Implementing the Meta-Theory of Deductive Systems

Frank Pfenning and Ekkehard Rohwedder
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890, U.S.A.
`fp+@cs.cmu.edu` and `er+@cs.cmu.edu`

Abstract. We exhibit a methodology for formulating and verifying meta-theorems about deductive systems in the Elf language, an implementation of the LF Logical Framework with an operational semantics in the spirit of logic programming. It is based on the mechanical verification of properties of transformations between deductions, which relies on type reconstruction and *schema-checking*. The latter is justified by induction principles for closed LF objects, which can be constructed over a given signature. We illustrate our technique through several examples, the most extensive of which is an interpretation of classical logic in minimal logic through a continuation-passing-style transformation on proofs.

1 Introduction

Formal deductive systems have become an important tool in computer science. They are used to specify logics, type systems, operational semantics and other aspects of languages. The role of such specifications is three-fold. Firstly, inference rules serve as a high-level notation which helps to explain the meaning of the language under consideration. This was one of Gentzen's original motivations for his calculus of natural deduction [10]. Secondly, they can form the basis for an implementation of a deductive system. For example, it is not difficult to translate an operational semantics presented in the style of natural deduction [18, 13, 4] into an implementation of an interpreter. Thirdly, deductive systems help in developing the meta-theory of a language. For example, the soundness of a type system with respect to an operational semantics is most easily expressed as a property of two inference systems.

The LF Logical Framework [16] has been designed to provide an appropriate language for the high-level specification of deductive systems. In LF, *judgments* are represented as *types* and *deductions* as *objects*. The validity of a deduction is reduced to the well-typedness of the representing object. Since type-checking in the LF type theory is decidable, purported deductions can be checked automatically for validity.

However, LF is a powerful basis for much more comprehensive tasks than mere proof-checking. Unification and proof search algorithms have been developed [7, 27, 28, 24] and it has been amenable to an operational interpretation which is realized in the Elf programming language [21, 23]. A wide range of deductive systems have been specified in LF and implemented in Elf [1, 15, 19, 20].

In this paper we investigate the use of Elf to implement the *meta-theory* of deductive systems, thus addressing the third of the principal applications listed above. Our approach is based on three observations. Firstly, in LF deductions are represented as objects and can thus be part of higher-level judgments. For example, it is easy to write down rules defining the judgment of "normal form" as it applies to natural deductions. Secondly, proofs of properties of deductive systems (henceforth called meta-proofs) often rely on transformations between deductive systems. Such transformations can be represented in LF (and implemented in Elf) as judgments relating deductions. Thirdly, due to the rich type structure of LF, it is often possible to check certain properties of such judgments purely mechanically.

Thus our methodology for the verification of meta-theorems presents itself as follows. Stage 1 (**Syntax**) is the representation of the object languages under consideration. Stage 2 (**Judgments**) is the definition of the deductive systems following the LF methodology. Stage 3 (**Deduction Transformations**) is the formulation of transformations between the deductive systems as higher-level judgments. Stage 4 (**Schema-Checking**) is the mechanical verification of a property of the transformations axiomatized in Stage 3. We have carried out this methodology for a number of examples, the most intricate of which is a verification of the subject reduction property for Mini-ML [5, 20]. Currently, Stage 4 is done mostly by hand, as we have not yet implemented a general schema-checker. As we will illustrate, in the current Elf implementation schema-checking can be directly achieved through a set of queries which can be constructed from a signature on a case-by-case basis.

It is unclear if the methodology and implementation (when it is complete) as described so far deserves the label "automated theorem prover". Clearly, it can verify many theorems far beyond the scope of current theorem provers. Moreover, a significant part of the verification is carried out automatically during term reconstruction, type-checking, and schema-checking. On the other hand, for difficult meta-theorems, the transformations constructed in Stage 3 must be carefully engineered so as to be amenable to schema-checking.

The difficulty of constructing the crucial transformations at Stage 3 varies greatly from problem to problem. In some cases —exemplified in Section 3— a routine automatic construction of the transformation from the inference systems given appears feasible as well as sufficient for complete verification. In other cases it is difficult— either because of the sheer intricacy and size of the systems involved, or because of the inherent difficulty of the meta-proof. An example of the latter is the normalization theorem for the polymorphic λ-calculus, which is beyond the scope of our techniques as we have developed them so far. We have been able to verify:

- evaluation of a Mini-ML expression results in a value if it terminates,
- equivalence of an algorithmic and a more declarative operational semantics for Mini-ML,
- type-soundness of Mini-ML (sometimes called the subject reduction theorem), including polymorphism,
- correctness of a compiler from Mini-ML to a variant of the Categorical Abstract Machine [14],

- the deduction theorem for an axiomatic formulation of propositional logic in the style of Hilbert,
- equivalence of natural deduction and Hilbert's calculus,
- soundness of two theorem provers, one using Prolog-style depth-first search and one employing bounded search,
- equivalence of two formulations of the Lambek-calculus,
- correctness of 8 different logic interpretations in the propositional calculus,
- soundness of an algorithm for deciding equality in the simply-typed λ-calculus.

