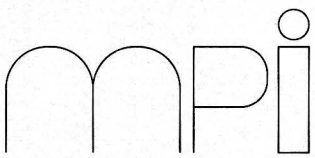
MAX-PLANCK-INSTITUT FÜR INFORMATIK

Natural Semantics and Some of its Meta-Theory in Elf

Spiro Michaylov Frank Pfenning

MPI-I-91-211 August 1991



Im Stadtwald
W 6600 Saarbrücken
Germany

Natural Semantics and Some of its Meta-Theory in Elf

Spiro Michaylov Frank Pfenning

MPI-I-91-211 August 1991

Author's Address

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890
U.S.A.

Acknowledgements

We are grateful to the Max-Planck-Institute for Computer Science in Saarbrücken where the second author completed the paper during an extended visit.

Abstract

Operational semantics provide a simple, high-level and elegant means of specifying interpreters for programming languages. In natural semantics, a form of operational semantics, programs are traditionally represented as first-order tree structures and reasoned about using natural deduction-like methods. Hannan and Miller combined these methods with higher-order representations using λProlog.

In this paper we go one step further and investigate the use of the logic programming language Elf to represent natural semantics. Because Elf is based on the LF Logical Framework with dependent types, it is possible to write programs that reason about their own partial correctness. We illustrate these techniques by giving type checking rules and operational semantics for Mini-ML, a small functional language based on a simply typed λ-calculus with polymorphism, constants, products, conditionals, and recursive function definitions. We also partially internalize proofs for some meta-theoretic properties of Mini-ML, the most difficult of which is subject reduction.

Keywords

Logical Frameworks, Operational Semantics, Type Systems, Logic Programming

Contents

1 Introduction

Operational semantics provide a simple, high-level and elegant means of specifying interpreters for programming languages. Natural semantics was inspired by the work of Plotkin [19] on operational semantics and extended by G. Kahn [12] and others. Programs are traditionally represented as first-order tree structures and reasoned about using natural deduction-like methods.

This method was extended and refined by Burstall and Honsell [1] and Hannan and Miller [6, 7]. Using higher-order abstract syntax, programs are represented by simply-typed λ-terms and schema variables in inference rules become higher-order variables. Further, they applied methods for introducing and discharging assumptions (for expressing hypothetical judgments) and parameters (for expressing generic or universal judgments) to this setting of natural semantics. Thus it was natural that Hannan and Miller's implementation language was λProlog [14]. λProlog was a particularly appropriate tool because its embedded implication mechanism is suitable for expressing hypothetical judgments in natural deduction, and it's higher-order features make it suitable for representing programs in terms of higher-order abstract syntax.

Mini-ML was introduced by Clément *et al.* [3] as a small programming language based on the simply typed λ-calculus with products, conditionals, and recursive function definitions. In this paper we use Elf [16, 17], a logic programming language based on the LF Logical Framework [10], as an implementation language for natural semantics. Elf embodies the same features that make λProlog so suitable for implementing natural semantics. Additionally, the dependent types of LF make it possible for programs to reason about deductions of goals and subgoals, making Elf suitable for reasoning about the properties of an implementation within that very implementation. In particular, programs can express aspects of their own partial correctness proof.

This document is intended for readers who are at least acquainted with operational semantics and logic programming, although some familiarity with the previous work on natural semantics, and the use of λProlog, is an advantage.

The remainder of this document is organized as follows. We describe our example language which is based on Mini-ML (and called Mini-ML throughout), giving its concrete syntax, type checking rules and operational semantics. Then we introduce our implementation language Elf, using examples from the description of Mini-ML. Then we describe the Elf code for Mini-ML type inference and an implementation of the operational semantics. We then show, through a number of examples, how some aspects of the meta-theory of Mini-ML can be represented in Elf as judgments on deductions. The most complicated example in this class is the subject reduction property for our language. Complete listings of the Elf code discussed are given in the appendices.

2 The Mini-ML Language

Mini-ML [3] is a functional programming language based on a simply typed λ-calculus with polymorphism, constants, products, conditionals, and recursive function definitions. The language we define here is a slight variation because patterns are replaced with explicit projections to simplify the presentation. The only substantial disadvantage of this is that it makes the definition of mutually recursive functions a little untidy. Furthermore, we add a fixpoint operator `fix` to make it easier to describe the typing and semantics of the `letrec` construct, although `fix` need not necessarily be made accessible to the programmer through the concrete syntax.

In the remainder of this section we describe the concrete syntax of our variant of Mini-ML and give the rules for typing and the operational semantics.

2.1 Concrete Syntax

We use the meta-variables x, e and τ, possibly subscripted, for the syntactic categories of variables, expressions and types, respectively. As usual, application associates to the left and type arrows associate to the right. Parentheses are used for disambiguation.

```
e ::=       true
          | false
          | if e1 then e2 else e3
          | z
          | s
          | pred
          | zerop
          | < e1 , e2 >
          | fst e
          | snd e
          | lambda x . e
          | e1 e2
          | let x = e1 in e2
          | letrec x = e1 in e2
          | fix x . e
          | x

τ ::=       Bool
          | Nat
          | τ -> τ
          | τ * τ
```

The following is a Mini-ML expression implementing addition (in curried form)

```
letrec add = lambda x .
                lambda y .
                   if (zerop x) then y
                                else (s (add (pred x) y))
          in add
```

which has type `Nat -> Nat -> Nat`.

2.2 Typing Rules

In the following we give typing rules for Mini-ML. Type judgments are of the form $\Pi \vdash e \in \tau$ where Π is a type environment, e is a Mini-ML expression and τ is a Mini-ML type. We use "," for environment concatenation and $[\, e \,/\, x \,]\, e'$ for the result of substituting e for x in e', renaming bound variables as necessary to avoid capture. Furthermore, we say that $\Pi(x) = \tau$ if the rightmost occurrence of x in Π is assigned τ. Type assignment in an environment is written as $x : \tau$.

