Compiler Verification in LF

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

Frank Pfenning
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3890, U.S.A.
Email: fp@cs.cmu.edu

Abstract

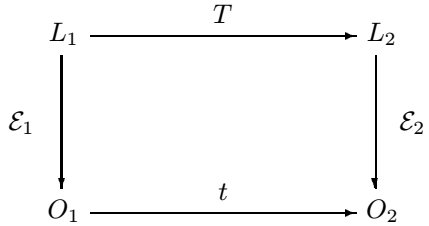
We sketch a methodology for the verification of compiler correctness based on the LF Logical Framework as realized within the Elf programming language. We have applied this technique to specify, implement, and verify a compiler from a simple functional programming language to a variant of the Categorical Abstract Machine (CAM).

1 Introduction

Compiler correctness is an essential aspect of program verification as almost all programs are compiled before being executed. Unfortunately, even for small languages and simple compilers, proving their correctness can be an enormous task, and verifying these proofs becomes an equally difficult task. Our goal is to develop techniques for mechanizing proofs of compiler correctness. To this end we employ

1. the LF Logical Framework [13] to specify relationships between source and target languages;
2. the Elf programming language [21] to provide an operational semantics for these relationships; and
3. a related meta-theory [22] to reason about the LF specifications.

The usual diagram presented in discussions of compiler correctness is:



in which L_1 and L_2 are the source and target languages; O_1 and O_2 are collections (domains) of semantic values; $\mathcal{E}_1$ and $\mathcal{E}_2$ are mappings from programs to values (semantics); T and t are the translations or compilations from L_1 to L_2 and O_1 to O_2, respectively.

To mechanically verify properties of this diagram for given semantics and program translators we must represent them formally. This task becomes significantly easier if the semantics and the translations use a similar style and can be specified in a single meta-language. Then we need only a single means for reasoning about specifications in this meta-language to verify properties of the diagram. We use operational semantics to provide meanings for source and target programs. An appealing feature of using operational semantics is the ability to specify the meanings of high-level and low-level languages in a common framework. More importantly, in *natural semantics* [14] (a form of operational semantics) *deductions* provide explicit evidence for the relation between a program and its meaning.

Given representations for the judgments found in the above diagram, we can reason about the correctness of translations as follows.

Soundness: Given any deduction of the judgments "$\mathcal{E}_1(p_1) = o_1$" and "$T(p_1) = p_2$," we must be able to construct deductions of the judgments "$\mathcal{E}_2(p_2) = o_2$" and "$t(o_1) = o_2$."

Completeness: Given any deduction of the judgments "$T(p_1) = p_2$" and "$\mathcal{E}_2(p_2) = o_2$," we must be able to construct deductions for the remaining two judgments.

Totality: Given any program p_1, there exists a p_2 such that the judgment "$T(p_1) = p_2$" holds.

1.1 Deductive Systems as a Common Framework

We use deductive systems to specify semantics, translations between programs and between values *and* relationships among these judgments.

Operational Semantics as Deductive Systems. We can specify operational semantics as deductive systems. In such semantics deductions represent computations. Compiler correctness then reduces to verifying a correspondence between deductions (*i.e.*, computations) in the two semantics. The correctness proof found in [6] for a simple compiler from λ-terms to CAM (Categorical Abstract Machine) code uses this idea. The semantics for the λ-calculus is given as a deductive system axiomatizing a relation $M \rightarrow M'$ between λ-terms, and the semantics for the CAM is given as a rewriting system specifying steps in a machine state. The informal proof of compiler correctness proceeds by showing that, for every deduction of $M \rightarrow M'$, there is a corresponding computation of machine states that reduces the compiled form of M to the compiled form of M'.

Program Translations as Deductive Systems. We can describe translations (compilations) between programs as deductive systems, axiomatizing relations between programs in source and target languages. This idea appears in [7], where the compiler from [6] is given as a deductive system; and the semantics for both source and target languages are given as deductive systems. These are all specified in Typol, a programming language supporting the definition of natural semantics [14]. The informal verification of compiler correctness proceeds by induction on the structure of the deductions. For example, given a computation deduction for source language program p_1 and a program translation deduction (for compiling p_1 to p_2) one must construct a computation deduction for target language program p_2.

Program Verifications as Deductive Systems. We go a step further by also formalizing relations between operational semantics and translations as deductive systems. This idea is suggested, though not formalized, in [7] where general inference rules based on the diagram above are presented. Unfortunately, Typol is too weak to support the direct manipulation of its own deductions, and so the specification of these relations cannot be directly implemented in Typol. We use the programming language Elf in which relations between deductions can be defined as deductive systems. These definitions provide the basis for mechanical verification of compiler correctness.

1.2 Deductions as Objects

A key aspect of this work is the ability to axiomatize relations between deductions in a natural and elegant way and then reason about these relations. This capability relies on the *deductions-as-objects* and *judgments-as-types* analogy of the LF Logical Framework, as our meta-language Elf is based on this framework. In LF each deduction (of a judgment) is represented by an object whose type is identified with that judgment. Thus deductions representing computations in an operational semantics become objects whose types identify the judgments in the semantics. We can then specify "higher-level" judgments that take as arguments objects representing deductions.

We can manipulate objects representing deductions in languages with weaker type systems, such as Typol or λProlog [16], but such specifications require explicit reasoning about the validity of such objects. For example, in these languages we could introduce new term constructors corresponding to the inference rules of an operational semantics, but this approach has two immediate shortcomings. First, the connection between the operational semantics and these proof terms cannot be internalized in the language. Second, the types of these proof terms cannot reflect the particular instances of judgments in the operational semantics. For example, objects representing computations $e_1 \hookrightarrow v_1$ and $e_2 \hookrightarrow v_2$ would have the same type. Thus it may be very difficult to verify that the proof terms are valid, since it is an external, rather than an internal condition. In LF, we instead obtain an exact correspondence between deductions of a judgment and closed, canonical, well-typed objects of the representation type. To put it yet another way: proof-checking is reduced to type-checking.