In each of these cases the meta-proofs turned out to be very natural, compact, and relatively easy to construct, since they are operationally meaningful (as translations between deductions). They are also very close to an informal argument one might give to prove the corresponding meta-theorems and, with an appropriate interface, could be used to *explain* the meta-theorems and their proofs.

Schema-checking as presented in this paper verifies properties of signatures. Therefore, our work draws upon a calculus for LF signatures [17]. In that paper, it is also shown how some simple meta-theoretic properties can be witnessed directly by *realizations* (functions between signatures). The limitations of this alternative approach are also discussed, but more work is required to understand the precise relationship to schema-checking. Another line of investigation is followed by Basin and Constable [2], who propose using inductively defined types in the NuPrl type theory in order to represent deductive systems. However, their approach is not especially tailored towards developing the meta-theory of deductive systems, but applies an already existing apparatus to a new and more difficult problem. We feel that one can gain significantly by moving to a meta-language such as LF which has been specifically designed for the task of formalizing deductive systems. One can then take advantage of built-in support for such ubiquitous concepts as free and bound variables, substitution, hypothetical reasoning, or schematic judgments.

The primary difficulty in applying our methodology is to construct the transformations between deductions. Due to the strong constraints imposed by the dependent types, we believe that in many cases such transformations could be constructed automatically. Other work in inductive theorem proving and logic programming such as, for example [3, 12, 26], should be applicable in our setting to aid in the construction of such transformations. Closely related to the ideas presented here is work by Fribourg [9] in the simpler setting of Horn clauses. Again, his ideas could add to the degree of automation available within our methodology.

The remainder of this paper is organized as follows: In Section 2 we present some of the basic ideas behind Elf, schema-checking, and its connection to proof by induction, using the natural numbers as a very simple example. In Section 3 we illustrate the use of transformations between deductions and introduce the important notion of *unit refinement*. As an example we demonstrate that evaluation of an expression in a simple functional language always returns a value. In Section 4 we sketch a more comprehensive example, verifying an interpretation of classical logic in minimal logic by way of a continuation-passing-style (CPS) transformation on proofs. We end with a summary and some speculation about future directions in Section 5.

2 LF Signatures and Induction

Syntax. The first stage in the representation of a deductive system is to declare the underlying languages. We begin with a particularly simple and familiar example: the natural numbers.

```
nat : type.
z   : nat.
s   : nat -> nat.
```

`nat` is declared as a type and `z` and `s` as constructors for data of this type. Valid (that is, well-typed), closed objects of type `nat` represent natural numbers. We refer to a list of declarations such as the ones above as a *signature*. A calculus of signatures for Elf is described in [17]. As this module calculus has not yet been implemented, we only use the Elf core language in this exposition.

Judgments. The calculus of functions underlying LF is not very rich: it permits only λ-abstraction and application, and functions cannot be defined by primitive recursion, for example. This is an important requirement and cannot easily be relaxed, because it would destroy the adequacy of encodings of deductive systems in LF (see [16] and [17] for further discussion). Thus, non-trivial operations on objects must be defined as relations. Fortunately, such relations are operationally adequate within the Elf programming language, as Elf gives them an operational reading in the style of Prolog. We consider a simple `double` predicate.

```
double : nat -> nat -> type.
dbl_z  : double z z.
dbl_s  : double (s N) (s (s M)) <- double N M.
```

The relation `double` is realized as a so-called *type family* indexed by two objects which are natural numbers. For readers familiar with the Curry-Howard isomorphism between propositions and types, it should come as no great surprise that we can represent relations this way. The left-arrow is mere syntactic sugar, and `B <- A` and `A -> B` are parsed into the same internal form. The constants `dbl_z` and `dbl_s` construct objects which represent deductions. For example,

```
dbl_s (dbl_s (dbl_z)) : double (s (s z)) (s (s (s (s z)))).
```

represents a deduction which establishes that 4 is the double of 2. This object would be constructed by the Elf interpreter when executing the query

```
?- double (s (s z)) M.
```

Here `M` is a free variable which is treated as a logic variable. That is, we simultaneously search for objects `M : nat` and `P : double (s (s z)) M`.