Harper [9] studied several formulations for the Damas-Milner fragment of ML [4], with a view toward their formalization in LF, and discussed their suitability for direct execution. We use a simpler encoding that is sufficient for our purposes. The idea for this also goes back to Hannan and Miller [7].

of_t
$$\frac{}{\Pi \vdash \texttt{true} \in \texttt{Bool}}$$

of_f
$$\frac{}{\Pi \vdash \texttt{false} \in \texttt{Bool}}$$

of_if
$$\frac{\Pi \vdash e_1 \in \texttt{Bool} \qquad \Pi \vdash e_2 \in \tau \qquad \Pi \vdash e_3 \in \tau}{\Pi \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \in \tau}$$

of_z
$$\frac{}{\Pi \vdash \texttt{z} \in \texttt{Nat}}$$

of_s
$$\frac{}{\Pi \vdash \texttt{s} \in \texttt{Nat -> Nat}}$$

of_pred
$$\frac{}{\Pi \vdash \texttt{pred} \in \texttt{Nat -> Nat}}$$

of_zerop
$$\frac{}{\Pi \vdash \texttt{zerop} \in \texttt{Nat -> Bool}}$$

of_pair
$$\frac{\Pi \vdash e_1 \in \tau_1 \qquad \Pi \vdash e_2 \in \tau_2}{\Pi \vdash \texttt{<} e_1\texttt{,} \; e_2 \texttt{>} \in \tau_1 \texttt{ * } \tau_2}$$

of_fst
$$\frac{\Pi \vdash e \in \tau_1 \texttt{ * } \tau_2}{\Pi \vdash \texttt{fst } e \in \tau_1}$$

of_snd
$$\frac{\Pi \vdash e \in \tau_1 \texttt{ * } \tau_2}{\Pi \vdash \texttt{snd } e \in \tau_2}$$

of_lam
$$\frac{\Pi\texttt{, } x \texttt{ : } \tau_1 \vdash e \in \tau_2}{\Pi \vdash \texttt{lambda } x \texttt{ . } e \in \tau_1 \texttt{ -> } \tau_2}$$

of_app
$$\frac{\Pi \vdash e_1 \in \tau_2 \texttt{ -> } \tau_1 \qquad \Pi \vdash e_2 \in \tau_2}{\Pi \vdash e_1 \, e_2 \in \tau_1}$$

of_let
$$\frac{\Pi \vdash e_1 \in \tau_1 \qquad \Pi \vdash [\, e_1 \,/\, x \,] e_2 \in \tau_2}{\Pi \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \in \tau_2}$$

of_letrec
$$\frac{\Pi \vdash \texttt{fix } x \texttt{ . } e_1 \in \tau_1 \qquad \Pi \vdash [\, (\texttt{fix } x \texttt{ . } e_1) \,/\, x \,] e_2 \in \tau_2}{\Pi \vdash \texttt{letrec } x = e_1 \texttt{ in } e_2 \in \tau_2}$$

of_fix
$$\frac{\Pi\texttt{, } x \texttt{ : } \tau \vdash e \in \tau}{\Pi \vdash \texttt{fix } x \texttt{ . } e \in \tau}$$

of_var
$$\frac{\Pi(x) = \tau}{\Pi \vdash x \in \tau}$$

2.3 Operational Semantics

As an additional notational convention here, we adopt α, possibly subscripted, for expressions that are the result of evaluation. Evaluation is expressed as the judgment $\vdash e \Rightarrow \alpha$, where e is the original expression and α is the value to which it evaluates, again a Mini-ML expression.

Note that we have chosen `pred zero` to be undefined, and hence there is no corresponding rule. Thus there are two possibilities of how an attempt to construct a deduction of $\vdash e \Rightarrow \alpha$ could fail if e is given and α unknown. We might encounter a situation where no rule is applicable (as in `pred z` $\Rightarrow \alpha$), or we might construct an infinite tree (for non-terminating Mini-ML computations).

eval_t
$$\frac{}{\vdash \texttt{true} \Rightarrow \texttt{true}}$$

eval_f
$$\frac{}{\vdash \texttt{false} \Rightarrow \texttt{false}}$$

eval_z
$$\frac{}{\vdash \texttt{z} \Rightarrow \texttt{z}}$$

eval_s
$$\frac{}{\vdash \texttt{s} \Rightarrow \texttt{s}}$$

eval_pred
$$\frac{}{\vdash \texttt{pred} \Rightarrow \texttt{pred}}$$

eval_zerop
$$\frac{}{\vdash \texttt{zerop} \Rightarrow \texttt{zerop}}$$

eval_pair
$$\frac{\vdash e_1 \Rightarrow \alpha_1 \qquad \vdash e_2 \Rightarrow \alpha_2}{\vdash \texttt{<}\, e_1\texttt{,}\ e_2 \,\texttt{>} \Rightarrow \texttt{<}\, \alpha_1\texttt{,}\ \alpha_2 \,\texttt{>}}$$

eval_fst
$$\frac{\vdash e \Rightarrow \texttt{<}\, \alpha_1\texttt{,}\ \alpha_2 \,\texttt{>}}{\vdash \texttt{fst}\ e \Rightarrow \alpha_1}$$

eval_snd
$$\frac{\vdash e \Rightarrow \texttt{<}\, \alpha_1\texttt{,}\ \alpha_2 \,\texttt{>}}{\vdash \texttt{snd}\ e \Rightarrow \alpha_2}$$

