a string name. The evaluation of primitive functions can be specified via the rule

$$\frac{eval\ \{L, M\}\ (prim\ Op) \qquad eval\ \{L, N\}\ R \qquad oper(Op, Q) = V}{eval\ \{L, M\,\hat{}\,N\}\ V},$$

where $oper(Op, Q) = V$ denotes the property that applying the primitive function Op to the argument Q yields result V.

Now, following the derivation of Section 5.2, we can produce the following two rules:

$$\begin{array}{lcl}\langle (M\,\hat{}\,N) :: C,\ E :: L,\ S\rangle & \Rightarrow & \langle M :: N :: \psi_4 :: C,\ E :: E :: L, S\rangle \\ \langle \psi_4 :: C,\ L,\ R :: (prim\ Op) :: S\rangle & \Rightarrow & \langle C,\ L,\ oper(Op, R) :: S\rangle.\end{array}$$

Adding these two rules to our CLS machine produces a problem similar to the one encountered with the conditional statement. The machine has two possible reduction steps for applications: one for the case of primitive function application (introducing the constant ψ_4) and one for application of a lambda abstraction (introducing the constant ψ'). By factoring these two rules using the transformation described in Appendix B, performing some unfolding and renaming some collection of constants, we can simplify the description of application to the rules below (for a new constant ψ_5 : *fotm*).

$$\begin{array}{lcl}\langle (M\,\hat{}\,N) :: C,\ E :: L,\ S\rangle & \Rightarrow & \langle M :: N :: \psi_5 :: C,\ E :: E :: L,\ S\rangle \\ \langle \psi_5 :: C,\ L,\ R :: \{E', M\} :: S\rangle & \Rightarrow & \langle M :: C,\ (R :: E') :: L,\ S\rangle \\ \langle \psi_5 :: C,\ L,\ R :: (prim\ Op) :: S\rangle & \Rightarrow & \langle C,\ L,\ oper(Op, R) :: S\rangle.\end{array}$$

7. Related Work

Previous work using abstract machines as an intermediate-level representation for a language has relied on informal methods for defining the machines or for relating them to source languages, or both. The principal advantage of this approach is the flexibility afforded in designing the structure of machines, as there are no a priori restrictions on the data structures or control structures that can be used. Considerations of efficiency, not evident in the semantic specifications, can be accommodated. The principal disadvantage, however, is the potential difficulty of proving that the abstract machines correctly implement a given language. Another disadvantage is the potential difficulty in extending the machines to accommodate new language features.

The prototypical abstract machine for a functional programming language is the SECD machine, and a proof of its correctness can be found in [Plo76]. While demonstrating the correctness of this machine with respect to an operational semantics, this proof does not provide much intuition regarding the overall structure of the machine or how it can be extended to handle a more substantial language. Our methods, however, provide a basis for proving such machines correct and, as demonstrated in Section 6, provide support for extending the machines to accommodate additional language features.

An example of an abstract machine used in an actual implementation of a language is the Functional Abstract Machine [Car84], a variant of the SECD machine, which has been optimized to allow fast function application. This machine, designed for implementing the ML language, addresses issues including efficient local variable storage, efficient closure construction and function manipulation, and exception handling. The design of this machine was influenced both by the structure of ML, for which it provides an intermediate level of representation, and also by concerns for efficient handling of data structures. This machine would not naturally be constructed by the naive strategy we have presented without a better understanding of architectural concerns. We believe, however, that the techniques we have developed could be adapted to prove the correctness of the FAM relative to an operational semantics for the core subset of ML that it implements.

Our use of operational semantics as the basis for language definitions supports a simple means for describing both semantics and machines in a single framework, namely proof systems. This uniformity contributes to the simple construction and verification of machines from semantics. In addition, operational semantics provides a natural setting for explicitly separating static and dynamic semantics. An alternative approach to language definition and implementation uses denotational semantics to define languages but also uses abstract machines as an implementation mechanism. One example of this approach uses a *two-level meta-language* to distinguish between compile-time and run-time operations of a denotational semantics and translates the run-time operations into an intermediate-level machine code [NN88]. The abstract machine and compiler defined in this way, though strongly influenced by denotational semantics, are given informally, and a correctness proof is still required to relate the denotational semantics, target language semantics, and compiler. Unfortunately, this proof requires some complex mathematical machinery (e.g., Kripke-like relations) to define a relation between the denotations of programs and the actions performed by abstract machines. This complexity seems inherent in this approach to compiler correctness due to fundamental differences between denotational definitions and abstract machine definitions.