1.3 Related Work

Our work has its origins in natural semantics and its use to describe translations as outlined above [7]. We do not construct proofs of correctness that are significantly different from those found in that work. We do, however, mechanize the process of proof construction, exploiting the facilities of the Elf language to ensure partial correctness (through type constraints) and to manage the many necessary mathematical details (through term reconstruction). The idea of using deductive systems to prove properties of compilers is also found in [23] where judgments are used to verify that a compiler correctly uses the static properties of its symbol tables. Using the theorem prover Isabelle-91 [18], [3] gives a mechanized proof of compiler correctness. This work is based on using denotational semantics

for the source language of the compiler, rather than operational semantics. The relationship between code generation and denotational semantics is covered in [17], although this work does not address mechanical verification of this relationship. A mechanical verification of a code generator for an imperative language using the Boyer–Moore theorem prover is described in [24]. This verification is also based on an operational semantics, but both source and target languages are very different from what we consider here.

In [15, 22] some other examples of properties of programming languages that can be verified using LF are given. Even the most complicated proof there (type soundness for Mini-ML) is still significantly simpler than the verification of the compilation from a simple functional language, Mini-ML, to a variant of the CAM which we have completed and which is sketched in this paper.

Overview. The remainder of this paper is organized as follows. In the next section we review LF, its realization in the programming language Elf, and the basic principles for representing languages in this setting. In Section 3 we briefly discuss how language translations can be axiomatized in LF, providing a simple example. In Section 4 we describe, both informally and formally as an Elf signature, the partial verification of the compiler introduced in Section 3. In Section 5 we outline the final step required to complete the verification. In Section 6 we give an overview of some general verification techniques that we have used to verify simple compilers. Finally in Section 7 we conclude and briefly describe some future work.

2 Deductive Systems in LF

We briefly review the LF logical framework [13] as realized in Elf [19, 21, 20]. A tutorial introduction to the Elf core language can be found in [15].

The LF calculus is a three-level calculus for *objects*, *families*, and *kinds*. Families are classified by kinds, and objects are classified by *types*, that is, families of kind Type.

Kinds	K	$::=$	$\text{Type} \mid \Pi x{:}A.\ K$
Families	A	$::=$	$a \mid \Pi x{:}A_1.\ A_2 \mid \lambda x{:}A_1.\ A_2 \mid A\,M$
Objects	M	$::=$	$c \mid x \mid \lambda x{:}A.\ M \mid M_1\,M_2$

Family-level constants are denoted by a, object-level constants by c. We also use the customary abbreviation $A \to B$ and sometimes $B \leftarrow A$ for $\Pi x{:}A.\ B$ when x does not appear free in B. Similarly, $A \to K$ can stand for $\Pi x{:}A.\ K$ when x does not appear free in K. A *signature* declares the kinds of family-level constants a and types of object-level constants c.

The notion of definitional equality we consider here is based on $\beta\eta$-conversion. Type-checking remains decidable (see [5]) and it has the advantage over the original formulation with only β-conversion that every term has an equivalent canonical form.

2.1 Realization in Elf

We present the concrete syntax for Elf in the form of a grammar, where optional constituents are enclosed within $\langle\ \rangle$. The concrete syntax of the core language is very closely modeled after the abstract syntax presented earlier and is also stratified. We use *term* to refer to an entity which may be from any of the three levels. In the last column we list the corresponding cases in the definition of LF above.

kindexp	::=	`type`	Type
	|	`{`*id* $\langle$`:`*famexp*$\rangle$`}` *kindexp*	$\Pi x{:}A.\ K$
	|	*famexp* `->` *kindexp*	$A \to K$
famexp	::=	*id*	a
	|	`{`*id* $\langle$`:`*famexp*$_1\rangle$`}` *famexp*$_2$	$\Pi x{:}A_1.\ A_2$
	|	`[`*id* $\langle$`:`*famexp*$_1\rangle$`]` *famexp*$_2$	$\lambda x{:}A_1.\ A_2$
	|	*famexp objexp*	$A\,M$
	|	*famexp*$_1$ `->` *famexp*$_2$	$A_1 \to A_2$
	|	*famexp*$_2$ `<-` *famexp*$_1$	$A_1 \to A_2$
objexp	::=	*id*	c or x
	|	`[`*id* $\langle$`:`*famexp*$\rangle$`]` *objexp*	$\lambda x{:}A.\ M$
	|	*objexp*$_1$ *objexp*$_2$	$M_1\,M_2$

The terminal *id* stands either for a bound variable, a free variable, or a constant at the level of families or objects. Bound variables and constants in Elf can be arbitrary identifiers, but free variables in a declaration or query must begin with an uppercase letter. `A -> B` and `B <- A` both stand for $A \to B$. The latter is reminiscent of Prolog's "backwards" implication `:-` and improves the readability of some Elf programs. As usual, `->` is right associative, while `<-` is left associative. Juxtaposition binds tighter than the arrows and is also left associative. Parentheses are used for grouping in the obvious manner. The scope of quantifications `{`x `:` A`}` and abstractions `[`x `:` A`]` extends to the next closing parenthesis, bracket, brace, or to the end of the term. Term reconstruction fills in the omitted types in quantifications `{`x`}` and abstractions `[`x`]` and omitted types or objects indicated by an underscore `_`. In case of ambiguity a warning or error message results. For a description of Elf's term reconstruction phase see [21].