eval_if_t
$$\frac{\vdash e_1 \Rightarrow \texttt{true} \qquad \vdash e_2 \Rightarrow \alpha}{\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \alpha}$$

eval_if_f
$$\frac{\vdash e_1 \Rightarrow \texttt{false} \qquad \vdash e_3 \Rightarrow \alpha}{\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \alpha}$$

eval_lam
$$\frac{}{\vdash \texttt{lambda}\ x\ \texttt{.}\ e \Rightarrow \texttt{lambda}\ x\ \texttt{.}\ e}$$

eval_app_lam
$$\frac{\vdash e_1 \Rightarrow (\texttt{lambda}\ x\ \texttt{.}\ e_1') \qquad \vdash e_2 \Rightarrow \alpha_2 \qquad \vdash [\,\alpha_2 \,/\, x\,]\, e_1' \Rightarrow \alpha}{\vdash e_1\ e_2 \Rightarrow \alpha}$$

eval_app_s
$$\frac{\vdash e_1 \Rightarrow \mathtt{s} \qquad \vdash e_2 \Rightarrow \alpha}{\vdash e_1\ e_2 \Rightarrow \mathtt{s}\ \alpha}$$

eval_app_zerop_t
$$\frac{\vdash e_1 \Rightarrow \mathtt{zerop} \qquad \vdash e_2 \Rightarrow \mathtt{z}}{\vdash e_1\ e_2 \Rightarrow \mathtt{true}}$$

eval_app_zerop_f
$$\frac{\vdash e_1 \Rightarrow \mathtt{zerop} \qquad \vdash e_2 \Rightarrow \mathtt{s}\ \alpha}{\vdash e_1\ e_2 \Rightarrow \mathtt{false}}$$

eval_app_pred_s
$$\frac{\vdash e_1 \Rightarrow \mathtt{pred} \qquad \vdash e_2 \Rightarrow \mathtt{s}\ \alpha}{\vdash e_1\ e_2 \Rightarrow \alpha}$$

eval_let
$$\frac{\vdash e_1 \Rightarrow \alpha_1 \qquad \vdash [\,\alpha_1 \,/\, x\,]\, e_2 \Rightarrow \alpha_2}{\vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Rightarrow \alpha_2}$$

eval_letrec
$$\frac{\vdash \mathtt{fix}\ x\ .\ e_1 \Rightarrow \alpha_1 \qquad \vdash [\,\alpha_1 \,/\, x\,]\, e_2 \Rightarrow \alpha_2}{\vdash \mathtt{letrec}\ x = e_1\ \mathtt{in}\ e_2 \Rightarrow \alpha_2}$$

eval_fix
$$\frac{\vdash [\,(\,\mathtt{fix}\ x\ .\ e\,) \,/\, x\,]\, e \Rightarrow \alpha}{\vdash \mathtt{fix}\ x\ .\ e \Rightarrow \alpha}$$

3 The Elf Language

Elf can be loosely described as a constraint logic programming language with higher-order unification, implication, types, and a powerful ability to reason about deductions of its goals. A formal discussion of the language can be found in [16, 17], but here we take a more pragmatic approach. As our running example we will use parts of the abstract syntax and type inference and semantics for a higher-order formulation of Mini-ML, based on the syntax and rules in Section 2. However, detailed descriptions of these will be left to Section 4. Since there are various notions of "proof" necessary in the discussion of a meta-logical language such as Elf, we will use the following terminology. *Deductions* are proofs of goals as they are carried out by the Elf interpreter, *derivations* are formal proofs of judgments using a specified set of inference rules, and *proofs* are meta-level proofs establishing a property of a set of inference rules.

The basic building block of an Elf program is a signature, that is, a sequence of constant declarations. Some of these declarations have the character of declarations for data constructors or predicates, while others have the character of declarations of inference rules or clauses. In principle, each declaration could be given all of those interpretations, but in practice few are meaningful.

Using Elf's module system, one can explicitly distinguish the various roles of declarations. For the purposes of this paper, we will only use the Elf core language and give interpretations informally. Further discussion on this subject can be found in Section 8 and [17].

3.1 Type and Type Family Declarations

The first kind of declaration is that of a constant *type* or *type family*. Types and type families are classified by *kinds*, the simplest of which is **type**.

famdecl ::= *famconst* : *kind*.

```
kind ::=    type
          | type -> kind
          | { var : type } kind
```

For example, we can declare the type `exp` of Mini-ML expressions and the type `tp` of Mini-ML types using

```
exp : type.
tp  : type.
```

This example falls under the first style of *kind*, which is the most common. We will consider the other two styles, and the *types* they require, later. For now, it suffices to say that we will need to be able to define judgments (such as $\vdash e \in \tau$) as types.

3.2 Constant Declarations

A data constructor declaration is of the form

```
decl ::=    const : type.

type ::=    atom
          | type -> type
          | { var : type } type

atom ::=    famconst obj*
```

so the abstract syntax for the Mini-ML type constructors `Nat` and `*` respectively can be declared as

```
nat   : tp.
cross : tp -> tp -> tp.
```

and that for the Mini-ML term constructors `z` and `< , >` is

```
z    : exp.
pair : exp -> exp -> exp.
```

This example only uses the first two (simple) styles of *type*. The third style (*dependent function types*) will become necessary when we need to declare types that describe inference rules defining a judgment. Note also that the sequence of objects *obj** is empty in these declarations, since `tp` and `exp` are simple types, and not type families. The dependent function type, { *var* : *type* } *type*, is elsewhere often written as $\Pi var : type.\ type$.

Before moving on to predicates and inference rules, let us give the definitions of *objects* which are used to represent data. For example, any *object* of type `exp` represents a Mini-ML expression, any *object* of type `tp` represents a Mini-ML type.

obj ::= *const*
| *var*
| [*var* : *type*] *obj*
| *obj obj*

Juxtaposition is left-associative, so that the representation of the Mini-ML expression `< z, z >` can be written as the object `pair z z` of type `exp`.