Free variables in declarations are implicitly quantified as in Prolog. In Elf, such implicit quantifiers are inferred during parsing. The explicit form for the last declaration above would be

```
dbl_s  : {N:nat} {M:nat} double N M -> double (s N) (s (s M)).
```

`{x:A} B` is Elf's concrete syntax for $\Pi x{:}A.\ B$, where Π is the dependent function type constructor. Thus, `dbl_s` is really a function of three arguments: it accepts a natural number `N`, a natural number `M`, and then a deduction establishing that `M` is the double of `N`. It constructs a deduction showing that `(s (s M))` is the double of `(s N)`. Implicit quantifiers give rise to implicit arguments, and in its full form the proof object from the example above would be

```
(dbl_s (s z) (s (s z)) (dbl_s z z (dbl_z)))
  :  double (s (s z)) (s (s (s (s z)))).
```

The gaps in the first form shown above are filled during term reconstruction in Elf, which employs an algorithm for solving constraints between types, described in [24]. For a further discussion on term reconstruction and the operational semantics of Elf, the reader is referred to [23].

Schema-Checking — Induction. In this first example, we will directly verify a property of `double`, so there is no need to formulate a translation between deductions as they arise in Sections 3 and 4. The meta-theorem we verify states that `double` is total in its first argument.

> For every valid, closed object `n` of type `nat`, there exists a valid, closed object `M` of type `nat` and a valid, closed object `P` of type `double n M`.

Implicit in this statement is the signature from which the constants in `n`, `M`, and `P` can be drawn, which consists of all the declarations we have considered so far. Henceforth we will omit the adjective "valid" and only consider valid objects. This meta-theorem can be proven by an induction over the canonical forms of LF types and objects constructible from constants in the given signature. This induction argument cannot be internalized. That is, there are no closed objects `M` and `P` such that `P: {n:nat} double n (M n)`.

Now we are at an important crossroads. One choice is to formalize LF and build meta-theorem proving tools so that statements about signatures of the form above can be expressed and verified. This option appears prohibitively complex. The second choice is to identify general, decidable criteria for the totality of relations. A desired theorem is then verified if we can show that it is equivalent to one which satisfies such a schematic criterion.

These alternatives have a connection to the ideas behind primitive recursion. If we would like to show that a function is total, we can either reason about the function within an appropriate logic, or we can present its definition in the form of a primitive recursion. Interestingly, neither choice is *a priori* weaker than the other. For example, the functions provably total in second-order arithmetic are exactly the functions which can be defined using a schema of primitive recursion at higher types [8].

As we hope to illustrate in this paper, the latter choice is a very natural one, and many examples can be treated very elegantly. Moreover, it does not preclude the application of automated theorem proving methods, as they can be used to synthesize schematic relations. This is one of Fribourg's basic observations [9] and illustrated in Section 3. But even without any automatic assistance beyond term reconstruction it is feasible to demonstrate non-trivial meta-theorems.

We now return to the example. The induction principle for objects constructed over the given signature is just the familiar induction principle for natural numbers.

> For any property P, if P holds for `z`, and, whenever P holds for a closed object `n` of type `nat` then it also holds for `(s n)`, then P holds for every closed object of type `nat`.

Interestingly, even though this principle cannot be internalized, instances of this schema for a certain P can be checked by formulating an appropriate Elf query. In this simple case, we can prove that `double` must be total in its first argument if the queries

```
?- double z Qz.
?- {n:nat} {m:nat} double n m -> double (s n) (Qs n m).
```

both succeed. Note that the substitution terms for `Qz` and `Qs`, namely

```
Qz = z,    Qs = [n:nat] [m:nat] s (s m)
```

can be used to synthesize a schema of primitive recursion for the functional version `dbl` of `double`:

```
dbl z = Qz = z
dbl (s n) = Qs n (dbl n) = dbl (s (s n))
```

Here, `[x:A] B` is Elf concrete syntax for $\lambda x{:}A.\ B$.

Many types, such as the type of `exp` defined below, are not inductive in the usual sense (see, for example, [6]). However, we can still derive a form of an induction principle for those types, as we limit ourselves to closed LF terms constructed over a fixed given signature.