The operational semantics of Elf arises from a computational interpretation of types. This is similar in spirit to the way a computational interpretation of formulas in Horn logic gives rise to Pure Prolog. A goal is given by a type instead of a formula. Instead of searching for a proof of a formula we are searching for a (closed) object of a given type. Thus, unlike in Prolog, search will construct a representation of a proof of a goal, which can be used in subsequent computations. Instead of Prolog's first-order unification, Elf solves constraints given as dependently typed functional equations. A constant declaration (representing an inference rule) plays the role of a clause, and a signature plays the role of a program. Due to space limitations, we must refer the reader to [19, 15, 21] for further material on the operational semantics of Elf.

2.2 Abstract Syntax in Elf

For the example used throughout this paper, the source language of the compiler is an untyped λ-calculus with only abstraction and application. The operational semantics of a richer language and some of its meta-theory has been investigated in [15], and the basic ideas here extend to this richer language.

$$\textit{Expressions} \quad e \quad ::= \quad x \mid e_1\, e_2 \mid \lambda x.\, e$$

The first step in the axiomatization of a language in Elf is the description of the abstract syntax of the language. Most of the time (as is the case here) the abstract syntax is simply a type, not an indexed type family. At the highest level, the abstract syntax is *higher-order*, that is, contains meta-level λ-abstractions. The idea is to use meta-level variables to represent object-level variables. This leads to a very intuitive and succinct, yet executable specification of the operational semantics of a language [11]. At lower levels we can define abstract syntaxes for languages resembling machine code. Objects representing expressions as defined above have Elf type `exp` and are built from the constructors `app` and `lam` declared below.

```
exp : type.
app : exp -> exp -> exp.
lam : (exp -> exp) -> exp.
```

2.3 Deductive Systems in Elf

The basic unit of a deductive system is a *judgment* defined through a set of *inference rules*. In the methodology of the LF Logical Framework *judgments* are represented by *types*, and *deductions* are represented by *objects*. An inference rule is considered a constructor for deductions and thus appears as a functional constant. When following the LF methodology one engineers the representations such that deductions and well-typed (valid) objects in canonical form stand in a 1–1 correspondence.

Here we consider this paradigm using the example of an operational semantics, presented as a deductive system. Various axiomatizations of operational semantics in LF are investigated in [2, 15]. The primary judgment is $e \hookrightarrow v$ (e evaluates to v). In this paper we consider only the call-by-value semantics, though the techniques described here also apply to other operational semantics, such as call-by-name.

$$\frac{}{\lambda x.\, e \hookrightarrow \lambda x.\, e}\ \text{ev1\_lam}$$

$$\frac{e_1 \hookrightarrow \lambda x.\, e_1' \qquad e_2 \hookrightarrow v_2 \qquad [v_2/x]e_1' \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}\ \text{ev1\_app}$$

A judgment of the form $e \hookrightarrow v$ is represented as a type `eval1 E V`, where `E` is the representation of e and `V` is the representation of v. Thus `eval1` is a type family, indexed by two expressions. Objects of such a type are deductions, and inference rules are constructors for deductions.

```
eval1 : exp -> exp -> type.

ev1_lam  : eval1 (lam E) (lam E).
ev1_app  : eval1 (app E1 E2) V
             <- eval1 E1 (lam E1')
             <- eval1 E2 V2
             <- eval1 (E1' V2) V.
```

One consequence of the use of higher-order abstract syntax is that meta-level β-reduction can be used in order to model object-level substitution, as is exemplified in the third premiss above by `(E1' V2)`. Given the operational semantics of Elf, a signature such as the one above can be executed directly to solve goals of the form `?- eval1` e `V`, where e is a closed object of type `exp` and `V` is a logic variable.

Free variables in declarations are implicitly Π-quantified. For example, after term reconstruction, the internal representation of the first constructor would be

```
ev1_lam : {E:exp -> exp}
            eval1 (lam E) (lam E).
```

The arguments corresponding to such implicit quantifiers are also implicit and are synthesized during the term reconstruction phase of type-checking. For example, the deduction showing that the expression $(\lambda x.\ x)(\lambda y.\ y)$ evaluates to $(\lambda y.\ y)$ is given by the object

```
ev1_app (ev1_lam) (ev1_lam) (ev1_lam)
```

where `ev1_lam` has one implicit argument and `ev1_app` has five implicit arguments (`E1`, `E2`, `V`, `E1'`, and `V`). In this particular example, the implicit arguments are uniquely determined if we also know the type of the whole expression. In our experience, the fully explicit forms of deductions are about 5-20 times larger than their inputs, and the factor is greater for transformations between deductions. This experience supports our belief that term reconstruction is crucial to making our methodology practical.

Term reconstruction will always terminate either with failure, a most general type, or an indication that the program contains insufficient type information. The algorithm we employ has shown itself to be practical and is not a bottleneck in the implementation.

3 Language Translations in LF

Compilation, like operational semantics, can be given as a deductive system axiomatizing a relation between source and target programs. Such a presentation affords a well-structured description of the relationships between constructs in the two languages. It is similar in some respects to the structural description of operational semantics, but it has more the character of a denotational (rather than an operational) semantics.

We demonstrate a compilation from higher-order syntax to first-order syntax, considering only the simple language introduced earlier. We start by informally presenting a first-order abstract syntax for λ-terms:

$$\textit{Expressions} \quad F \quad ::= \quad 1 \mid F\uparrow \mid \Lambda F \mid F_1 F_2$$

This notation uses a slight generalization of de Bruijn notation. Here, the constructor $\uparrow$ (reminiscent of the shift operator in the $\lambda\sigma$-calculus [1]) is used to generate indices greater than 1. For example, the term $\lambda x.\ \lambda y.\ xy$ is represented by $\Lambda\Lambda((1\uparrow)1)$. The representation in LF is straightforward.

```
fexp : type.

1 : fexp.
^ : fexp -> fexp.
$ : fexp -> fexp.
@ : fexp -> fexp -> fexp.
```

We also declare (not shown here) `^` as a postfix operator, `@` as an infix operator, and `$` as a prefix operator, with `^` binding tightest and `$` weakest.