The square brackets denote λ-abstraction, and [*var* : *type*] *obj* is more traditionally written as $\lambda var : type.\ obj$. We adopt the shorthand [*var*] and { *var* } for those cases of abstraction or quantification where we want Elf type reconstruction to fill in the details to the right of the ":". The scope of the abstraction [...] and quantification {...} extends to the end of the declaration or enclosing parentheses, so that the object `lam [x] app s x` has the same reading as `lam ([x] ((app s) x))`.

3.3 Predicates

Now we come to the syntax of predicate declarations. In Elf, predicates manifest themselves as type families and thus a type family declaration can be interpreted as a predicate declaration. For example, for type checking we need to declare the relation

```
of : exp -> tp -> type.
```

between a Mini-ML expression and its type. Again, this example does not require dependent *kinds*, which will be used later when we need predicates to include arguments that are constrained to be deductions. These "constraints" will be part of the declaration of a procedure.

3.4 Rules

Rule definitions (clauses) are nothing but constant definitions. One can think of the constant as *naming* the rule or clause. This name can then be used in the explicit construction of deductions.

When defining rules we often use the more conspicuous left-pointing arrow. One should bear in mind that this has no semantic significance, and `A <- B` and `B -> A` are parsed to the same internal representation. The backwards arrow is left assocative and a Prolog rule `p :- q, r.` would be represented via the type `p <- q <- r`.

Let us consider a few example rules. First, the type checking rule

of_z
$$\frac{}{\Pi \vdash z \in \mathtt{Nat}}$$

has the simple implementation

```
of_z        : of z nat.
```

The rule has the name `of_z`, and $\in$ has been replaced by the relation symbol `of`. The type environment Π is represented by additional Elf rules using embedded implication, as we shall see later.

To check the type of a Mini-ML pair, which has the inference rule

of_pair
$$\frac{\Pi \vdash e_1 \in \tau_1 \qquad \Pi \vdash e_2 \in \tau_2}{\Pi \vdash \langle e_1, e_2 \rangle \in \tau_1 * \tau_2}$$

we need to write the following rule, which has the name `of_pair`, and which recursively checks the types of the two elements of the pair.

```
of_pair    : of (pair E1 E2) (cross A1 A2)
                <- of E1 A1
                <- of E2 A2.
```

At this stage we should point out the role of the Elf front-end in preprocessing programs before Elf type reconstruction. Using the above example again, this adds explicit universal quantification to a rule for all free named variables. That is, `of_pair` becomes

```
of_pair :
   {E2:exp} {A2:tp} {E1:exp} {A1:tp}
      of E2 A2 -> of E1 A1 -> of (pair E1 E2) (cross A1 A2).
```

and the deduction showing that `< z, z >` has type `Nat * Nat` would be represented as `(of_pair z nat z nat of_z of_z)`. It should be easy to see how this much syntax could get out of hand, so the usual style of programming in Elf is to omit such quantifiers whenever possible, as in Prolog. This causes the corresponding components of deductions to be suppressed as well. However, occasionally explicit quantification is needed either for embedded implication or for obtaining access to otherwise hidden arguments in deductions.

The variable name convention of Elf is much like that of Prolog. Any token beginning with an upper case letter is automatically a variable, and if an explicit quantifier is not provided, one will be added by type reconstruction. Additionally, variables that are explicitly quantified need not start with an upper case letter. We follow the convention that bound variables which will become logic variables (and thus subject to instantiation) during execution of a query are written in uppercase and bound variables which become parameters (and thus act like constants to unification) are written in lowercase.

In Elf, unlike Prolog, an underscore character does not denote an anonymous universally quantified variable, although it can sometimes be used for this purpose. Instead, it describes an existentially quantified anonymous variable, and type reconstruction is at liberty to replace an underscore with any term, depending on the available type information. Thus `{x}B` and `{x:_}B` are fully equivalent. In Prolog, since no variables are ever instantiated during parsing, this distinction does not arise.

3.5 Queries

Queries to the Elf interpreter consist of a type, possibly with free variables which act as logic variables. The goal is to find a term of the given type. In the most common interpretation, this term represents a deduction of the judgment which is represented by the type. During the solution of a goal, the free variables in the query are instantiated as necessary, as in Prolog. For example, the query

```
?- of (pair z z) A.
```

would give the answer `(cross nat nat)` for `A`, and the deduction `(of_pair of_z of_z)` which represents a derivation that `< z, z >` has type `Nat * Nat`. As will be described in more detail later, the names of rules are used as constructors in assembling deductions.

The operational model of Elf is superficially much like that of Prolog and λProlog. A computation rule similar to Prolog's left-right atom selection rule is employed. Unification is modulo $\beta\eta$-convertibility, where certain equations are postponed as constraints. A goal of the form `A -> B` is solved by adding `A` to the set of rules available for goal reduction. The third form (universal quantification) is solved by replacing the universally quantified variable with a new constant as in λProlog.

In order to make this operational model work, distinctions between types and goals which are erased through the use of the judgments-as-types principle, must be reintroduced to some extent. Further discussion on this issue can be found in Section 8.

4 An Implementation of Mini-ML in Elf

Here we work through the implementation of Mini-ML in Elf, starting with its higher-order abstract syntax, a type checker, and the first of two versions of its operational (natural) semantics.