3 A Functional Language Fragment

To illustrate our technique further, consider a fragment of some functional language (here $\langle\,\rangle$ represents a unit constant):

$$\textit{Expressions}\quad e \quad ::= \quad x \mid \lambda x.e \mid e\ e \mid \langle\,\rangle \mid \langle e, e\rangle \mid \pi_1(e) \mid \pi_2(e)$$

Syntax. The above syntax can be formulated directly with the following declarations:

```
exp : type.                    unit: exp.
lam : (exp -> exp) -> exp.     pair: exp -> exp -> exp.
app : exp -> exp -> exp.       pi1 : exp -> exp.        pi2 : exp -> exp.
```

The binding construct $\lambda x.e$ is represented using *higher-order abstract syntax*, whereby meta-language abstraction represents object-level binding. This also means that Elf variables of type `exp` serve as variables of our object language.

Judgments. The following is an inference system defining (call-by-value) evaluation of expressions:

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

$$\frac{}{\langle\,\rangle \hookrightarrow \langle\,\rangle}\,\text{unit} \qquad \frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{\langle e_1, e_2\rangle \hookrightarrow \langle v_1, v_2\rangle}\,\text{pair} \qquad \frac{e \hookrightarrow \langle v_1, v_2\rangle}{\pi_i e \hookrightarrow v_i}\,\text{pi}_i$$

Three more rules determine which expressions are considered to be values:

$$\frac{}{\downarrow \langle\,\rangle}\,\text{unit} \qquad \frac{}{\downarrow \lambda x.\, e}\,\text{lam} \qquad \frac{\downarrow e_1 \qquad \downarrow e_2}{\downarrow \langle e_1, e_2\rangle}\,\text{pair}$$

These inference rules can be transcribed directly into Elf.

```
eval         : exp -> exp -> type.
eval_unit    : eval unit unit.
eval_pair    : eval (pair E1 E2) (pair V1 V2) <- eval E1 V1 <- eval E2 V2.
eval_pi1     : eval (pi1 E) V1                <- eval E (pair V1 V2).
eval_pi2     : eval (pi2 E) V2                <- eval E (pair V1 V2).
eval_lam     : eval (lam E) (lam E).
eval_app_lam : eval (app E1 E2) V
                    <- eval E1 (lam E1')
                    <- eval E2 V2
                    <- eval (E1' V2) V.

value        : exp -> type.
val_unit     : value unit.
val_pair     : value E1 -> value E2 -> value (pair E1 E2).
val_lam      : value (lam E).
```

Deduction Transformations. We would now like to state and verify a simple property relating evaluation and values, namely that the evaluation of an expression always yields a value. This is accomplished via the relation:

```
vr : eval E V -> value V -> type.
```

We have to write this relation in such a way that it can be used to establish the following:

> For every closed object `p` of type `eval e v` (where `e`, `v` are closed objects of type `exp`), there exists a closed object `VP` of type `value v` and a closed object `R` of type `vr p VP`.

To substantiate the above claim we need to define the transformation `vr` by covering the possible cases for `eval`, *i.e.*, for each proof that some expression `E` evaluates to `V` we have to supply a deduction showing that `V` is indeed a value. This is accomplished with the following clauses:

```
vr_unit     : vr (eval_unit) (val_unit).
vr_pair     : vr (eval_pair P2 P1) (val_pair VP1 VP2)
                 <- vr P1 VP1
                 <- vr P2 VP2.
vr_pi1      : vr (eval_pi1 P) VP1 <- vr P (val_pair VP1 VP2).
vr_pi2      : vr (eval_pi2 P) VP2 <- vr P (val_pair VP1 VP2).
vr_lam      : vr (eval_lam) (val_lam).
vr_app_lam  : vr (eval_app_lam P3 P2 P1) VP3 <- vr P3 VP3.
```

Schema-Checking As in the previous section, schema-checking can be performed by formulating a sequence of queries which check the various cases of the induction proof.

```
?- vr (eval_unit) Qunit.
?- vr (eval_lam)  Qlam.
?- {e1:exp} {e2:exp} {v1:exp} {v2:exp} {m:eval e2 v2} {n: eval e1 v1}
   {q: value v2} {q':value v1}
     vr m q -> vr n q'
       -> vr (eval_pair m n) (Qpair e1 e2 v1 v2 m n q q').
?- {e:exp} {v1:exp} {v2:exp} {m:eval e (pair v1 v2)} {q: value (pair v1 v2)}
     vr m q -> vr (eval_pi1 m) (Qpi1 e v1 v2 m q).
?- {e1:exp} {e1':exp -> exp} {e2:exp} {v:exp} {v2:exp} {m:eval (e1' v2) v}
   {n: eval e2 v2} {o: eval e1 (lam e1')} {q: value v} {q': value v2}
   {q'': value (lam e1')}
     vr m q -> vr n q' -> vr o q''
       -> vr (eval_app_lam m n o) (Qapp_lam e1 e1' e2 v v2 m n o q q' q'').
```