The operational semantics we define below uses environments as a means for mapping free variables to values (avoiding any explicit use of substitution) and closures for representing terms whose free variables have values provided by an environment. The following is the syntax for environments and values, which in this restricted language just consist of closures:

$$\begin{array}{llll} \textit{Environments} & L & ::= & \cdot \mid L; w \\ \textit{Values} & W & ::= & \{L, F\} \end{array}$$

Again, the representation in Elf is obvious (omitting the infix declaration for `;`).

```
env : type.
val : type.

clo   : env -> fexp -> val.
empty : env.
;     : env -> val -> env.
```

The operational semantics for this language (the target of the first phase of compilation) axiomatizes a relation between an environment L, a term F and a closure W written $L \rhd F \hookrightarrow_2 W$. This can be read as "*in environment L the term F evaluates to closure W*." Figure 1 contains the operational semantics for evaluation using the first-order syntax.

Its representation in Elf is also routine and does not employ any higher-order constructs.

```
eval2 : env -> fexp -> val -> type.

ev2_$ : eval2 L ($ F) (clo L ($ F)).
ev2_@ : eval2 L (F1 @ F2) W
         <- eval2 L F1 (clo L1 ($ F1'))
         <- eval2 L F2 W2
         <- eval2 (L1 ; W2) F1' W.
ev2_1 : eval2 (L ; W) 1 W.
ev2_^ : eval2 (L ; W') (F ^) W
         <- eval2 L F W.
```

$$\frac{}{L \rhd \Lambda F \hookrightarrow_2 \{L, \Lambda F\}}\,\text{ev2}\Lambda$$

$$\frac{L \rhd F_1 \hookrightarrow_2 \{L_1, \Lambda F_1'\} \qquad L \rhd F_2 \hookrightarrow_2 W_2 \qquad (L_1; W_2) \rhd F_1' \hookrightarrow_2 W}{L \rhd (F_1\, F_2) \hookrightarrow_2 W}\,\text{ev2@}$$

$$\frac{}{(L; W) \rhd 1 \hookrightarrow_2 W}\,\text{ev2\_1} \qquad \frac{L \rhd F \hookrightarrow_2 W}{(L; W') \rhd F\uparrow \hookrightarrow_2 W}\,\text{ev2}\uparrow$$

Figure 1: Operational Semantics using a first-order syntax

Finally we define a compilation or translation between expressions in the higher-order syntax and those in the first-order syntax. To define this translation inductively, we consider the target to be a first-order term and an environment (which would be initialized to the empty environment at the top-level). We axiomatize relations $L \rhd F \leftrightarrow e$ and $W \leftrightarrow_v e$, providing translations between expressions and values, respectively, in the two languages.

$$\frac{\dfrac{}{w \leftrightarrow_v x}\,\nu\mathcal{T} \\ \vdots \\ L; w \rhd F \leftrightarrow e}{L \rhd \Lambda F \leftrightarrow \lambda x.e}\,\text{tr}\Lambda^{\nu\mathcal{T}}$$

$$\frac{L \rhd F_1 \leftrightarrow e_1 \qquad L \rhd F_2 \leftrightarrow e_2}{L \rhd F_1\, F_2 \leftrightarrow e_1 e_2}\,\text{tr@}$$

$$\frac{W \leftrightarrow_v e}{L; W \rhd 1 \leftrightarrow e}\,\text{tr1}$$

$$\frac{L \rhd F \leftrightarrow e}{L; W \rhd F\uparrow \leftrightarrow e}\,\text{tr}\uparrow$$

$$\frac{L \rhd \Lambda F \leftrightarrow \lambda x.e}{\{L, \Lambda F\} \leftrightarrow_v \lambda x.e}\,\text{vtr}$$

The first rule exemplifies a *hypothetical judgment*: we assume that a new value w corresponds to x and then translate e in an environment extended by w. Together with the tr@ and tr$\uparrow$ rules, this achieves the correct translation of bound variables to their de Bruijn indices. Note also that in the trΛ rule, x and w must be "new," that is, they may not occur free in any other assumption of the subdeduction or the final conclusion. Thus, this rule also exemplifies a *schematic judgment*. In LF, deduction of hypothetical and schematic judgments are represented as functions: in this case a function which, given a closure W, an expression v, and a deduction of $W \leftrightarrow_v v$ returns a deduction of $L; W \rhd F \leftrightarrow [v/x]e$. In the representation of the proof of Theorem 1 we will exploit this functional representation.

Note also that in the tr1 and tr$\uparrow$ rules, the structure of the higher-order expression does not change. This means that the compilation relation is nondeterministic.

```
trans  : env -> fexp -> exp -> type.
vtrans : val -> exp -> type.

tr_$ : trans L ($ F) (lam E)
          <- {w:val}{x:exp} vtrans w x ->
                trans (L ; w) F (E x).

tr_@ : trans L (F1 @ F2) (app E1 E2)
          <- trans L F1 E1
          <- trans L F2 E2.

tr_1 : trans (L ; W) 1 E
          <- vtrans W E.

tr_^ : trans (L ; W) (F ^) E
          <- trans L F E.

vtr : vtrans (clo L ($ F)) (lam E)
          <- trans L ($ F) (lam E).
```

4 Compiler Verification

Following the specification of the compiler in the previous section, we now tackle the task of verifying its correctness. We begin with an informal statement of

the two main theorems which verify the compiler together with the totality of the compilation relation. In the theorems below and in the remainder of this paper we will abbreviate "*J is derivable*" by "*J*." Furthermore, we write $\mathcal{D} :: J$ when $\mathcal{D}$ is a deduction of the judgment J.

