From Operational Semantics to Abstract Machines: Preliminary Results

MS-CIS-90-21
LINC LAB 168

John Hannan
Dale Miller

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

April 1990

From Operational Semantics to Abstract Machines: preliminary results*

John Hannan and Dale Miller†

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389 USA

Abstract

The operational semantics of functional programming languages is frequently presented using inference rules within simple meta-logics. Such presentations of semantics can be high-level and perspicuous since meta-logics often handle numerous syntactic details in a declarative fashion. This is particularly true of the meta-logic we consider here, which includes simply typed λ-terms, quantification at higher types, and β-conversion. Evaluation of functional programming languages is also often presented using low-level descriptions based on abstract machines: simple term rewriting systems in which few high-level features are present. In this paper, we illustrate how a high-level description of evaluation using inference rules can be systematically transformed into a low-level abstract machine by removing dependencies on high-level features of the meta-logic until the resulting inference rules are so simple that they can be immediately identified as specifying an abstract machine. In particular, we present in detail the transformation of two inference rules specifying call-by-name evaluation of the untyped λ-calculus into the Krivine machine, a stack-based abstract machine that implements such evaluation. The initial specification uses the meta-logic's β-conversion to perform substitutions. The resulting machine uses de Bruijn numerals and closures instead of formal substitution. We also comment on a similar construction of a simplified SECD machine implementing call-by-value evaluation. This approach to abstract machine construction provides a semantics-directed method for motivating, proving correct, and extending such abstract machines.

*To appear in the Proceedings of the 1990 ACM Conference on Lisp and Functional Programming.

†Both authors have been supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018.

1 Introduction

Operational semantics provides a simple, high-level and elegant means of specifying interpreters for programming languages [13]. Abstract machines provide an intermediate representation of a language implementation that facilitates portability, code optimizations and native machine code generation [2]. In this paper we demonstrate how to take specifications of evaluators in the high-level, flexible style of operational semantics and derive, through simple and formally justifiable steps, abstract machines that implement that evaluator. While providing a direct proof of the correctness of derived abstract machines, derivations also provide guidance for extending such machines and illustrate relationships among various abstract machines.

The work in this paper can be seen as a counterpart to work in semantics-directed compiler generation based on denotational semantics and attribute grammars [11]. The goal in those settings is to provide methods for constructing provably correct, low-level implementations of programming languages starting from high-level semantic descriptions of the languages. The idea of transforming interpreters is not new [12, 16, 18]. In the setting here an interpreter is given as a set of inference rules and evaluation is described by proof construction in a fixed meta-logic. Transforming interpreters is a process of transforming the inference rules specifying an interpreter using techniques such as fold/unfold and partial evaluation [8]. Our approach is more goal directed than some of these efforts as we have abstract machines as our target implementation. These abstract machines are specified as term rewriting systems and the connection between operational semantics and abstract machines is established by defining a restricted class of operational semantic specifications that have a simple translation to term rewriting systems. Thus our entire translation from operational semantics to abstract machines can be presented using a single meta-logic.

Our meta-logic and others related to it (such as the Logical Framework (LF) [7] and the Calculus of Constructions (CC) [3]) have been investigated by several researchers recently as specification languages for functional programming systems [1, 6] and for logics [5, 17]. This logic provides a flexible domain for specifying relations on simple functional programs since such programs can be represented as terms (in several different ways) in this meta-language. In this paper we focus on various *evaluation relations*, that is, binary relations that relate a functional program with its value (under some assumption of the evaluation strategy). Sets of inference rules will be used to axiomatize such evaluation predicates.

This paper is organized as follows. In Section 2 we give a general definition of abstract machines and present the meta-logic in which we specify operational semantics. We then define a class of proof systems in that logic that correspond trivially to abstract machines. Sections 3 and 4 provide an extended example of transforming a high-level specification of call-by-name evaluation of the untyped λ-calculus into a specification of an abstract machine. Section 3 presents a translation from an operational semantics using quantification over simply typed λ-terms to one using strictly first-order quantification. Section 4 presents a translation from this first-order specification into the class of proof systems defined in Section 2. The resulting proof system is equivalent to the Krivine machine [4]. We briefly comment on how the derivation can be modified to yield other evaluators. We conclude in Section 5.

2 Abstract Machines and Operational Semantics

Abstract machines have been effectively used as intermediate or low-level architectures suitable for supporting serious implementations of a wide variety of programming languages, including imperative, functional and logic programming languages. As these classes of languages differ widely in their foundations, the machines tailored to a particular class exhibit little resemblance to machines for other classes. Therefore, we do not attempt to characterize all such machines with a single definition. We focus instead on a class of abstract machines designed for implementing λ-calculus based functional programming languages. Even here, we need to be more specific, as these implementations can be split roughly into two classes, according to their treatment of substitution [4]:

- *Graph reduction machines*, in which substitution is performed by attaching terms (graphs) to the appropriate leaves. The G-machine [10] is a sophisticated example of this class;
- *Environment machines*, in which substitution is performed by updating an environment that is kept separate from the program. The SECD machine is the archetypical example of this class.

We focus on the latter class in the remainder of this paper.

2.1 Abstract Evaluation Systems

Many of the environment-based abstract machines have common aspects and similar structures and we define here the formal notion of *Abstract Evaluation System* (AES) that captures and abstracts some of this commonality. We define an AES in terms of a term rewriting system (TRS). We assume some familiarity with rewriting systems, its terminology and the notion of computation in a rewriting system [9]. Recall that a TRS 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 Σ and free variables from X, and $\mathcal{V}(t)$ denotes the set of free variables occurring in t. We shall restrict our attention to first-order systems, i.e., Σ is a first-order signature, though this is not essential.