Unfortunately, the cases for `eval_pi1` and `eval_pi2` fail, while the others succeed. When analyzing the rule

```
vr_pi1 : vr (eval_pi1 P) VP1 <- vr P (val_pair VP1 VP2).
```

the reason becomes clear: in order to show totality of `vr` in its first argument, we need to know that, in this case, `vr P (val_pair VP1 VP2)` always succeeds! But this relies on a subtle point: `P` has type `eval E (pair V1 V2)` for some `E`, `V1`, and `V2`, and therefore the second argument of `vr P VP` must have type `value (pair V1 V2)`. But there is only one rule constructing a deduction with this conclusion, and therefore any closed `VP` of this type *must* have the form `(val_pair VP1 VP2)` for some `VP1` and `VP2`. This observation gives rise to the following *unit refinement* principle:

> For any property P such that P holds of `q : value (pair v1 v2)` there exist `q1 : value v1` and `q2 : value v2` such that P holds of `value_pair q1 q2`.

Note that such a principle holds whenever the principal constructor of deductions of a given judgment is uniquely determined from the judgment. This kind of reasoning arises quite often in informal meta-proofs, sometimes in a more general form using an auxiliary induction. It is closely related to the notion of *iff*-completion of logic programs.

If we Skolemize this unit refinement principle and refine our induction accordingly, the queries can now be executed successfully. The additional assumptions appear as constants `q1` and `q2`.

```
?- {e:exp} {v1:exp} {v2:exp} {m:eval e (pair v1 v2)} {q:value (pair v1 v2)}
   {q1:value (pair v1 v2) -> value v1} {q2:value (pair v1 v2) -> value v2}
     ({Q:value (pair v1 v2)} vr m Q -> vr m (val_pair (q1 Q) (q2 Q)))
     -> vr m q
     -> vr (eval_pi1 m) (Qpi1 e v1 v2 m q q1 q2).
```

This succeeds with substitution `Qpi1 = [e][v1][v2][m][q][q1][q2] q1 q`.

4 Logic Interpretations and CPS Transform

In [11], Griffin presents a number of interpretations between logics and shows how they can be viewed as computational simulations. We have transcribed and verified 8 of these interpretations. In this section we will verify the type-soundness of the *continuation-passing-style* (CPS) transform of Plotkin [25], which is one of Griffin's examples.

Syntax. Once more, the first task will be to represent the logics under consideration. Here we deal with a propositional logic, which allows their direct interpretation as types of a programming language (using the Curry-Howard isomorphism). We use α and β to range over formulas.

$$\textit{Formulas} \quad \alpha \quad ::= \quad p \mid \bot \mid \alpha \to \alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha$$

The representation of these in Elf is straightforward. Propositional variables p are not directly represented, but become meta-variables, that is, variables in the meta-language Elf.

```
form : type.
bot : form.                     imp : form -> form -> form.
and : form -> form -> form.     or  : form -> form -> form.
```

Judgments. Now we would like to formulate the necessary judgments. These are provability in minimal and classical propositional logic. Instead of separating expressions in a programming language and proofs in the propositional calculi, we can think of the deductions of formulas in these logics directly as functional programs.

$$\begin{array}{llll} \textit{Proofs} & M & ::= & x \mid \lambda x.M \mid MM \mid \langle M, M\rangle \mid \pi_1(M) \mid \pi_2(M) \\ & & & \mid \mathrm{inj}_1(M) \mid \mathrm{inj}_2(M) \mid \mathrm{case}(M, \lambda x.M, \lambda x.M) \end{array}$$

Not all expressions which follow this grammar are actually meaningful. For example, $\pi_1(\lambda x.x)$ does not make sense as a proof. In the Elf representation below we directly capture the conditions under which a proof is meaningful or *valid*. Essentially, a proof is indexed by the formula that it establishes. This can also be thought of as a refinement of the representation of untyped programs in Section 3, by indexing expressions by their type, thus dividing and restricting the space of legal programs. For a further discussion of such issues of representation, the reader is referred to [16].

```
pf : form -> type.
```

```
%% minimal logic
lam  : (pf A -> pf B) -> pf (imp A B).
app  : pf (imp A B) -> pf A -> pf B.
pair : pf A -> pf B -> pf (and A B).
pi1  : pf (and A B) -> pf A.
pi2  : pf (and A B) -> pf B.
inj1 : pf A -> pf (or A B).
inj2 : pf B -> pf (or A B).
case : pf (or A B) -> (pf A -> pf C) -> (pf B -> pf C) -> pf C.
aa   : pf bot -> pf A.                    %% intuitionistic logic
kk   : pf (imp (imp A bot) A) -> pf A.    %% classical logic
```

For example, `app` takes a proof of $\alpha \to \beta$ and a proof of α and returns a proof of β.[1] This representation is *adequate* in the sense that every natural deduction of a formula α can be represented by an object of type `pf A` and vice versa. Here `A` is the representation of α.