4.1 Higher-Order Abstract Syntax

Now we are ready to develop a scheme for expressing Mini-ML programs in Elf, using higher-order abstract syntax. This approach was inspired by Church [2] and Martin-Löf [15] and is used pervasively in many applications of λProlog and Elf. In each case, a λ-calculus-based meta-language (here Elf) is used to represent expressions of the object language (here Mini-ML). This enables variable binding in the object language to be represented with the help of λ-abstraction in the meta-language. This, in turn, enables substitutions in the inference rules we have seen to be implemented using β-reduction in the meta-language, which avoids, for example, the requirement to perform explicit α-conversion to prevent capture of bound variables. We will see many examples of this later on. Here we define a translation function $()^+$ from Mini-ML expressions and types to Elf expressions.

$$
\begin{aligned}
(\texttt{true})^+ &= \texttt{true} \\
(\texttt{false})^+ &= \texttt{false} \\
(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)^+ &= (\texttt{if } (e_1)^+ \ (e_2)^+ \ (e_3)^+) \\
(\texttt{z})^+ &= \texttt{z} \\
(\texttt{s})^+ &= \texttt{s} \\
(\texttt{pred})^+ &= \texttt{pred} \\
(\texttt{zerop})^+ &= \texttt{zerop} \\
(\texttt{<} e_1 \texttt{, } e_2 \texttt{>})^+ &= (\texttt{pair } (e_1)^+ \ (e_2)^+) \\
(\texttt{fst } p)^+ &= (\texttt{fst } (p)^+) \\
(\texttt{snd } p)^+ &= (\texttt{snd } (p)^+) \\
(\texttt{lambda } x \texttt{ . } e)^+ &= (\texttt{lam [}x\texttt{:exp]} \ (e)^+) \\
(e_1 \ e_2)^+ &= (\texttt{app } (e_1)^+ \ (e_2)^+) \\
(\texttt{let } x = e_1 \texttt{ in } e_2)^+ &= (\texttt{let } (e_1)^+ \ (\texttt{[}x\texttt{:exp]} \ (e_2)^+)) \\
(\texttt{letrec } x = e_1 \texttt{ in } e_2)^+ &= (\texttt{letrec } (\texttt{[}x\texttt{:exp]} \ (e_1)^+) \ (\texttt{[}x\texttt{:exp]} \ (e_2)^+)) \\
(\texttt{fix } x \texttt{ . } e)^+ &= (\texttt{fix [}x\texttt{:exp]} \ (e)^+) \\
(x)^+ &= x
\end{aligned}
$$

Let us consider, for example, an expression of the form `let` $x = e_1$ `in` e_2. The variable x is bound in e_2, but not in e_1. Therefore, the representing constant `let` has two arguments: the first the representation of e_1, the second the representation of e_2, abstracted over x. Since x can occur in e_2 in place of an expression, its type is `exp`, and the type of `let` becomes `exp -> (exp -> exp) -> exp`.

The representation of Mini-ML types is simpler, since it involves no variable binding constructs.

$$\begin{aligned} (\texttt{Bool})^+ &= \texttt{bool} \\ (\texttt{Nat})^+ &= \texttt{nat} \\ (\tau_1 \texttt{ * } \tau_2)^+ &= \texttt{(cross } (\tau_1)^+ \ (\tau_2)^+\texttt{)} \\ (\tau_1 \texttt{ -> } \tau_2)^+ &= \texttt{(arrow } (\tau_1)^+ \ (\tau_2)^+\texttt{)} \end{aligned}$$

In the defining signature, we require no syntactic category for identifiers since all Mini-ML variables correspond directly to Elf variables. Thus we only need expressions (`exp` in Elf) and types (`tp` in Elf, since `type` is a keyword).

```
exp    : type.

true   : exp.
false  : exp.
if     : exp -> exp -> exp -> exp.

z      : exp.
s      : exp.
pred   : exp.
zerop  : exp.

pair   : exp -> exp -> exp.
fst    : exp -> exp.
snd    : exp -> exp.

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

let    : exp -> (exp -> exp) -> exp.

letrec : (exp -> exp) -> (exp -> exp) -> exp.
fix    : (exp -> exp) -> exp.

tp     : type.

bool   : tp.
nat    : tp.
cross  : tp -> tp -> tp.
arrow  : tp -> tp -> tp.
```

The most complex constructor, `letrec`, warrants some special consideration. In an expression of the form `letrec` $x = e_1$ `in` e_2, the variable x is bound in e_1 as well as e_2—this is how it differs from

`let`. In the absence of pairing in Elf, the most natural representation is through a constant `letrec` with two arguments (the representations of e_1 and e_2), both abstracted over x. This introduces two abstractions whereas the concrete syntax contains only one. This means that we might have to α-convert one to the other in order to obtain the concrete Mini-ML syntax for a `letrec` expression. For a more general solution in the presence of meta-language pairing, see [18].

As an example of how the representation works, the addition program from section 2 would be expressed as:

```
(letrec
   ([add] (lam [x] (lam [y]
                        (if (app zerop x)
                            y
                            (app s (app (app add (app pred x)) y))))))
   ([add] add))
```

and its type would be expressed as `(arrow nat (arrow nat nat))`.