DEFINITION 2.1 *An* Abstract Evaluation System *is a quadruple* (Σ,R,r_0,S) *such that* $(\Sigma,R \cup \{r_0\})$ *is a term rewriting system and* $S \subseteq R$.

We are interested only in rewriting sequences that begin with r_0 and terminate with some rule from S. Furthermore, if $s \xrightarrow{r} t$ is a step in a sequence with $r \in R \cup \{r_0\}$ then $s \rightarrow t$ must be an instance of r, i.e., rewrite steps must occur at the outermost level (no embedded rewritings). Rule r_0 can be understood as "loading" the machine to an initial state given an input program and the rules S denote the successful termination of the machine and can be understood as "unloading" the machine and producing the answer or final value. Note that a computation sequence starting with r_0 may terminate in an unsuccessful state, i.e., the last step in the sequence is an instance of some rule not in S.

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

			M		$\Rightarrow$	(	nil	nil	$M::nil$	nil)
(	S	E	$(M\hat{}N)::C$	D)	$\Rightarrow$	(	S	E	$M::N::ap::C$	D)
(	S	E	$\lambda M::C$	D)	$\Rightarrow$	($\{E,\lambda M\}::S$		E	C	D)
(	S	$X::E$	$n+1::C$	D)	$\Rightarrow$	(	S	E	$n::C$	D)
(	S	$X::E$	$0::C$	D)	$\Rightarrow$	(	$X::S$	E	C	D)
($X::\{E',\lambda M\}::S$		E	$ap::C$	D)	$\Rightarrow$	(	nil	$X::E'$	$M::nil$	$(S,E,C)::D$)
(	$X::S$	E	nil	$(S',E',C')::D$)	$\Rightarrow$	(	$X::S'$	E'	C'	D)
(	$X::S$	E	nil	nil)	$\Rightarrow$		X			

FIGURE 1
The Krivine machine (top) and SECD machine

The SECD machine [14] and Krivine machine [4] are both AESs and variants of these are given in Figure 1. The syntax for λ-terms uses de Bruijn notation with $\hat{}$ (infix) and λ as the constructors for application and abstraction, respectively. The constants *nil* and :: are the machines' list constructors and $\{E,M\}$ denotes the closure of term M with environment E. The first rule given for each machine is the "load" rule or r_0 of their AES description. The last rule given for each is a single "unload" rule. 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 (E,C,S) in which E is an environment, C is a single term to be evaluated and S is a stack of arguments. A state in the SECD machine is a quadruple (S,E,C,D) 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. Although Landin's original description of the SECD machine uses variables names, our use of de Bruijn numerals does not change the essential mechanism of that machine.

In certain situations, it might be very sensible to have a more restricted definition of AES. In particular, adding the conditions that no left-hand side of a rewrite rule can have repeated variables would make implementing an AES simpler since it would not require a costly "runtime" equality check to be certain that the duplicated variables are instantiated to the same term. Also requiring an AES to be deterministic, that is, such that no two rewrite rules can be used on the same term, is also sensible, especially in the context of modelling evaluation of functional programs. We note that all the examples presented in this paper do, in fact, satisfy both of these additional restrictions.

2.2 A Meta-Logic

We briefly describe here the logic in which we shall present specifications of operational semantics as sets of inference rules. The terms of the logic are the simply typed λ-terms over given finite sets of base types and constants (particular to each application). The atomic propositions of our meta-logic will be constructed from a finite set of n-ary predicate symbols, each with a given type. In our examples below, propositions are either atomic or universally quantified atoms, say $(\forall x\, A)$ where, in this paper, x is either of first or second-order type. To manipulate such propositions the meta-logic comes equipped with inference rules for universal introduction and $\beta\eta$-convertibility. A specification of an operational semantics is a collection of inference figures whose conclusion is an atomic formula. Axioms are considered as inference rules with no premises. A proof in this language will be understood in the standard sense of proofs in natural deduction.

Collections of inference rules can often be directly implemented. The language TYPOL of the CENTAUR system [13], used to specify operational semantics similar to the style presented here, can be compiled into Prolog. The first-order Horn clauses of Prolog, however, are not strong enough to directly implement our inference rules due to the lack of simply typed terms and universal quantification. A suitable extension of Prolog, for example λProlog [15], does directly implement such inference rules. All the examples presented here have been implemented and tested in λProlog.

2.3 AES as Inference Rules

As just mentioned, techniques from logic programming can be used to implement evaluators axiomatized in this logic. Since the meta-logic (viewed as a logic programming language) incorporates many features, such as $\beta\eta$-conversion, backtracking, recursion, unification, and meta-variables, it is easy to write perspicuous specifications of evaluation. These high-level aspects of the meta-logic, however, distinguish such specifications of evaluation from lower-level "abstract machines" in which very few of these high-level devices are available. It is possible, however, to write abstract machines with this meta-logic by making certain that the axiomatization of evaluation predicates is particularly simple; that is, they are first-order, do not have branching inference rules, and make very limited use of variables.