The proof constructors `aa` (representing the intuitionistic rule of absurdity) corresponds to an aborting operator, and `kk` corresponds to Scheme's `call/cc`, though this correspondence is not explored here.

Deduction Transformations. We now present the interpretation $\overline{(\,)}$ which maps classically provable formulas into formulas provable in minimal logic.

$$\overline{\alpha} = \neg\neg\alpha^*$$

$$\begin{aligned}\bot^* &= \bot \\ p^* &= p \\ (\alpha \wedge \beta)^* &= \alpha^* \wedge \beta^* \\ (\alpha \vee \beta)^* &= \alpha^* \vee \beta^* \\ (\alpha \to \beta)^* &= \alpha^* \to \overline{\beta}\end{aligned}$$

The mutually recursive definitions of translations $\overline{(\,)}$ and $(\,)^*$ are easily represented relationally in Elf.[2] This representation is operationally adequate.

```
cps- : form -> form -> type.
cps* : form -> form -> type.

dblneg : cps- A (imp (imp A* bot) bot) <- cps* A A*.

cps*_bot : cps* bot bot.
cps*_and : cps* (and A B) (and A* B*) <- cps* A A* <- cps* B B*.
cps*_or  : cps* (or A B)  (or A* B*)  <- cps* A A* <- cps* B B*.
cps*_imp : cps* (imp A B) (imp A* B-) <- cps* A A* <- cps- B B-.
```

This represents the translations above in the following sense: $\alpha^* = \beta$ holds iff `cps* A B`, where `A` is the representation of α, and `B` is the representation of β. Similary, translation $\overline{(\,)}$ is represented by `cps-`. We will verify the following theorem:

[1]The familiar rules for natural deduction are easily recognized: `lam` $\simeq$ $\supset$-Intro, `app` $\simeq$ $\supset$-Elim, `pair` $\simeq$ $\wedge$-Intro, `pi`$_i$ $\simeq$ $\wedge$-Elim, `inj`$_i$ $\simeq$ $\vee$-Intro, and `case` $\simeq$ $\vee$-Elim.

[2]Note that `A*`, `B-`, *etc.* are Elf variables.

If α is provable in classical logic, then $\overline{\alpha}$ is provable in minimal logic.

This is shown by a translation on the deductions: every classical deduction of α is transformed into an minimal deduction of $\overline{\alpha}$.

$$\begin{array}{rcl}
\overline{x} & = & \lambda k.kx \\
\overline{\lambda x.M} & = & \lambda k.k(\lambda x.\overline{M}) \\
\overline{MN} & = & \lambda k.\overline{M}(\lambda m.\overline{N}(\lambda n.mnk)) \\
\overline{\langle M, N\rangle} & = & \lambda k.\overline{M}(\lambda m.\overline{N}(\lambda n.k\langle m, n\rangle)) \\
\overline{\pi_i(M)} & = & \lambda k.\overline{M}(\lambda m.k(\pi_i(m))) \\
\overline{\mathrm{inj}_i} & = & \lambda k.\overline{M}(\lambda m.k(\mathrm{inj}_i(m))) \\
\overline{\mathrm{case}(M, \lambda x.N, \lambda y.Q)} & = & \lambda k.\overline{M}(\lambda m.\mathrm{case}(m, \lambda x.\overline{N}k, \lambda y.\overline{Q}k)) \\
\overline{\mathcal{A}(M)} & = & \lambda k.\overline{M}(\lambda m.m) \\
\overline{\mathcal{K}(M)} & = & \lambda k.\overline{M}(\lambda m.m(\lambda z.\lambda d.kz)k)
\end{array}$$

The representation of this translation in Elf is very direct and reproduced below. It is worth noting that the higher-level judgment connecting deductions of α and $\overline{\alpha}$ must explicitly refer to this translation. It would not suffice to specify

```
  cps : pf A -> pf A- -> type.
```

since we need to show how to relate a proof of `A` to a proof of `A-`, where `A-` is the translation of `A` under $\overline{(\,)}$. Thus the main judgment is

```
  cps : pf A -> cps- A A- -> pf A- -> type.
```