4.2 Typing Rules

Next we describe and define a relation `of` between a Mini-ML expression and its type. The code is more or less a direct translation of the inference rules in Section 2, and is given in full in Appendix B. Here we will discuss the more interesting rules in some detail. First, the declaration of the predicate `of` is:

```
of  : exp -> tp -> type.
```

That is, the first argument is a Mini-ML expression and the second argument is its Mini-ML type. In the previous section we saw two very simple rules for `of_z` and `of_pair`. Recall that the inference rules were

of_z
$$\frac{}{\Pi \vdash \texttt{z} \in \texttt{Nat}}$$

of_pair
$$\frac{\Pi \vdash e_1 \in \tau_1 \qquad \Pi \vdash e_2 \in \tau_2}{\Pi \vdash \texttt{<}e_1\texttt{,}\ e_2\texttt{>} \in \tau_1\ \texttt{*}\ \tau_2}$$

and the Elf code was

```
of_z      : of z nat.
of_pair   : of (pair E1 E2) (cross A1 A2)
              <- of E1 A1
              <- of E2 A2.
```

Most of the other cases are just as simple. The rule for `lambda` illustrates universal quantification in a goal, as well as the use of assumptions. This technique is familiar from λProlog.

of_lam
$$\frac{\Pi\texttt{,}\ x\ \texttt{:}\ \tau_1 \vdash e \in \tau_2}{\Pi \vdash \texttt{lambda}\ x\ \texttt{.}\ e \in \tau_1\ \texttt{->}\ \tau_2}$$

The scope of the quantifier {x:exp} below extends all the way to the end of the declaration. Thus this rule has one subgoal, as expected.

```
of_lam      : of (lam E) (arrow A1 A2)
                  <- {x:exp} of x A1 -> of (E x) A2.
```

Thus the derivation of the premiss of of_lam is represented as a function which maps expressions e_1 and deductions showing that e_1 has type τ_1 to derivations showing that $[\, e_1 \,/\, x \,]\, e_2$ has type τ_2. When this rule is used in the deduction of a goal, it replaces the parameter of the higher-order term with a new parameter and assumes a rule (effectively a lemma) about the typing of that variable. Hence, whenever the variable is encountered in subsequent type checking this assumed rule (fact) will be used to retrieve its type. The rule for `fix`

of_fix

$$\frac{\Pi\,,\, x : \tau \vdash e \in \tau}{\Pi \vdash \texttt{fix}\ x\ .\ e \in \tau}$$

is implemented using the same technique:

```
of_fix      : of (fix E) A <- {x:exp} of x A -> of (E x) A.
```

The rules for `let` and `letrec` illustrate the utility of higher order terms, as both of the inference rules require substitutions. The two are similar so we will restrict our attention to `let`.

of_let

$$\frac{\Pi \vdash e_1 \in \tau_1 \qquad \Pi \vdash [\, e_1 \,/\, x \,]\, e_2 \in \tau_2}{\Pi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \in \tau_2}$$

One possible implementation simply uses Elf application to represent substitution:

```
of_let'     : of (let E1 E2) A2
                  <- of E1 A1
                  <- of (E2 E1) A2
```

There is another, equally valid pair of rules for `let` and `letrec`. These have the advantage that they do not require substitution the way those above do, and they are in some sense more natural. We include that for `let` here because it demonstrates some more of the expressive power of Elf.

```
of_let      : of (let E1 E2) A2
                  <- of E1 A1
                  <- {x:exp} ({A:tp} of x A <- of E1 A)
                               -> of (E2 x) A2.
```

The first condition in the body of the rule ensures that the let-bound term E1 is well-typed in isolation. The second condition uses an extension of the technique used to implement `lam`. The let-bound variable in the body is replaced by a parameter x, and a rule is assumed which ensures that whenever x is encountered while type-checking E2, the expression E1 will be type-checked in its place. Note that the type A of x is universally quantified *within* the asserted rule, so that x can be given different (Mini-ML) types at different occurrences in E2.

Both of the rules we have given for typechecking the `let` construct are somewhat unconventional, as the bound expression is type-checked every time it is encountered in the body. The more usual approach is to construct the type of the bound expression once and abstract over all the free variables remaining in the type which are not free in the current type environment. However, these free variables cannot be statically predetermined. Thus, if Mini-ML type unification is to be represented by Elf unification, such abstraction cannot be represented in Elf. Dietzen and Pfenning [5] show how this problem can be overcome for λProlog by adding a special construct called `rule` to the language, and a similar extension to Elf seems to be possible. For a number of variants of declarative presentations of the type system with explicit type schemas, see [9].

4.3 Natural Operational Semantics

Sometimes the most "natural" semantics for a programming language is "nondeterministic" in the sense that its execution would require backtracking in our chosen implementation language. However, the corresponding "deterministic" version is usually more efficient, and probably corresponds more closely to practical applications. We will first give a nondeterministic (but nevertheless executable) semantics for Mini-ML (note the nondeterminism in the handling of if-then-else and application) and later (in Section 6) what we call the "algorithmic" one, which is deterministic. This is again a more or less direct translation of the rules given in Section 2. Again we will leave the full listing to Appendix C, and describe some highlights here.

The semantics we describe here is a call-by-value semantics. This could easily be modified to describe call-by-name instead (see [7]). We begin with the declaration of the evaluation predicate `neval`:

```
neval      : exp -> exp -> type.
```

The first argument is a Mini-ML expression, and the second argument is the value of that Mini-ML expression. To begin with a trivial example, consider the rule

eval_z

$$\frac{}{\vdash \mathtt{z} \Rightarrow \mathtt{z}}$$

which can be implemented as

```
neval_z          : neval z z.
```

Again the implementation of pairs is quite simple, for example

eval_pair

$$\frac{\vdash e_1 \Rightarrow \alpha_1 \qquad \vdash e_2 \Rightarrow \alpha_2}{\vdash \mathtt{<}\, e_1 \,,\, e_2 \,\mathtt{>} \Rightarrow \mathtt{<}\, \alpha_1 \,,\, \alpha_2 \,\mathtt{>}}$$

eval_fst

$$\frac{\vdash e \Rightarrow \mathtt{<}\, \alpha_1 \,,\, \alpha_2 \,\mathtt{>}}{\vdash \mathtt{fst}\ e \Rightarrow \alpha_1}$$

are implemented as

```
neval_pair       : neval (pair E1 E2) (pair V1 V2)
```

```
                        <- neval E1 V1
                        <- neval E2 V2.
neval_fst       : neval (fst E) V1
                        <- neval E (pair V1 V2).
```