DEFINITION 2.2 *A set of proof rules $\mathcal{I}$ is* AES-defining *if there are two binary predicate symbols p,q (not necessarily distinct) such that*

(i) every axiom is of the form $(p\ s\ t)$ with $\mathcal{V}(t) \subseteq \mathcal{V}(s)$ for some terms s,t;

(ii) there is a distinguished inference rule of the form

$$\frac{(p\ t\ V)}{(q\ s\ V)}$$

with $\mathcal{V}(t) \subseteq \mathcal{V}(s)$ for some terms s,t and variable V not free in s or t;

(iii) every other inference rule is of the form

$$\frac{(p\ t\ V)}{(p\ s\ V)}$$

with $\mathcal{V}(t) \subseteq \mathcal{V}(s)$ for some terms s,t and variable V not free in s or t.

From a given AES-defining proof system $\mathcal{I}$ it is straightforward to construct an equivalent AES (Σ,R,r_0,S) such that for all terms s,t, $\mathcal{I} \vdash (q\ s\ t)$ if and only if $s \xrightarrow{r_0} s' \xrightarrow{*} s'' \xrightarrow{r} t$ for some $r \in S$ and some terms s',s''. The arrow $\xrightarrow{*}$ denotes zero or more rewritings using R-rules.

Not surprisingly, most formalizations of operational semantics are not AES-defining proof systems. They can fail to satisfy the requirements of AES-defining proof systems in a number of ways. For example:

(i) they contain inference rules with multiple formulas in their premises;

(ii) they contain inference rules of the form

$$\frac{(r'\ t\ V)}{(r\ s\ V)}$$

in which either r or r' is distinct from p;

(iii) they contain rules which have variables in the premise that do not also occur in the consequent; or

(iv) they contain inference rules that are not over a binary predicate or of the form

$$\frac{(p\ t\ y)}{(p\ s\ z)}$$

in which y and z are not the same variable.

We can easily rectify the former two cases via general methods, i.e., we can transform proof systems satisfying cases (i) or (ii) into equivalent proof systems in which these cases have been eliminated. The latter two cases, however, are more problematic and no single method appears universally applicable. The goal of the remainder of this paper is to demonstrate how a series of non-AES-defining proof systems that specify functional evaluation can be transformed into equivalent AES-defining ones.

3 Relating Substitutions and Closures

Untyped λ-terms can be represented using the two constants, $abs : (i \to i) \to i$ and $app : i \to i \to i$, that can be thought of as denoting the familiar functions that relate a space (here, the type i) with its function space. For example, the I combinator, $\lambda x.x$, is represented at the meta-level as the typed term $(abs\ \lambda x.x)$ while the S combinator, $\lambda x \lambda y \lambda z((xz)(yz))$ is represented at the meta-level as the typed term

$$(abs\ \lambda x(abs\ \lambda y(abs\ \lambda z(app\ (app\ x\ z)\ (app\ y\ z))))).$$

Object-level applications are encoded using the meta-level constant *app* and object-level abstraction is encoded using the meta-level constant *abs* along with a use of meta-level abstraction.

The two inference rules below provide a very immediate specification of call-by-name evaluation in which values are λ-terms in weak-head normal form.

$$eval\ (abs\ M)\ (abs\ M) \qquad \frac{eval\ M\ (abs\ R) \qquad eval\ (R\ N)\ V}{eval\ (app\ M\ N)\ V}$$

(Capital letters denote meta-level variables.) Both inference rules contain variables of the functional type $i \to i$. If we call this collection of rules $\mathcal{E}$, then we have $\mathcal{E} \vdash (eval\ s\ t)$ iff t encodes the weak-head normal form (whnf) of the untyped λ-term encoded by s.

These inference rules are a very high-level specification of evaluation in several senses. In this section we shall focus on two of these aspects, namely, the use of meta-level β-conversion to automatically do substitution of terms for bound variables and the use of λ-terms instead of simpler first-order terms in the meta-language. To prepare for a change in the representation of object-level untyped λ-terms, we introduce the two predicates *isabs* and *isapp* that determine whether or not their arguments are object-level abstractions or application. Explicitly naming these operations is valuable since their function is performed differently once closures and deBruijn's notation are introduced. This change can then be isolated to these predicates. We shall also isolate the β-redex in the second inference rule with an explicit use of an *apply* predicate. This is useful since we need to remove this β-redex. The resulting inference system is

$$\frac{isapp\ A\ M\ N \qquad eval\ M\ Q \qquad apply\ N\ Q\ P \qquad eval\ P\ V}{eval\ A\ V} \qquad \frac{isabs\ M}{eval\ M\ M}$$

$$isabs\ (abs\ M) \qquad isapp\ (app\ M\ N)\ M\ N \qquad apply\ N\ (abs\ R)\ (R\ N)$$

Clearly this proof systems proves the same *eval*-atoms as the simplier one above.

To change the representation of object-level λ-terms into first-order terms, we need to remove the occurrence of β-conversion implicit in the above proof system. We shall do this by introducing a constant $clo : i \rightarrow (i \rightarrow i) \rightarrow i$ that "names" β-redexes and permitting our encodings of terms to contain occurrences of this constant. The intended meaning of a term containing *clo* is the λ-term that results from replacing all subterms of the form $(clo\ t\ \lambda x.s)$ by $((\lambda x.s)t)$ and normalizing. By properties of substitution, it follows that the intended meanings of $(clo\ t\ (app\ s_1\ s_2))$ and $(app\ (clo\ t\ s_1)\ (clo\ t\ s_2))$ are the same, as are the intended meanings of $(clo\ t\ \lambda x(abs\ \lambda y.r))$ and $(abs\ \lambda y(clo\ t\ \lambda x.r)))$ (here, x and y are distinct variables that may be free in the term r but not free in the term t).