where `A` and `A-` are implicitly quantified. This judgment can be specified with the following rules (omitting the symmetric cases for π_2 and inj_2):

```
cps_lam: cps (lam M) (dblneg (cps*_imp CpsB- CpsA*)) (lam [k] app k (lam M-))
         <- ({x} {x-} cps x (dblneg CpsA*) (lam [k] app k x-)
                      -> cps (M x) CpsB- (M- x-)).
cps_app: cps (app M N) CpsB-
             (lam [k] app M- (lam [m-] app N- (lam [n-] app (app m- n-) k)))
         <- cps M (dblneg (cps*_imp CpsB- CpsA*)) M-
         <- cps N (dblneg CpsA*) N-.
cps_pair: cps (pair M N) (dblneg (cps*_and CpsB* CpsA*))
              (lam [k] app M- (lam [m-] app N- (lam [n-]app k (pair m- n-))))
          <- cps M (dblneg CpsA*) M-
          <- cps N (dblneg CpsB*) N-.
cps_pi1: cps (pi1 M) (dblneg CpsA*)
             (lam [k] app M- (lam [m-] app k (pi1 m-)))
         <- cps M (dblneg (cps*_and CpsB* CpsA*)) M-.
cps_inj1: cps (inj1 M) (dblneg (cps*_or CpsB* CpsA*))
              (lam [k] app M- (lam [m-] app k (inj1 m-)))
          <- cps M (dblneg CpsA*) M-.
cps_case: cps (case M N Q) (dblneg CpsC*)
              (lam [k] app M- (lam [m-] case m- ([x-] app (N- x-) k)
                                                ([y-] app (Q- y-) k)))
          <- cps M (dblneg (cps*_or CpsB* CpsA*)) M-
          <- ({x} {x-} cps x (dblneg CpsA*) (lam [k] app k x-)
                       -> cps (N x) (dblneg CpsC*) (N- x-))
          <- ({y} {y-} cps y (dblneg CpsB*) (lam [k] app k y-)
                       -> cps (Q y) (dblneg CpsC*) (Q- y-)).
cps_aa: cps (aa M) (dblneg CpsA*)
            (lam [k] app M- (lam [m-] m-))
        <- cps M (dblneg cps*_bot) M-.
cps_kk: cps (kk M) (dblneg CpsA*)
            (lam [k] app M- (lam [m-]app (app m- (lam [z]lam [d]app k z)) k))
        <- cps M (dblneg (cps*_imp (dblneg CpsA*)
                                   (cps*_imp (dblneg cps*_bot) CpsA*)))
               M-.
```

What does type-checking of this signature guarantee for us here? Reexamining the type of `cps`

```
cps : pf A -> cps- A A- -> pf A- -> type.
```

shows us that if a query `?- cps M CpsA N.` succeeds for some `M`, `CpsA`, and `N`, then `M` will be a proof of some formula `A`, `N` will be a proof of `A-`, and `CpsA` will be a deduction showing that `A-` is the translation of `A`. This is an important property, and many obviously incorrect translations will be caught at this stage because of type-checking errors, but it does not guarantee our theorem in any way—this is where additional *schema-checking* is required.

Schema-Checking. In order to demonstrate the theorem, we need to show that `cps-` is total in its first argument, and we also need to show that `cps` is total in its first two arguments.

Showing the totality of `cps-` demonstrates that for a given `A` we can construct an `A-` and a deduction showing that `cps- A A-`. This amounts to showing that $\overline{(\,)}$ is a total function on formulas.

Showing that `cps` is total in its first two arguments means that for a given proof `M` of type `pf A` and translation `cps- A A-` there exists a proof `M-` of type `pf A-`. That such translations always exist almost verifies the claimed theorem: It remains to be shown that the translated proof lies within the minimal fragment of the logic under consideration. This can be guaranteed through proper use of the module system for Elf, which is beyond the scope of this paper and can be found in [17].

Translations $\overline{(\,)}$ and $(\,)^*$ are mutually recursive. We add the induction hypotheses for both functions in constructing the query for this particular case (the other cases can be proven similarly):

```
?- {m:form} {n:form} {p:form} {q:form} {r:form} {s:form}
   cps* m p -> cps- m q -> cps* n r -> cps- n s
     -> cps* (imp m n) (Qand m n p q r).

Qand = [m] [n] [p] [q] [r] imp p (imp (imp r bot) bot)
```

In the induction proof for the totality of `cps` we will have to employ unit refinement again: Any proof `q : cps- A A-` will necessarily use the clause `dblneg`. In Skolemized form, this means that there is a total function which maps any deduction of `q : cps- A A-` to a deduction `q' : cps* A A*` such that `q` has the form dblneg q'. A special case of this can be expressed by the first two of the following six declarations—the others express the totality of cps- and cps*.[3]

```
  cps-_refine: cps- A A- -> cps* A A*.
  cps-_refine_lemma: cps X Q Z -> cps X (dblneg (cps-_refine Q)) Z.
  cps*_tot: form -> form.          cps*_tot_lemma: cps* M (cps*_tot M).
  cps-_tot: form -> form.          cps-_tot_lemma: cps- N (cps-_tot N).
```