Theorem 1 (Soundness) *For any e, v, L, and F, if $e \hookrightarrow v$ and $L \rhd F \leftrightarrow e$ then there exists a W such that $L \rhd F \hookrightarrow_2 W$ and $W \leftrightarrow_v v$.*

Theorem 2 (Completeness) *For any L, F, W and e, if $L \rhd F \hookrightarrow_2 W$ and $L \rhd F \leftrightarrow e$ then there exists a v such that $e \hookrightarrow v$ and $W \leftrightarrow_v v$.*

Before we go into the informal proof of the soundness theorem of this first phase of the compiler, we review the general idea behind the representation of this proof. The informal proof will be *constructive*, *i.e.*, we will exhibit a method for constructing a W and derivations $L \rhd F \hookrightarrow_2 W$ and $W \leftrightarrow_v v$, given derivations of $e \hookrightarrow v$ and $L \rhd F \leftrightarrow e$. The associated function between derivations is not representable in LF (since it is primitive recursive, but obviously not schematic). Instead, we will represent it as a *relation* between deductions (which is given as a type family). That is, the LF representation of the (meta-theoretic) proof will be as a type family

```
map : trans L F E -> eval1 E V
        -> eval2 L F W -> vtrans V W
        -> type.
```

Each case in the induction proof analyzes the last inference in the deductions of $L \rhd F \leftrightarrow e$ and/or $e \hookrightarrow v$ and constructs appropriate deductions of $L \rhd F \hookrightarrow_2 W$ and $W \leftrightarrow_v v$. An appeal to the induction hypothesis manifests itself as a recursive call to `map`.

In some cases, we can draw conclusions about the last inference in a deduction or some of its constituents from the form of the judgment. We generically refer to this kind of reasoning step as *inversion* in the proof below.

Proof of Theorem 1. The proof is by induction on the structures of $\mathcal{P} :: e \hookrightarrow v$ and $\mathcal{T} :: L \rhd F \leftrightarrow e$ (either $\mathcal{P}$ is decreased or $\mathcal{P}$ remains fixed and $\mathcal{T}$ is decreased).

$$\textit{Case: } \mathcal{T} = \frac{\begin{array}{c}\mathcal{VT} \\ W \leftrightarrow_v e\end{array}}{L; W \rhd 1 \leftrightarrow e}\,\text{tr1}.$$

Assume that $\mathcal{P} :: e \hookrightarrow v$. By inversion on $\mathcal{VT}$, $e = v = \lambda x.\, e'$ for some e'. Hence $\mathcal{P}$ must be ev1_lam. By definition we have ev2_1 $:: L; W \rhd 1 \hookrightarrow_2 W$, and since $e = v$ we have $\mathcal{VT} :: W \leftrightarrow_v v$.

The computational content of this case is captured by the following rule.

```
mp_1 : map (tr_1 VT) ev1_lam ev2_1 VT.
```

$$\textit{Case: } \mathcal{T} = \frac{\begin{array}{c}\mathcal{TR} \\ L \rhd F \leftrightarrow e\end{array}}{L; W' \rhd F\uparrow \leftrightarrow e}\,\text{tr}\uparrow.$$

Again we assume $\mathcal{P} :: e \hookrightarrow v$. By the induction hypothesis on $\mathcal{TR}$ we know that there exists a W, a $\mathcal{Q} :: L \rhd F \hookrightarrow_2 W$, and a $\mathcal{VT} :: W \leftrightarrow_v v$. Applying ev2$\uparrow$ to $\mathcal{Q}$ yields a deduction of $L; W' \rhd F\uparrow \hookrightarrow_2 W$, which is what we needed to construct.

The computational content is captured by the following rule. Note that the appeal to the induction hypothesis appears as a recursive call to `map`.

```
mp_^ : map (tr_^ TR) P  (ev2_^ Q) VT
         <- map TR P Q VT.
```

For the remaining two cases we assume that the two cases above do not also apply.

$$\textit{Case: } \mathcal{P} = \frac{}{\lambda x.\, e \hookrightarrow \lambda x.\, e}\,\text{ev1\_lam}.$$

Assume $\mathcal{TR}' :: L \rhd F \leftrightarrow \lambda x.\, e$. By inversion, $\mathcal{TR}'$ must end in trΛ and $F = \Lambda F'$ for some F' and thus ev2$\Lambda :: L \rhd F \hookrightarrow_2 \{L, \Lambda F'\}$. But applying the rule vtr to $\mathcal{TR}'$ yields a deduction of $\{L, \Lambda F'\} \leftrightarrow_v \lambda x.\, e$.

The representation of this case in Elf:

```
mp_lam : map (tr_$ TR) ev1_lam
             ev2_$ (vtr (tr_$ TR)).
```

Case: $\mathcal{P} =$

$$\frac{\begin{array}{ccc}\mathcal{P}_1 & \mathcal{P}_2 & \mathcal{P}_3 \\ e_1 \hookrightarrow \lambda x.\, e_1' & e_2 \hookrightarrow v_2 & [v_2/x]e_1' \hookrightarrow v\end{array}}{e_1\, e_2 \hookrightarrow v}\,\text{ev1\_app}.$$

In this case we appeal to the induction hypothesis on $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$. Assume $\mathcal{T} :: L \rhd F \leftrightarrow e_1\, e_2$. By inversion, assuming the last inference is neither tr$\uparrow$ nor tr1, we know there are deductions $\mathcal{T}_1 :: L \rhd F_1 \leftrightarrow e_1$ and $\mathcal{T}_2 :: L \rhd F_2 \leftrightarrow e_2$ where $F = F_1\, F_2$.