Terms denoting untyped λ-terms may now have the structure (in $\beta\eta$-normal form)

$$(\dagger) \qquad (clo\ t_0\ \lambda x_0(\ldots(clo\ t_n\ \lambda x_n\ s)\ldots)) \quad (n \geq 0)$$

in which s is either $(app\ s_1\ s_2)$ (for some s_1, s_2) or $(abs\ t)$ (for some t) or one of the bound variables $x_0, \ldots, x_n$. As terms are now more complex, specifying evaluation of terms requires more involved inference rules. For example, rules for *isabs* and *isapp* must now also determine whether terms of the form $(\dagger)$ denote an *abs* or *app*. Since the term s in $(\dagger)$ can be a bound variable x_i, we introduce the *isvar* predicate to test for that condition and to return the term t_i that corresponds to that bound variable. Given this motivation for introducing and treating the constant *clo*, it is an easy matter to show that the rules listed in Figure 2 prove the same *eval*-atoms as the previous set of inference rules (modulo the meaning of *clo*).

If these inference rules are read from bottom to top, the universal quantifier can be seen as introducing a new eigenvariable to play the role of the bound variable of the λ-abstraction that it instantiates. Consider, for example, proving the formula

$$isvar\ nil\ (clo\ t_0\ \lambda x_0(\ldots(clo\ t_n\ \lambda x_n\ s)\ldots))\ t.$$

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

$$isvar\ ((c_n, t_n) :: \cdots :: (c_0, t_0) :: nil)\ s'\ t$$

in which s' is the result of replacing the bound variables $x_0, \ldots, x_n$ with the (distinct) eigenvariables $c_0, \ldots, c_n$, respectively. The proof of this latter formula arises essentially from determining which eigenvariable is s' and then checking that t is the term associated with it.

The above collection of inference rules successfully avoids 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.

Let $\hat{s}$ denote the first-order term that results from representing the meta-term s using deBruijn's syntax. Writing the λ-term $(clo\ t_0\ \lambda x_0(\ldots(clo\ t_n\ \lambda x_n\ s)\ldots))$ using this first-order syntax, we obtain the term

$$\frac{isapp\ A\ M\ N \qquad eval\ M\ R \qquad apply\ N\ R\ P \qquad eval\ P\ V}{eval\ A\ V}$$

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

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

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

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

$$apply\ N\ (abs\ R)\ (clo\ N\ R) \qquad \frac{\forall x\ (apply\ N\ (R\ x)\ (S\ x))}{apply\ N\ (clo\ T\ R)\ (clo\ T\ S)}$$

FIGURE 2
Evaluation with "higher-order" closures

$(clo'\ \hat{t_0}\ (\ldots(clo'\ \hat{t_n}\ \hat{s})\ldots))$ in which a variable index i in s denotes a variable to be replaced by the term t_{n-i}. (Here, clo' treats its second argument as an abstraction by modifying offsets of variables.) Adopting the syntax for terms presented in Section 2, the above term could be represented instead simply as $\{t_n::t_{n-1}::\cdots::t_0::nil, s\}$. 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 encoded as simply $\{\lambda\lambda 0::\lambda 0::nil,\ 0\hat{}1\}$. This second 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 or 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 the term s with $\{nil, s\}$ and $\{\ell, \{t::nil, s\}\}$ with $\{t::\ell, s\}$. Given this change in the representation of terms, we can easily rewrite our inference rules to the following set of rules.

$$\frac{isabs\ M}{eval\ M\ M} \qquad \frac{isvar\ M\ T \quad eval\ T\ V}{eval\ M\ V} \qquad \frac{isapp\ A\ M\ N \quad eval\ M\ R \quad apply\ N\ R\ P \quad eval\ P\ V}{eval\ A\ V}$$

$$isabs\ \{L, \lambda M\} \qquad isapp\ \{L, M\hat{}N\}\ \{L, M\}\ \{L, N\} \qquad isvar\ \{T::L, 0\}\ T \qquad \frac{isvar\ \{L, n\}\ T}{isvar\ \{X::L, n+1\}\ T}$$

$$apply\ N\ \{L, \lambda M\}\ \{N::L, M\}$$

Assuming that we are only interested in proving *eval*-atoms, these clauses can be simplified, using fold/unfold transformations, to the inference rules below, which we call $\mathcal{E}_0$. They are further transformed in the next section.

$$eval\ \{L, \lambda M\}\ \{L, \lambda M\} \qquad \frac{eval\ \{L, M\}\ \{L', \lambda M'\} \quad eval\ \{\{L, N\}::L', M'\}\ V}{eval\ \{L, M\hat{}N\}\ V}$$

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

4 From Inference System to AES

We now need to convert the inference rules in $\mathcal{E}_0$ to an AES-defining set of inference rules to obtain an abstract machine for computing the call-by-name semantics for untyped λ-terms. The proof rules in $\mathcal{E}_0$ fail to be AES-defining for at least two reasons. First, the proof system is not "linear"; that is, the rule for application requires proofs to have a branching structure. Second, the rule for application also has two variables, M' and L', in the premise that are not in the conclusion. The former problem has a general solution in that branching inference systems can uniformly be converted into non-branching systems. This process is described immediately below. The latter problem seems to have no general solution. The bulk of this section describes a series of transformations to address that problem.