Representative for schema-checking we display the query for the case of pairs and its result:

```
?- {a:form} {b:form} {a*: form} {b*: form} {x:pf a} {y:pf b}
   {m:cps- a (imp (imp a* bot) bot)} {p:pf (imp (imp a* bot) bot)}
   {n:cps- b (imp (imp b* bot) bot)} {q:pf (imp (imp b* bot) bot)}
     cps x m p -> cps y n q
       -> cps (pair x y) (Qpair a b a* b* x y m p n q)
                         (Rpair a b a* b* x y m p n q).
Rpair = [a] [b] [a*] [b*] [x] [y] [m] [p] [n] [q]
          lam ([k:pf (imp (and a* b*) bot)]
                 app p
                     (lam ([m-:pf a*]
                             app q (lam ([n-:pf b*] app k (pair m- n-)))))),
Qpair = [a] [b] [a*] [b*] [x] [y] [m] [p] [n] [q]
          dblneg (cps*_and (cps-_refine n) (cps-_refine m)).
```

5 Conclusion

We have outlined a practical methodology for the implementation and verification of the meta-theory of deductive systems. This methodology employs the LF logical framework as a basis and consists of four stages: (1) representation of syntax, (2) specification of judgments, (3) implementation of transformations between deductions, and (4) checking that the transformations are total in some of their arguments. This last stage is called *schema-checking* and relies on induction over the closed valid terms which can be constructed over a signature in the LF type theory. While a significant part of the verification is automatic through term reconstruction, type-checking, and schema-checking, much of the work is still mechanical and, we hope, amenable to methods from the field of inductive theorem proving. This would mean that an Elf theorem prover would try to synthesize an appropriate deduction transformation, such as tr or cps above.

We currently have a small prototype implementation of schema-checking as an extension of the current Elf core language [22]. Several further verifications of standard meta-theorems in logic and computer science using the methods described here are subject of current work.

[3]We omit from this discussion the unit refinements for the different cases of cps* that are needed in other parts of the proof.

References

[1] Arnon Avron, Furio A. Honsell, and Ian A. Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1987.

[2] David A. Basin and Robert L. Constable. Metalogical frameworks. Submitted, July 1991.

[3] Alan Bundy, Frank van Harmelen, Alan Smaill, and Andrew Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag LNAI 449, 1990.

[4] Rod Burstall and Furio Honsell. Operational semantics in a natural deduction setting. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 185–214. Cambridge University Press, 1991.

[5] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[6] Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88*, pages 50–66. Springer-Verlag LNCS 417, December 1988.

[7] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[8] Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30:151–185, 1983.

[9] Laurent Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In David H.D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 685–699. MIT Press, June 1990.

[10] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[11] Timothy G. Griffin. Logical interpretations as computational simulations. Draft paper. Talk given at the North American Jumelage, AT&T Bell Laboratories, Murray Hill, New Jersey 1991, October 1991.

[12] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(5):635–660, 1991.

[13] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *ACM Conference on Lisp and Functional Programming*, pages 323–332. ACM Press, 1990.

[14] John Hannan and Frank Pfenning. Compiler verification in LF. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, Santa Cruz, California, June 1992. IEEE Computer Society Press. To appear. Preliminary version available as POP Report 91–003, School of Computer Science, Carnegie Mellon University.

[15] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[16] Robert Harper, Furio Honsell, and Gordon 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.

[17] Robert Harper and Frank Pfenning. Modularity in the LF logical framework. Submitted. Available as POP Report 91–001, School of Computer Science, Carnegie Mellon University, November 1991.

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

[19] Ian A. Mason. Hoare's logic in the LF. Technical Report ECS-LFCS-87-32, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1987.

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

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

[22] Frank 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.

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

[24] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85. IEEE Computer Society Press, July 1991.

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

[26] Lutz Plümer. *Termination Proofs for Logic Programs.* Springer-Verlag LNAI 446, 1991.

[27] David Pym. *Proofs, Search and Computation in General Logic.* PhD thesis, University of Edinburgh, 1990. Available as CST-69-90, also published as ECS-LFCS-90-125.

[28] David Pym and Lincoln Wallen. Investigations into proof-search in a system of first-order dependent function types. In M.E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, Germany*, pages 236–250. Springer-Verlag LNCS 449, July 1990.