The rules

eval_let
$$\frac{\vdash e_1 \Rightarrow \alpha_1 \qquad \vdash [\alpha_1 / x] e_2 \Rightarrow \alpha_2}{\vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow \alpha_2}$$

eval_letrec
$$\frac{\vdash \texttt{fix}\ x\ .\ e_1 \Rightarrow \alpha_1 \qquad \vdash [\alpha_1 / x] e_2 \Rightarrow \alpha_2}{\vdash \texttt{letrec}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow \alpha_2}$$

for `let` and `letrec` are also quite straightforward:

```
neval_let       : neval (let E1 E2) V2
                        <- neval E1 V1
                        <- neval (E2 V1) V2.

neval_letrec    : neval (letrec E1 E2) V
                        <- neval (fix E1) V1
                        <- neval (E2 V1) V2.
```

Notice again the use of meta-level application `(E2 V1)` to achieve substitution.

The operationally interesting aspects of `neval` in this case are those involving branching and application. For example, consider the straightforward implementation of

eval_if_t
$$\frac{\vdash e_1 \Rightarrow \texttt{true} \qquad \vdash e_2 \Rightarrow \alpha}{\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \alpha}$$

eval_if_f
$$\frac{\vdash e_1 \Rightarrow \texttt{false} \qquad \vdash e_3 \Rightarrow \alpha}{\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \alpha}$$

which is

```
neval_if_t      : neval (if E1 E2 E3) V
                        <- neval E1 true
                        <- neval E2 V.
neval_if_f      : neval (if E1 E2 E3) V
                        <- neval E1 false
                        <- neval E3 V.
```

Any `if` statement will be evaluated by first evaluating `E1`. If it evaluates to `true`, `E2` is evaluated and the result is returned through `V`. However, if `E1` evaluates to `false`, backtracking occurs, the second rule is matched, and `E1` is *again* evaluated. It again evaluates to `false`, `E3` is evaluated and the result is returned through `V`. So the first source of inefficiency is that `E1` may be evaluated

twice. The second is that even if E1 does evaluate to true and failure occurs later, the second rule will again be matched on backtracking and E1 will be re-evaluated. In addition to the inefficiency, clearly this behavior is very far from what one would expect in an implementation of Mini-ML. In Section 6 we will consider an alternative.

The same phenomenon occurs in the rules for evaluating an application. We will just consider two examples, although there are five rules involved, resulting in quite substantial inefficiency.

eval_app_lam
$$\frac{\vdash e_1 \Rightarrow (\texttt{lambda}\ x\ .\ e_1') \qquad \vdash e_2 \Rightarrow \alpha_2 \qquad \vdash [\,\alpha_2 / x\,]\, e_1' \Rightarrow \alpha}{\vdash e_1\, e_2 \Rightarrow \alpha}$$

eval_app_s
$$\frac{\vdash e_1 \Rightarrow \texttt{s} \qquad \vdash e_2 \Rightarrow \alpha}{\vdash e_1\, e_2 \Rightarrow \texttt{s}\ \alpha}$$

These are implemented as

```
neval_app_lam    : neval (app E1 E2) V
                       <- neval E1 (lam E1')
                       <- neval E2 V2
                       <- neval (E1' V2) V.
neval_app_s      : neval (app E1 E2) (app s V)
                       <- neval E1 s
                       <- neval E2 V.
```

The situation is much as in the previous example: E1 is being applied to E2, and E1 must be evaluated before it is uniquely determined which rule applies. Potentially, this may be done several times to evaluate just a single application. Again we will consider the alternative later.

Embodied in these rules is the decision that we would like to implement a call-by-value semantics for Mini-ML. It would be a simple matter to rewrite the rules to implement a call-by-name discipline instead [6].

Next we will look at a sample evaluation expressed both as a derivation in our formal system and as an Elf deduction. The example expression is (lambda x . lambda y . x y) pred (s z).

(1)	$\vdash$ lambda x . lambda y . x y $\Rightarrow$ lambda x . lambda y . x y	By neval_lam
(2)	$\vdash$ pred $\Rightarrow$ pred	By neval_pred
(3)	$\vdash$ lambda y . pred y $\Rightarrow$ lambda y . pred y	By neval_lam
(4)	$\vdash$ (lambda x . lambda y . x y) pred $\Rightarrow$ lambda y . pred y	By neval_app_lam from 1, 2, 3
(5)	$\vdash$ s $\Rightarrow$ s	By neval_s
(6)	$\vdash$ z $\Rightarrow$ z	By neval_z
(7)	$\vdash$ s z $\Rightarrow$ s z	By neval_app_s from 5, 6
(8)	$\vdash$ (lambda y . pred y) (s z) $\Rightarrow$ pred s z	By neval_app_lam from 3, 7
(9)	$\vdash$ pred (s z) $\Rightarrow$ z	By neval_app_pred_s from 2, 7
(10)	$\vdash$ (lambda x . lambda y . x y) pred (s z) $\Rightarrow$ z	By neval_app_lam from 4, 7, 9

By looking at the structure of the derivation we can see that the Elf deduction for the query

```
?- neval (app (app (lam [x] (lam [y] (app x y))) pred) (app s z)) V.
```

to the program in Appendix C is going to be:

```
10      neval_app_lam
9               (neval_app_pred_s
7                       (neval_app_s
6                               neval_z
5                               neval_s)
2                       neval_pred)
7               (neval_app_s
6                       neval_z
5                       neval_s)
4               (neval_app_lam
3                       neval_lam
2                       neval_pred
1                       neval_lam)
```

where we have marked individual lines of the deduction with the numbers of the corresponding lines in the derivation above. Note that the order of subdeductions is the opposite of what might be expected, due to the use of the left arrow in the formulation of the rule—an artifact that we will explain later. After this example it should be clear how deductions of the evaluation judgment can be viewed as evaluation traces. We will take advantage of this observation in later sections, when we would like to prove properties of Mini-ML by induction over the construction of evaluation traces.