We first describe how to eliminate the branching of the inference rules. Let $\mathcal{E}$ be a set of inference rules whose premises contain only atomic formulas (that is, no universally quantified atoms). Such a set of inference rules can be replaced by inference rules in which multiple premises are added to a stack of atomic

formulas to be proved. Using this stack eliminates branching in the inference rules and also imposes the order in which atomic formula will be proved. The choice of this order is important since it makes later transformations possible (see Proposition 4.5).

DEFINITION 4.1 *Let $\mathcal{E}$ be a proof system whose premises contain only atomic formulas and let prove be some one-place predicate. Define $\mathcal{E}^*$ to be the proof system consisting of the following axiom and inference rules:*

(i) $(prove\ nil) \in \mathcal{E}^*$;

(ii) if $\dfrac{A_1 \cdots A_n}{A_0} \in \mathcal{E}$ *for* $n \geq 0$ *then* $\dfrac{prove\ A_1{::}\cdots{::}A_n{::}G}{prove\ A_0{::}G} \in \mathcal{E}^*$ *for variable* G *not free in any* A_i.

The proof systems $\mathcal{E}$ and $\mathcal{E}^*$ are related by the following proposition.

PROPOSITION 4.2 *Let $\mathcal{E}$ be a proof system whose premises contain only atomic formulas. Then for any atomic formulas $a_1, a_2, \ldots, a_n$*

$$\mathcal{E} \vdash a_i \quad for\ 1 \leq i \leq n \quad iff \quad \mathcal{E}^* \vdash prove\ a_1{::}a_2{::}\cdots{::}a_n{::}nil.$$

The proof is a straightforward induction on the structure of proofs. Notice that for atomic a we have $\mathcal{E} \vdash a$ iff $\mathcal{E}^* \vdash prove\ a{::}nil$.

Thus any proof system (involving only atomic formulas) can be converted to a proof system in which all proof trees are straight lines. In so doing, some of the non-determinism implicit in $\mathcal{E}$ is given up in $\mathcal{E}^*$. Using this transformation we can move from $\mathcal{E}_0$ into $\mathcal{E}_1$ given as:

$$prove\ nil \quad (1.1) \qquad \frac{prove\ G}{prove\ (eval\ \{L, \lambda M\}\ \{L, \lambda M\}){::}G} \quad (1.2)$$

$$\frac{prove\ (eval\ \{L,M\}\ \{L', \lambda M'\}){::}(eval\ \{\{L,N\}{::}L', M'\}\ V){::}G}{prove\ (eval\ \{L, M\hat{}N\}\ V){::}G} \quad (1.3)$$

$$\frac{prove\ (eval\ X\ V){::}G}{prove\ (eval\ \{X{::}L, 0\}\ V){::}G} \quad (1.4) \qquad \frac{prove\ (eval\ \{L,n\}\ V){::}G}{prove\ (eval\ \{X{::}L, n{+}1\}\ V){::}G} \quad (1.5)$$

LEMMA 4.3 *For all atomic formula a, $\mathcal{E}_0 \vdash a$ iff $\mathcal{E}_1 \vdash prove\ a{::}nil$.*

The proof follows immediately from Proposition 4.2.

A useful transformation on proof systems involves considering partial instantiations of an inference rule. Clearly we can always add (to a proof system) instances of a rule already in the system. Furthermore, if we can show that a given set of instances (of the same rule) is exhaustive, then we can also eliminate the original rule. For this reason and because lists are either *nil* or constructed from ::, rule (1.2) can be replaced by the two inference rules:

$$\frac{}{prove\ (eval\ \{L,\lambda M\}\ \{L,\lambda M\})::nil}\ (1.2a) \qquad \frac{prove\ A::G}{prove\ (eval\ \{L,\lambda M\}\ \{L,\lambda M\})::A::G}\ (1.2b)$$

Notice that the premise of 1.2a is always trivially provable. Note further that rules 1.2b, 1.3, 1.4 and 1.5 all have premises whose list arguments are non-*nil* and hence instances of 1.1 cannot occur immediately above instances of these rules. Thus instances of 1.1 can only occur immediately above instances of 1.2a as noted above and if we fold 1.1 and 1.2a we get the axiom

$$(prove\ (eval\ \{L,\lambda M\}\ \{L,\lambda M\})::nil)$$

and then we no longer need rule 1.1 for non-trivial proofs. Then taking rules 1.2a, 1.2b, 1.3, 1.4 and 1.5 (relabeling them as shown) yields $\mathcal{E}_2$:

$$prove\ (eval\ \{L,\lambda M\}\ \{L,\lambda M\})::nil\ \ (2.1) \qquad \frac{prove\ A::G}{prove\ (eval\ \{L,\lambda M\}\ \{L,\lambda M\})::A::G}\ (2.2)$$

$$\frac{prove\ (eval\ \{L,M\}\ \{L',\lambda M'\})::(eval\ \{\{L,N\}::L',M'\}\ V)::G}{prove\ (eval\ \{L,M\widehat{}N\}\ V)::G}\ (2.3)$$

$$\frac{prove\ (eval\ X\ V)::G}{prove\ (eval\ \{X::L,0\}\ V)::G}\ (2.4) \qquad \frac{prove\ (eval\ \{L,n\}\ V)::G}{prove\ (eval\ \{X::L,n+1\}\ V)::G}\ (2.5)$$

LEMMA 4.4 *For all* s,t, $\mathcal{E}_1 \vdash (prove\ (eval\ s\ t)::nil)$ *iff* $\mathcal{E}_2 \vdash (prove\ (eval\ s\ t)::nil)$.