The distinction between compile-time and run-time operations provided by this two-level meta-language corresponds partly to the separation of operational semantics into distinct static and dynamic semantics specifications. In our work we have started with a dynamic semantics and not attempted to make any further distinction in the semantics regarding stages of operation. Thus our abstract machines operate on the same language, though perhaps in a different syntax, as the operational semantics. Additional separation of these abstract machines into compile-time and run-time operations, producing a level of machine code similar to that produced in the above approach, can be found in [Han91].

An alternative approach to constructing abstract machines from semantics is explored in [Wan82] where a continuation-style denotational semantics is used instead of operational semantics. The proposed technique involves choosing special-purpose combinators to eliminate λ-bound variables in the semantics and discovering standard forms for the resulting terms, to obtain target code that represents the meaning of the source code. The resulting abstract machine interprets these combinators. This approach relies on heuristics to choose appropriate combinators and discover properties of these combinators. Verification of the abstract machines constructed using this method is not addressed.

Finally, all the proof systems used in this paper can be naturally represented in the dependent type system LF [HHP87]. It is then possible to use the Elf implementation [Pfe91] of LF to implement the various transformations we have presented. Initial work in this direction is reported in [HP92].

8. Summary

We presented a general method for translating the operational semantics of evaluators into abstract machines. We believe that the choice of operational semantics as a starting point for semantics-directed implementations of programming languages is advantageous as it provides both a high-level and natural specification language and, as argued here, can be translated into intermediate architectures found in real, hand-crafted compilers. Though not automatic, our method provides a flexible approach for designing representations of languages that are suitable to machine implementation.

The translation from operational semantics to abstract machine proceeds via a series of transformations on the operational semantics. These specifications are given by proof systems axiomatizing evaluation relations, and the proofs constructible in these systems provide concrete objects representing the computation steps performed during evaluation. By examining the structure of these proofs, we can identify properties (of proofs and/or the proof system) that allow us to transform the proof system and also the proofs. Because proofs are syntactic objects that can be encoded as terms in a logic, we believe that most of the manipulations described here can be performed by an automated system, providing, at least, machine checked proofs of the construction of abstract machines.

Specifications within a second-order meta-logic can leave many aspects of computation implicit: implementations of the meta-logic must provide for such features as backtracking, unification, stacking of goals, etc. The central goal of the transformations presented here has been to make some of these implicit aspects more explicit within the specification. Thus, as the transformations continue, fewer features of the meta-logic are required. In our examples, transformations are continued until the the meta-logic can be modeled using a simple, iterative rewriting system characterized here as an abstract machine.

Appendix A. Unfolding Rules

One common kind of transformation on inference rules is a simple unfolding of two rules, producing a single new rule. Let R_1 and R_2 be two inference rules of the form

$$\frac{A_1 \quad A_2 \quad \cdots \quad A_n}{A} \qquad\qquad \frac{B_1 \quad B_2 \quad \cdots \quad B_m}{B}$$

$$\frac{\begin{array}{c}\Xi_1\\ \theta(\sigma(B_1))\end{array} \cdots \begin{array}{c}\Xi_2\\ \theta(\sigma(B_{i-1}))\end{array} \quad \begin{array}{c}\Xi_3\\ \theta(\sigma(A_1))\end{array} \cdots \begin{array}{c}\Xi_4\\ \theta(\sigma(A_n))\end{array} \quad \begin{array}{c}\Xi_5\\ \theta(\sigma(B_{i+1}))\end{array} \cdots \begin{array}{c}\Xi_6\\ \theta(\sigma(B_m))\end{array}}{\theta(\sigma(B))} \qquad (*)$$

$$\frac{\begin{array}{c}\Xi_1\\ \theta(\sigma(B_1))\end{array} \cdots \begin{array}{c}\Xi_2\\ \theta(\sigma(B_{i-1}))\end{array} \quad \dfrac{\begin{array}{c}\Xi_3\\ \theta(\sigma(A_1))\end{array} \cdots \begin{array}{c}\Xi_4\\ \theta(\sigma(A_n))\end{array}}{\theta(\sigma(B_i))} \quad \begin{array}{c}\Xi_5\\ \theta(\sigma(B_{i+1}))\end{array} \cdots \begin{array}{c}\Xi_6\\ \theta(\sigma(B_m))\end{array}}{\theta(\sigma(B))} \qquad (\dagger)$$