By the induction hypothesis on $\mathcal{P}_1$ and $\mathcal{T}_1$ we know that there exists a $\mathcal{Q}_1 :: L \rhd F_1 \hookrightarrow_2 w_1$ and $\mathcal{VT}_1 ::$

$w_1 \leftrightarrow_v \lambda x.\, e_1'$. Again by inversion we find that $w_1 = \{L_1, \Lambda F_1'\}$ and there exists $\mathcal{T}_1'' :: L_1 \rhd \Lambda F_1' \leftrightarrow \lambda x.\, e_1'$. By a second application of inversion we see that for any w', x and deduction $\mathcal{VT}' :: w' \leftrightarrow_v x$ there exists a deduction $\mathcal{T}_1' :: L; w' \rhd F_1' \leftrightarrow e_1'$.

By the induction hypothesis on $\mathcal{P}_2$ and $\mathcal{T}_2$ we know that there exists a $\mathcal{Q}_2 :: L \rhd F_2 \hookrightarrow_2 W_2$ and $\mathcal{VT}_2 :: W_2 \leftrightarrow v_2$.

Now we use the deduction schema $\mathcal{T}_1'$ with W_2 for w', v_2 for x and $\mathcal{VT}_2$ for $\mathcal{VT}'$ to conclude that there is a deduction $\mathcal{T}_3 :: L; W_2 \rhd F_1' \leftrightarrow [v_2/x]e_1'$.

Now we can apply the induction hypothesis to $\mathcal{P}_3$ and $\mathcal{T}_3$ to obtain $\mathcal{Q}_3 :: L; W_2 \rhd F_1' \hookrightarrow_2 W$ and $\mathcal{VT} :: W \leftrightarrow_v v$.

By an application of the rule ev2@ to premisses $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_3$ we can then conclude that $L \rhd F_1\, F_2 \hookrightarrow_2 W$.

This reasoning is captured by the following rule.

```
mp_app :
  map (tr_@ T2 T1) (ev1_app P3 P2 P1)
      (ev2_@ Q3 Q2 Q1) VT
   <- map T1 P1 Q1 (vtr (tr_$ T1'))
   <- map T2 P2 Q2 VT2
   <- map (T1' _ _ VT2) P3 Q3 VT.
```

Note that whenever the informal proof appeals to inversion, we implement this through matching against a compound term, for example, `(vtr (tr_$ T1'))` in `mp_app`. Note also that the substitution into the deduction $\mathcal{T}_1'$ is modeled by a meta-level application (which implements the substitution by β-reduction at the meta-level). The two omitted arguments are called W_2 and v_2 in the informal proof. They are not deduction objects and have therefore been elided for stylistic reasons. □

5 Schema-Checking

In the previous section we have seen how some aspects of an informal proof of compiler correctness can be represented as a higher-level judgment, that is, a judgment relating deductions. This higher-level judgment can be executed using the operational interpretation of Elf: given a translation $\mathcal{T} :: L \rhd F \leftrightarrow e$ and an evaluation $\mathcal{P} :: e \hookrightarrow v$, we can pose the query

```
?- map T P Q VT.
```

Execution of this query will bind `Q` to an evaluation $\mathcal{Q} :: L \rhd F \hookrightarrow_2 W$ and `VT` to a value translation $\mathcal{VT} :: W \leftrightarrow_v v$.

The LF type system guarantees the proper relationships between the various deductions. However, it cannot guarantee that *every* query of the form above succeeds. Here we come up against the inherent limitations of the LF type theory: only the relation `map` is representable, but not the function which takes the first two arguments of `map` and returns the latter two.

Thus the methodology for formalization and verification of one phase of the compilation presents itself as follows:

1. Represent the abstract syntax of the various languages in question (`exp`, `fexp`, `val`, `env`);
2. represent the compilation relation between source and target languages (`trans`, `vtrans`);
3. represent the verification relation between evaluations in the source and target language (`map`); and finally
4. check that the verification relation is total in some arguments.

The checking of Stage 4 must currently be done by hand. However, for certain classes of relations (analogous to the class of primitive recursive functions) this checking can be automated. This process, called *schema-checking*, is discussed in more detail in [22]. It relies on induction over the structure of closed, valid LF terms over a given signature. All the examples in this paper and their extension to Mini-ML can be verified mechanically by schema-checking.

In our experience, most problems in an attempted verification are caught already by the type reconstruction and checking phase, due to the multiple levels of dependencies in relations between deductions.

6 Verification Techniques

In the previous sections we have argued, largely by example, how compiler verification can be performed in LF. The actual proofs of correctness that we construct may not be significantly different from ones constructed by hand, but they will be much more concise (before term reconstruction) and machine checked. Since there are a wide variety of compilation techniques and strategies we cannot hope to provide a general method for axiomatizing the verification relation for an arbitrary compiler. We have, however, examined some particular methods and shown correctness results for various stages of compilers.

A particular method of compiler construction that we have examined is the one based on a series of papers describing the translation of operational semantics to

intermediate level abstract machines [12, 10]. In this work compilers are constructed from operational semantics via a three stage process. We assume that the source language is defined via an operational semantics that axiomatizes a relation between programs and values. Furthermore, we assume that this definition uses a higher-order abstract syntax to represent programs (and possibly other structures). (See [11] for a discussion of semantics specified in this style.)

The first stage is the translation from a semantics using this higher-order syntax to one using a first-order syntax, similar to the example of the previous section. The higher-order syntax affords simple descriptions of program manipulations due to its use of $\beta\eta$-equivalence as an equality relation over terms. A first-order syntax lacks such a built-in relation and so some operations on programs must be explicitly specified.