Note that $\mathcal{E}_1$ and $\mathcal{E}_2$ are not precisely equivalent since $\mathcal{E}_1 \vdash (prove\ nil)$ but $\mathcal{E}_2 \nvdash (prove\ nil)$. However we are not interested in this case: we are only interested in atomic *prove* statements that contain a non-empty stack.

The premise of rule (2.3) contains two occurrences of each of the two meta-level variables L' and M' which do not occur in the conclusion. This use of meta-level variables must be removed to approximate more closely an AES-defining proof system. There is some redundancy of information in a stack of provable *eval*-atoms and by exploiting this fact we can eliminate one set of occurrences of L' and M'. The following proposition explicitly describes this redundancy.

PROPOSITION 4.5 *Let* Π *be an* $\mathcal{E}_2$ *proof of* (*prove* a::*nil*) *and let* (*prove* $a_1::\cdots::a_n::nil$), *for* $n \geq 2$, *be a node in this proof. 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. Let *prove* $a_1::\cdots::a_n::nil$ be the node 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 (2.3) since the conclusion of any other inference rule applied to that formula would also not have the desired form. If the inference rule was (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 (2.2) in an $\mathcal{E}_2$ proof of *prove* a::*nil* is also an instance of

$$\frac{\textit{prove}\ (\textit{eval}\ \{\{L_1,M_1\}::L,M\}\ V)::G}{\textit{prove}\ (\textit{eval}\ \{L,\lambda M\}\ \{L,\lambda M\})::(\textit{eval}\ \{\{L_1,M_1\}::L,M\}\ V)::G}.$$

This inference rule could therefore replace (2.2). The structural information of Proposition 4.5 can be used in a more interesting fashion: we can now modify $\mathcal{E}_2$ to get the proof system $\mathcal{E}_3$ so that atomic formulas of the form

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

in $\mathcal{E}_2$ are replaced by nodes of the form

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

in $\mathcal{E}_3$ proofs. Here, the new constant *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 (*eval'* s t) occurring in the context

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

states that $\{s::\ell,t'\}$ evaluates to t. The proof system $\mathcal{E}_3$ is presented below.

$$prove\ (eval\ \{L,\lambda M\}\ \{L,\lambda M\})::nil \quad (3.1)$$

$$\frac{prove\ (eval\ \{\{L_1,M_1\}::L,M\}\ V)::G}{prove\ (eval\ \{L,\lambda M\}\ \{L,\lambda M\})::(eval'\ \{L_1,M_1\}\ V)::G} \quad (3.2)$$

$$\frac{prove\ (eval\ \{L,M\}\ \{L',\lambda M'\})::(eval'\ \{L,N\}\ V)::G}{prove\ (eval\ \{L,M\hat{}N\}\ V)::G} \quad (3.3)$$

$$\frac{prove\ (eval\ X\ V)::G}{prove\ (eval\ \{X::L,0\}\ V)::G} \quad (3.4) \qquad \frac{prove\ (eval\ \{L,n\}\ V)::G}{prove\ (eval\ \{X::L,n+1\}\ V)::G} \quad (3.5)$$

LEMMA 4.6 *For all s,t, $\mathcal{E}_2 \vdash (prove\ (eval\ s\ t)::nil)$ iff $\mathcal{E}_3 \vdash (prove\ (eval\ s\ t)::nil)$.*

The correctness of $\mathcal{E}_3$ follows from Proposition 4.5 and the discussion above.

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'* that takes a pair of lists. The explicit pairing constructors can be eliminated in this way. This is done in the proof system $\mathcal{E}_4$:

$$prove'\ \{L,\lambda M\}::nil\ \{L,\lambda M\}::nil \quad (4.1)$$

$$\frac{prove'\ \{\{L_1,M_1\}::L,M\}::S\ V::T}{prove'\ \{L,\lambda M\}::\{L_1,M_1\}::S\ \{L,\lambda M\}::V::T} \quad (4.2)$$

$$\frac{prove'\ \{L,M\}::\{L,N\}::S\ \{L',\lambda M'\}::V::T}{prove'\ \{L,M\hat{}N\}::S\ V::T} \quad (4.3)$$

$$\frac{prove'\ X::S\ V::T}{prove'\ \{X::L,0\}::S\ V::T} \quad (4.4) \qquad \frac{prove'\ \{L,n\}::S\ V::T}{prove'\ \{X::L,n+1\}::S\ V::T} \quad (4.5)$$

Given the motivation for $\mathcal{E}_4$ above and the fact that the syntactic distinction between the constructors *eval* and *eval'* is not relevant, the following is immediate.

LEMMA 4.7 *For all* s,t, $\mathcal{E}_3 \vdash (prove\ (eval\ s\ t)::nil)$ *iff* $\mathcal{E}_4 \vdash (prove'\ s::nil\ t::nil)$.

Recall that an AES-defining system must have an "output variable" as part of the atomic 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 provide the following trivial proposition:

PROPOSITION 4.8 *For any* s,t *if* Π *is an* $\mathcal{E}_4$*-proof of* $(prove'\ s::nil\ t::nil)$ *then the leaf node of* Π *is* $(prove'\ t::nil\ t::nil)$.

PROOF. We prove the following slightly stronger statement that implies the proposition: If Π is an $\mathcal{E}_4$-proof of $(prove'\ s::nil\ t::nil)$ then every node of Π is an instance of $(prove'Y\ V_1::\cdots::V_n::t::nil)$ for some $n \geq 0$.