Fig. 25. Eliminating derived rules

for $n, m \geq 1$, respectively, such that for some i, $1 \leq i \leq m$, A and B_i unify with unifier σ. Then the following rule, denoted as $\sigma(R_1/R_2)_i$, is a *derived* rule from R_1 and R_2:

$$\frac{\sigma(B_1)\cdots\sigma(B_{i-1}) \quad \sigma(A_1)\cdots\sigma(A_n) \quad \sigma(B_{i+1})\cdots\sigma(B_m)}{\sigma(B)}.$$

We call R_1 and R_2 the *unfolded* rules. We assume no clashes of the universally quantified variables occurring in these terms, as these variables can always be renamed. Note that for two given inference rules, a number of derived rules can exist based on possible choices of i and σ.

Theorem 20. Let $\mathcal{E}$ be a proof system including two rules R_1 and R_2 with $\sigma(R_1/R_2)_i$ a derived rule for some σ and i. Then for all formulas A, $\mathcal{E} \vdash A$ if and only if $\mathcal{E} \cup \{\sigma(R_1/R_2)_i\} \vdash A$.

Proof. The proof in the forward direction is trivial as any $\mathcal{E}$ proof is also a $\mathcal{E} \cup \{\sigma(R_1/R_2)_i\}$ proof. In the reverse direction we show that for any A and any $\mathcal{E} \cup \{\sigma(R_1/R_2)_i\}$ proof of A all occurrences of $\sigma(R_1/R_2)_i$ can be replaced by instances of R_1 and R_2. Assume R_1, R_2, σ, i and $\sigma(R_1/R_2)_i$ are as described above. Now let Π be any $\mathcal{E} \cup \{\sigma(R_1/R_2)_i\}$ proof containing some instance of $\sigma(R_1/R_2)_i$ and let Ξ be a subproof of Π whose last inference rule is an instance of $\sigma(R_1/R_2)_i$. This subproof must be of the form $(*)$ in Figure 25 for some substitution θ. Since $\sigma(B_i)$ equals $\sigma(A)$, we can construct another proof Ξ' of $\theta(\sigma(B))$ that contains one less instance of $\sigma(R_1/R_2)_i$ as given by $(\dagger)$ in Figure 25. Repeating this process until there are no more instances of $\sigma(R_1/R_2)_i$ yields an $\mathcal{E}$ proof. □

While it is not generally true that the inference rule $\sigma(R_1/R_2)_i$ can replace R_1 and R_2, such replacement is possible at several points in this paper. When this is possible, we need to show that in any proof using R_1 and R_2, these two rules occur together (as illustrated above) and hence can be replaced by instances of the unfolded rule (setting σ to the most general unifier, in this case).

Appendix B. Factoring Inference Rules

Call a proof system *deterministic* if for any provable proposition there exists a single (normal) proof of that proposition. Proof systems specifying the evaluation of functional programs usually are deterministic. Unfortunately, this property does not necessarily imply that the naive bottom-up construction of proofs is free from backtracking. For example, a bottom-up strategy may need to backtrack because for a given proposition, multiple inference rules may be applicable, even though only one can eventually lead to a proof.

We consider a class of proof systems for which bottom-up proof search requires backtracking, and we describe a transformation that effectively removes the need to backtrack. Our transformation is closely related to the factorization of context free production rules with common initial segments, and it has also been examined in [Sil90]. The following transformation can be used in conjunction with other transformations to remove the need for backtracking in some proof systems. For notational convenience we abbreviate $\alpha_1 :: \cdots :: \alpha_n :: C$ as $\vec{\alpha} \mathbin{@} C$.

Factoring Transformation. Let $\mathcal{E}$ be a proof system in which every non-axiom inference rule is of the form

$$\frac{\mathit{prove}\ \ \vec{\alpha'} \mathbin{@} C\ \ \delta'}{\mathit{prove}\ \ \alpha :: C\ \ \delta}$$

where C is either *nil* or a meta-variable. Furthermore, assume $\mathcal{E}$ has a pair of rules of the form

$$\left(\frac{\mathit{prove}\ \ \vec{\alpha'} \mathbin{@} \vec{\beta} \mathbin{@} C\ \ \delta'}{\mathit{prove}\ \ \alpha :: C\ \ \delta}\ ,\quad \frac{\mathit{prove}\ \ \vec{\alpha'} \mathbin{@} \vec{\gamma} \mathbin{@} C\ \ \delta'}{\mathit{prove}\ \ \alpha :: C\ \ \delta} \right)$$

for some terms α, $\vec{\alpha'}$, $\vec{\beta}$, $\vec{\gamma}$, δ and δ'. Let $\mathcal{E}^{ch}$ be a proof system that contains the following three classes of inference rules:

1 For a new, fixed constant *choice*, the two inference rules

$$\frac{\mathit{prove}\ \ C_1 \mathbin{@} C\ \ Z}{\mathit{prove}\ \ (\mathit{choice}\ C_1\ C_2) :: C\ \ Z} \quad \text{and} \quad \frac{\mathit{prove}\ \ C_2 \mathbin{@} C\ \ Z}{\mathit{prove}\ \ (\mathit{choice}\ C_1\ C_2) :: C\ \ Z}$$

are members of $\mathcal{E}^{ch}$. Here, Z, C, C_1, and C_2 are schema variables for both of these rules.

2 For each pair of rules of the form displayed above, add to $\mathcal{E}^{ch}$ the inference rule

$$\frac{\mathit{prove}\ \ \vec{\alpha'} \mathbin{@} (\mathit{choice}\ \vec{\beta}\ \vec{\gamma}) :: C\ \ \delta'}{\mathit{prove}\ \ \alpha :: C\ \ \delta}.$$

3 Finally, every inference rule of $\mathcal{E}$ not in such a pair is added to $\mathcal{E}^{ch}$.

This transformation provides a means for delaying a choice between two inference rules (during bottom-up proof construction) until after their common initial segment ($\vec{\alpha}$) has been "processed."

Theorem 21. Let $\mathcal{E}^{ch}$ be obtained from some system $\mathcal{E}$ as given by the Factoring Transformation above and let $(\mathit{prove}\ \vec{\alpha}\ \delta)$ be a closed proposition containing no occurrences of *choice*. Then $\mathcal{E} \vdash (\mathit{prove}\ \vec{\alpha}\ \delta)$ if and only if $\mathcal{E}^{ch} \vdash (\mathit{prove}\ \vec{\alpha}\ \delta)$.

The proof is by a straightforward induction on the height of proofs.

Acknowledgements. We are grateful to the reviewers of this paper and to Eva Ma for their detailed comments and suggestions for improving this paper. Both authors have been supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018 through the University of Pennsylvania. Hannan has also been supported by the Danish Natural Science Research Council under the DART project. Miller has also been supported by SERC Grant No. GR/E 78487 "The Logical Framework" and ESPRIT Basic Research Action No. 3245 "Logical Frameworks: Design Implementation and Experiment" while he was visiting the University of Edinburgh.

REFERENCES

[Bru72] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Car84] Luca Cardelli. Compiling a functional language. In *1984 Symposium on LISP and Functional Programming*, pages 208–217, ACM, 1984.

[CCM87] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cur90] Pierre-Louis Curien. *The $\lambda\rho$-calculus: An Abstract Framework for Environment Machines.* Technical Report, LIENS–CNRS, 1990.

[Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland Publishing Co., 1969.

[Gir86] Jean-Yves Girard. The system F of variable types: fifteen years later. *Theoretical Computer Science*, 45:159 – 192, 1986.

[Han90] John Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, 1990.

[Han91] John Hannan. Staging transformations for abstract machines. In P. Hudak and N. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 130–141, ACM Press, 1991.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, pages 194–204, IEEE Computer Society Press, 1987.

[HO80] Gérard Huet and D. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405, Academic Press, 1980.

[How91] Doug Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 162–172, IEEE Computer Society Press, 1991.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1992.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39, Springer-Verlag LNCS, Vol. 247, 1987.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(5):308–320, 1964.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory. Studies in Proof Theory Lecture Notes*, BIBLIOPOLIS, Napoli, 1984.

[Mar85] Per Martin-Löf. *Constructive Mathematics and Computer Programming*, pages 167–184. Prentice-Hall, 1985.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

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

[NN88] Flemming Nielson and Hanne Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56:59–133, 1988.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*, Cambridge University Press, 1991.

[Plo76] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1976.

[Plo81] Gordon Plotkin. *A Structural Approach to Operational Semantics*. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[Pra65] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.

[Sil90] Fabio da Silva. *Towards a Formal Framework for Evaluation of Operational Semantics Specifications*. LFCS Report ECS–LFCS–90–126, Edinburgh University, 1990.

[Wan82] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Trans. on Programming Languages and Systems*, 4(3):496–517, 1982.