The second stage is the translation from an operational semantics using inference rules to one using a restricted set of rewrite rules. This can also be viewed as a translation from a "big-step" semantics to a "small-step" one. Verifying this translation stage poses some interesting problems. First, a big-step semantics describes the reduction of programs to values, while a small-step semantics describes the reduction of programs, via atomic actions, to other programs. Second, inference rules may contain multiple premisses and so proofs may have a tree-like structure. Rewrite rules, however, contain no premisses and rewrite sequences have a much different structure. We have previously developed translation techniques for generating rewrite rules from inference rules. For example, one of the techniques introduces a stack to keep track of the various open premisses of a proof. To verify the translation we must demonstrate that the rewrite rules use the stack in a particular way to manipulate the intermediate values produced in subproofs of the operational semantics. The complete specification of this step, continuing the example of Section 4, can be found in the appendix. One can see there how rewrite rules which are used to describe computations of an abstract machine can be implemented easily as a deductive system of a special form.

The final stage is the translation of the rewrite rules, which define an interpreter, to a compiler producing abstract machine code and a new set of rewrite rules for executing this code. This step is based on the idea of staging transformations in which the original rewrite rules are decomposed into compilation and execution stages [10]. The compilation rules translate the source expressions into an intermediate-level abstract machine code, while the execution rules provide the definition for this abstract machine language. This method stops short of generating machine code for actual hardware, but some ideas towards this goal are presented in [9].

We have used this three stage strategy to construct and verify a compiler from the language Mini-ML (a core subset of Standard ML introduced in [4]) to a variant of the Categorical Abstract Machine [6]. Proofs of this translation exist [7, 6], but to our knowledge they have not previously been mechanically verified.

Many other approaches to compiling languages can be considered in the LF setting. An initial phase of many compilers consists of performing source-to-source program transformations to produce programs from which better code can be generated. One such transformation, particularly used for functional languages, is a continuation passing style transformation which introduces continuations as a means for simplifying the flow of information during a computation. Other examples are λ-lifting and closure conversion. These kinds of transformations can also be proved correct using the techniques outlined in this paper.

A more challenging problem, yet to be addressed in this context, is the optimization or transformation of programs based on flow analyses. Traditional approaches to this task are based on denotational semantics and use abstract interpretation to provide information about the flow of data through programs. Some initial work in using deductive systems to perform such analyses can be found in [8] in which the problem of strictness analysis is addressed.

7 Conclusion and Future Work

We have demonstrated how the LF Logical Framework provides a natural setting for verifying the correctness of compilers. It provides a uniform framework for the specification of abstract syntax, operational semantics, compilation, and relationships between computations. Such LF specifications can be given an operational reading and executed in Elf (for example, to evaluate or compile programs). Moreover, some properties of such specifications (in particular those expressing the correctness of a compiler) can be checked mechanically, thereby allowing the formulation of *specification*, *execution*, and *meta-theory* of a compiler in a single meta-language.

The most pressing task is, of course, the implementation of a checker for the totality of relations, following a simple inductive schema. Currently this check still has to be done by hand. Future work includes

using theorem provers to assist in the construction of judgments relating deductions. Although we have considered only a very simple programming language, the techniques presented here should be sufficient to treat more complex languages and different evaluation strategies. We plan to investigate these in future work. We also wish to consider the verification of other programming tools besides compilers, such as partial evaluators.

Acknowledgments: The first author is supported by a grant from the Danish Natural Science Research Council under the DART project. The second author is supported in part by the U.S. Air Force under Contract F33615-90-C-1465, ARPA Order No. 7597.

Appendix

We present here the complete Elf specification of a third operational semantics for evaluating λ-terms and the proof that it is equivalent to `eval2` defined in Section 3. This operational semantics uses the same first-order syntax for expressions as before, but is based on rewrite rules (which are implemented as inference rules in Elf). The informal description of this semantics is given in Figure 2. The rules describe one-step reductions on a machine state. The state consists of a list of instructions, a list of environments and a stack of values. There are two sorts of instructions: $ev(F)$ for expressions F and ap. This set of rules defines a slight variant of the SECD machine.

To specify this semantics in Elf we begin by declaring the types of the components of the state. Then we declare the various constructors for representing machine states:

```
clsInstruction : type.
clsProgram     : type.
clsEnv         : type.
clsState       : type.

ev    : fexp -> clsInstruction.
apply : clsInstruction.

pnil  : clsProgram.
&     : clsInstruction -> clsProgram
          -> clsProgram.

enil  : clsEnv.
;;    : env -> clsEnv -> clsEnv.

st    : clsProgram -> clsEnv -> env
          -> clsState.
```

We also declare `&` and `;;` as infix operators. We reason about expressions in this language via two judgments:

```
eval3 : env -> fexp -> val -> type.
cls   : clsState -> clsState -> type.
```

`eval3` provides a "top-level" judgment that directly reflects the correspondence to `eval2`. The `cls` judgment provides the encoding of the rewrite rules of Figure 2. The complete specification of these two judgments is given in Figure 3.

Figure 4 contains the definition of `map2`, a judgment for relating `eval2` and `eval3` deductions. It is defined in terms of another judgment, `mp2` which relates an `eval2` deduction and two `cls` deductions. The second `cls` deduction represents a subdeduction of the first one and is used to verify that the value `W` of an expression `F` occurs on the top of the stack after `F` has been evaluated. Space limitations prohibits us from giving an informal description of the definition of these judgments but the rules are relatively straightforward.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[2] R. Burstall and F. Honsell. Operational semantics in a natural deduction setting. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 185–214. Cambridge University Press, 1991.