Assume the statement does not hold and let $(prove'\ s_1::\cdots::nil\ t_1::\cdots::t_n::t'::nil)$ be the node in Π closest to the root that does not have the desired form. Since the root node of Π has the desired form, this formula must be the premise of some inference rule (4.2 - 4.5). But upon inspection of each of these rules it follows that the conclusion of this rule must also not have the desired form, contradicting the assumption. □

We now introduce a third argument to *prove'*, modifying the one $\mathcal{E}_4$ 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 $\mathcal{E}_5$, replaces *prove'* with the new predicate *prove''* of three arguments.

$$prove''\ \ \{L,\lambda M\}::nil\ \ \ \{L,\lambda M\}::nil\ \ \ \{L,\lambda M\} \quad (5.1)$$

$$\frac{prove''\ \ \{\{L_1,M_1\}::L,M\}::S\ \ \ V::T\ \ \ Z}{prove''\ \ \{L,\lambda M\}::\{L_1,M_1\}::S\ \ \ \{L,\lambda M\}::V::T\ \ Z} \quad (5.2)$$

$$\frac{prove''\ \ \{L,M\}::\{L,N\}::S\ \ \ \{L',\lambda M'\}::V::T\ \ Z}{prove''\ \ \{L,M\hat{}N\}::S\ \ \ V::T\ \ \ Z} \quad (5.3)$$

$$\frac{prove''\ \ X::S\ \ \ V::T\ \ \ Z}{prove''\ \ \{X::L,0\}::S\ \ \ V::T\ \ \ Z} \quad (5.4) \qquad \frac{prove''\ \ \{L,n\}::S\ \ \ V::T\ \ \ Z}{prove''\ \ \{X::L,n+1\}::S\ \ \ V::T\ \ \ Z} \quad (5.5)$$

LEMMA 4.9 *For all s,t, $\mathcal{E}_4 \vdash (prove'\ s{::}nil\ t{::}nil)$ iff $\mathcal{E}_5 \vdash (prove''\ s{::}nil\ t{::}nil\ t)$.*

The proof follows immediately from Proposition 4.8.

Now, $\mathcal{E}_5$ is not an AES-defining system only because in rule 5.3 the variables L' and M' occur in the premise but not in the conclusion. Consider simplifying the inference rules of $\mathcal{E}_5$ by replacing each inference rule of the form

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

where $prove'''$ is a new binary predicate. This kind of generalization is complementary to the instantiation transformations that we applied earlier. Let this resulting system be $\mathcal{E}_6$:

$prove'''\ \{L,\lambda M\}{::}nil\ \{L,\lambda M\}$ (6.1)

$$\frac{prove'''\ \{\{L',N\}{::}L,M\}{::}S\ Z}{prove'''\ \{L,\lambda M\}{::}\{L',N\}{::}S\ Z} \quad (6.2)$$

$$\frac{prove'''\ \{L,M\}{::}\{L,N\}{::}S\ Z}{prove'''\ \{L,M\hat{}N\}{::}S\ Z} \quad (6.3)$$

$$\frac{prove'''\ X{::}S\ Z}{prove'''\ \{X{::}L,0\}{::}S\ Z} \quad (6.4) \qquad \frac{prove'''\ \{L,n\}{::}S\ Z}{prove'''\ \{X{::}L,n{+}1\}{::}S\ Z} \quad (6.5)$$

Observe that by eliminating the second argument of $prove''$ we do not introduce any inference rules with variables that occur only in the premise. (In fact, we eliminate the one case remaining in $\mathcal{E}_5$.) Furthermore, observe that in the atomic formulas occurring in the premise of each $\mathcal{E}_5$ inference rule, every variable occurring in the second argument of $prove''$ occurs 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 $\mathcal{E}_5$-provable formula the first and second arguments are always lists of the same length and the second argument is always of the form $\{\ell_1,\lambda s_1\}{::}\{\ell_2,\lambda s_2\}{::}\cdots{::}nil$. Then for any $\mathcal{E}_5$-provable formula if the first and third arguments of this formula match the corresponding schema arguments in the premise of an inference rule, then the second argument must also match its corresponding schema in the premise of the rule.

LEMMA 4.10 *For all s,t, $\mathcal{E}_5 \vdash (prove''\ s{::}nil\ t{::}nil\ t)$ iff $\mathcal{E}_6 \vdash (prove'''\ s{::}nil\ t)$.*

PROOF. The proof in the forward direction is immediate: given a proof in $\mathcal{E}_5$ simply delete the second argument of all atomic formulas and change $prove''$ to $prove'''$. So we just need to show the reverse direction.

We prove the following, slightly more general, statement: For all ℓ_1, v, if $\mathcal{E}_6 \vdash (\mathit{prove}''' \ \ell_1 \ v)$ then there exists some ℓ_2 such that $\mathcal{E}_5 \vdash (\mathit{prove}'' \ \ell_1 \ \ell_2 \ v)$. The proof proceeds by induction on the height h of an $\mathcal{E}_6$-proof Π.

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

$$(\mathit{prove}''' \ \{\ell, \lambda s\} ::\mathit{nil} \ \{\ell, \lambda s\})$$

and for $\ell_2 = \{\ell, \lambda s\}$ we can construct the corresponding $\mathcal{E}_5$ proof Π':

$$(\mathit{prove}'' \ \{\ell, \lambda s\} ::\mathit{nil} \ \{\ell, \lambda s\} ::\mathit{nil} \ \{\ell, \lambda s\}).$$

step: Assume the statement holds for proofs of height $h \geq 1$ and let Π be a proof of height $h+1$. Then it is of the form

$$\frac{\Pi_1}{\mathit{prove}''' \ \ell \ v}$$

for some ℓ, v and some proof Π_1 of height h, with root $(\mathit{prove}''' \ \ell_1 \ v)$ for some ℓ_1. By the induction hypothesis we can construct the $\mathcal{E}_5$ proof Π_1' corresponding to Π_1 such that the root of Π_1' is of the form $(\mathit{prove}'' \ \ell_1 \ \ell_2 \ v)$ for some ℓ_2. Let the last inference rule of Π be some instance of $6.i$ (for $2 \leq i \leq 5$). We must show that we can construct an $\mathcal{E}_5$ proof $(\mathit{prove}'' \ \ell \ \ell' \ v)$ (for some ℓ') by applying an instance of rule $5.i$ to the root of Π_1'. For this we must only show that ℓ_2 can match with the schema given as the second argument to prove'' in the premise of rule $5.i$. But this follows from the observations made above regarding the independence of this schema. ℓ' is then just the instantiation of the second argument in the conclusion of $5.i$. □

Now we have the main result of this section:

THEOREM 4.11 *For all s, t, $\mathcal{E}_0 \vdash (\mathit{eval} \ s \ t)$ iff $\mathcal{E}_6 \vdash (\mathit{prove}''' \ s ::\mathit{nil} \ t)$.*

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

Notice that $\mathcal{E}_6$ is simply an AES-defining proof system for the Krivine machine if we add the following "load" rule:

$$\frac{\mathit{prove}''' \ A ::\mathit{nil} \ Z}{\mathit{krivine} \ A \ Z}.$$

This version of the machine stores the three elements of the state (environment L, term M, stack S) using just the one stack $\{L, M\} ::S$, i.e., the environment and term form the top element of stack. Extending the operational semantics to include recursion and various constants is easily incorporated into the above transformations.

We can apply the techniques demonstrated in this section and the previous one to the following proof system specifying call-by-value evaluation, in which values are again λ-terms in weak-head normal form.

			M	$\Rightarrow$	(	nil	$\{nil, M\}::nil$)
(	S	$\{E,(M\hat{}N)\}::C$)		$\Rightarrow$	(	S	$\{E,M\}::\{E,N\}::ap::C$)
(	S	$\{E,\lambda M\}::C$)		$\Rightarrow$	(	$\{E,\lambda M\}::S$	C)
(	S	$\{X::E,n+1\}::C$)		$\Rightarrow$	(	S	$\{E,n\}::C$)
(	S	$\{X::E,0\}::C$)		$\Rightarrow$	(	$X::S$	C)
(	$X::\{E',\lambda M\}::S$	$ap::C$)		$\Rightarrow$	(	S	$\{X::E',M\}::C$)
(	$X::S$	nil)		$\Rightarrow$		X	

FIGURE 3
A simplified SECD Machine

$$eval\ (abs\ M)\ (abs\ M) \qquad \frac{eval\ M\ (abs\ R) \qquad eval\ N\ S \qquad eval\ (R\ S)\ V}{eval\ (app\ M\ N)\ V}$$

While several different derivations starting from these two rules are possible, we have constructed a "dumpless" SECD machine based on them, as specified by the AES in Figure 3. The constant *ap* corresponds to a nullary constructor *eval''* obtained via methods similar to the introduction of *eval'* above, but in this case all the arguments to *eval''* can be inferred from their context and so they can be eliminated. This simplified machine avoids using a dump because the terms and their associated environments are maintained explicitly as closures. (While the SECD 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 1, there is a one-to-one correspondence between the rules except for the two rules that manipulate the dump of the SECD.

5 Conclusion

Abstract machines provide an important stage in the efficient implementation of functional languages but the construction of such machines and the implementation of high-level languages in them have previously received little attention as formal operations. We have provided some initial direction for performing these tasks by establishing a connection between operational semantics as proof systems and abstract machines. While our approach does not supply fully automatic methods for constructing abstract machines from proof

systems it does provide insight into the mechanisms of such machines and direction for extending them. Additionally, our work serves to fill the gap among efforts in semantics-directed compiler generation, where previous work has focused primarily on using denotational and algebraic semantics and attribute grammars. Indeed, our work is closely related to semantic evaluation using attribute grammars and some of the classifications of attribute grammars (e.g., well-defined, non-circular, etc.) may suggest corresponding classifications of proof systems that can be transformed into AES-defining ones.

References

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

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

[3] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[4] P.-L. Curien. *The $\lambda\rho$-calculus: An Abstract Framework for Environment Machines*. Technical Report, LIENS, 1988.

[5] A. Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, 1989.

[6] J. Hannan and D. Miller. A meta-logic for functional programming. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 24, pages 453–476, MIT Press, 1989.

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

[8] L. Hascoët. Partial evaluation with inference rules. *New Generation Computing*, 6(2–3):187–209, 1988.

[9] G. 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.

[10] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1987.

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

[12] N. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In *Proceedings of the First International Conference on Rewriting Techniques*, pages 124–140, Springer-Verlag LNCS, Vol. 202, 1985.

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

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

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

[16] F. Pagan. Converting interpreters into compilers. *Software–Practice and Experience*, 18(6):509–527, 1988.

[17] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, September 1989.

[18] M. Wand. From interpreter to compiler: a representational derivation. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, pages 306–324, Springer-Verlag LNCS, Vol. 217, 1984.