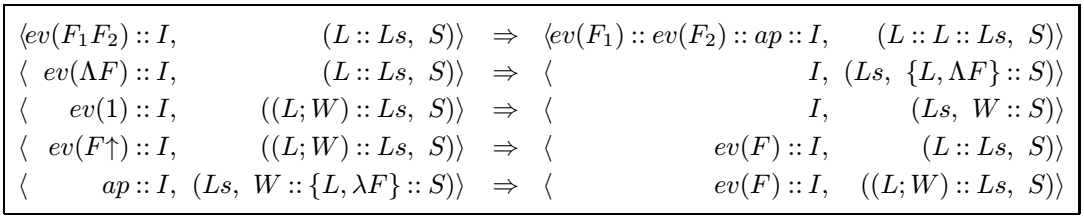
[3] B. Ciesielski and M. Wand. Using the theorem prover Isabelle-91 to verify a simple proof of compiler correctness. Unpublished Draft, 1991.

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

[5] T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

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

[7] J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 193–205, Cambridge, Massachusetts, 1986. IEEE Computer Society Press.



$$
\begin{array}{rcl}
\langle ev(F_1 F_2) :: I,\ (L :: Ls,\ S)\rangle & \Rightarrow & \langle ev(F_1) :: ev(F_2) :: ap :: I,\ (L :: L :: Ls,\ S)\rangle \\
\langle\ ev(\Lambda F) :: I,\ (L :: Ls,\ S)\rangle & \Rightarrow & \langle\ I,\ (Ls,\ \{L, \Lambda F\} :: S)\rangle \\
\langle\ \ ev(1) :: I,\ ((L; W) :: Ls,\ S)\rangle & \Rightarrow & \langle\ I,\ (Ls,\ W :: S)\rangle \\
\langle\ ev(F\uparrow) :: I,\ ((L; W) :: Ls,\ S)\rangle & \Rightarrow & \langle\ ev(F) :: I,\ (L :: Ls,\ S)\rangle \\
\langle\ ap :: I,\ (Ls,\ W :: \{L, \lambda F\} :: S)\rangle & \Rightarrow & \langle\ ev(F) :: I,\ ((L; W) :: Ls,\ S)\rangle
\end{array}
$$

Figure 2: The CLS Machine

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

[9] J. Hannan. Making abstract machines less abstract. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 618–635, Cambridge, Massachussetts, August 1991. Springer-Verlag LNCS 523.

[10] J. Hannan. Staging transformations for abstract machines. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 130–141, New Haven, Connecticut, June 1991.

[11] 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.

[12] J. Hannan and D. Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *ACM Conference on Lisp and Functional Programming*, pages 323–332, Nice, France, 1990.

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

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

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

[16] G. Nadathur and D. Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[17] F. Nielson and H. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56:59–133, 1988.

[18] L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical Report 189, Computer Laboratory, University of Cambridge, January 1990.

[19] F. Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.

[20] F. Pfenning. An implementation of the Elf core language in Standard ML. Available via ftp over the Internet, September 1991. Send mail to elf-request@cs.cmu.edu for further information.

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

[22] F. Pfenning and E. Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, Saratoga Springs, New York, June 1992. Springer-Verlag LNCS. To appear.

[23] M. Wand and Z.-Y. Wang. Conditional lambda-theories and the verification of static properties of programs. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 321–332, Philadelphia, Pennsylvania, 1990.

[24] W.D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5:493–518, 1991.

```
eval3_all : eval3 L F W
                <- cls (st ((ev F) & pnil) (L ;; enil)  empty     )
                       (st           pnil        enil  (empty ; W)).

cls_0 : cls (st pnil enil (empty ; W))
            (st pnil enil (empty ; W)).

cls_1 : cls (st ((ev (F1 @ F2))            & I) (L ;; Ls)      S) ST
     <- cls (st ((ev F1) & (ev F2) & apply & I) (L ;; L ;; Ls) S) ST.

cls_2 : cls (st ((ev ($ F)) & I) (L ;; Ls)  S                 ) ST
     <- cls (st               I        Ls  (S ; (clo L ($ F)))) ST.

cls_3 : cls (st ((ev 1) & I) ((L ; W) ;; Ls)  S     ) ST
     <- cls (st           I              Ls  (S ; W)) ST.

cls_4 : cls (st ((ev (F ^)) & I) ((L ; W) ;; Ls) S) ST
     <- cls (st ((ev F)     & I) ( L      ;; Ls) S) ST.

cls_5 : cls (st (apply  & I)             Ls  (S ; (clo L ($ F)) ; R)) ST
     <- cls (st ((ev F) & I) ((L ; R) ;; Ls)  S                     ) ST.
```

Figure 3: Operational Semantics of the CLS Machine

```
map2  : eval2 L F W -> eval3 L F W -> type.

mp2 : eval2 L F W
       -> cls (st ((ev F) & I) (L ;; Ls)  S     ) (st pnil enil (empty ; W'))
       -> cls (st           I        Ls  (S ; W)) (st pnil enil (empty ; W'))
       -> type.

map2_all : map2 Q (eval3_all R) <- mp2 Q R cls_0.

mp2_@ : mp2 (ev2_@ Q Q2 Q1) (cls_1 R) R'
               <- mp2 Q  R2 R'
               <- mp2 Q2 R1 (cls_5 R2)
               <- mp2 Q1 R  R1.

mp2_$ : mp2 ev2_$ (cls_2 R) R.

mp2_1 : mp2 ev2_1     (cls_3 R) R.
mp2_^ : mp2 (ev2_^ Q) (cls_4 R) R' <- mp2 Q R R'.
```

Figure 4: Translation between `eval2` and `eval3` deductions