We conclude this subsection with some observations on the correctness of this Elf encoding of Mini-ML. They are presented without proof. Since we will need to talk about the Elf types of programs, we will introduce the notation $\Pi_M \vdash_{LF} M : A$ to mean that the Elf term M has the Elf type A in the context Π_M containing x : exp for each free variable x in M. We tacitly assume α-conversion in the statement of the propositions below and elsewhere in this paper.

Proposition 1 *The encoding* $()^+$ *is a bijection between well-formed Mini-ML expressions and* $\beta\eta$*-equivalence classes of Elf terms* E *such that* $\Pi_M \vdash_{LF} E$: exp , *and also between well-formed Mini-ML types and* $\beta\eta$*-equivalence classes of Elf terms* A *such that* $\vdash_{LF} A$: tp .

Proposition 2 *Let* e *be a well-formed Mini-ML expression and* τ *a well-formed Mini-ML type, and let* E *and* A *be Elf terms such that* $(e)^+ = E$ *and* $(\tau)^+ = A$. *Then there is a bijection between derivations of* $\Pi \vdash e \in \tau$ *and* $\beta\eta$*-equivalence classes of deductions* P *such that* $\overline{\Pi} \vdash_{LF} P$: of E A. *Here for each* $x : \tau$ *in* Π *there are corresponding entries* x : exp *and* p_x : of x $(\tau)^+$ *in* $\overline{\Pi}$. *Furthermore, for a closed, well-typed* e *and the closed, well-typed Mini-ML expression* α *such that* $(\alpha)^+ = V$, *there is a bijection between derivations of* $\vdash e \Rightarrow \alpha$ *and deductions* Q *such that* $\vdash_{LF} Q$: eval E V.

The interested reader is referred to [10] for more formal versions of propositions like the above. These refer to the existence of *compositional* bijections, which are basically those where variables in the object language are represented by variables in the meta-language. Furthermore, they refer to *canonical* terms, which are merely the "obvious" representatives of the $\beta\eta$-equivalence classes above. The following proposition is essentially an operational version of the last proposition above, incorporating Elf's notion of search.

Proposition 3 *For an Elf term E such that $\vdash_{LF} E$: exp, let Q_1 be the Elf query* `?- of` E `T` *and Q_2 be the Elf query* `?- eval` E `V` *(where* `T` *and* `V` *are free and therefore interpreted as logic variables). Then exactly one of the following holds:*

- *Q_1 fails, exactly when E represents an ill-typed Mini-ML expression.*
- *Q_1 and Q_2 both succeed. This is exactly when E is well-typed and according to the semantics of Mini-ML it evaluates to a Mini-ML expression given as a binding to* `V`.
- *Q_1 succeeds and Q_2 fails. This is exactly when E represents a well-typed Mini-ML expression whose value is not defined, but which does not lead to any infinite derivations using the semantic rules.*
- *Q_1 succeeds and Q_2 does not terminate. This is exactly when the represented Mini-ML expression is well-typed but at least one infinite derivation is possible.*

Note that the only finitely failed `eval` queries are those that result in trying to evaluate `pred zero`.

5 Verification of a Simple Property

In this section we will use those features of Elf that correspond to dependent types in LF to verify a simple property of the Mini-ML semantics presented in the previous section. The property is that the result of evaluation is a value, a notion yet to be defined. As usual we will just describe some highlights in detail here, and then present the entire program in Appendix D.

Intuitively, a value is either a constant, an arbitrary λ-abstraction (since we do not evaluate underneath abstractions), or a data constructor applied to values. This inductive definition can be formalized through the following inference rules.

val_t
$$\frac{}{\vdash \texttt{true}\ \mathit{VALUE}}$$

val_f
$$\frac{}{\vdash \texttt{false}\ \mathit{VALUE}}$$

val_z
$$\frac{}{\vdash \texttt{z}\ \mathit{VALUE}}$$

val_s
$$\frac{}{\vdash \texttt{s}\ \mathit{VALUE}}$$

val_pred
$$\frac{}{\vdash \texttt{pred}\ \mathit{VALUE}}$$

val_zerop
$$\frac{}{\vdash \texttt{zerop}\ \mathit{VALUE}}$$

val_pair
$$\frac{\vdash e_1\ \mathit{VALUE} \qquad \vdash e_2\ \mathit{VALUE}}{\vdash \texttt{<}\ e_1\texttt{,}\ e_2\ \texttt{>}\ \mathit{VALUE}}$$

val_lam
$$\frac{}{\vdash \texttt{(lambda}\ x\ \texttt{.}\ e\texttt{)}\ \mathit{VALUE}}$$

val_app_s
$$\frac{\vdash e\ \mathit{VALUE}}{\vdash (\mathtt{s}\ e)\ \mathit{VALUE}}$$

Restricting our attention, for now, to the natural version, this is what we want to prove:

Theorem 1 *For closed, well-typed Mini-ML expressions e and α, if* $\vdash e \Rightarrow \alpha$ *then* $\vdash \alpha\ \mathit{VALUE}$.

Proof: By induction on the structure of a derivation that e evaluates to α. For each inference rule of the operational semantics, we need to construct a derivation that α is a value, given (by induction hypothesis) that the right-hand sides of the premisses of the rules are values.

Most cases are completely straightforward. For example, for the rule eval_lam we have to show that `lambda` x `.` e is a value, but this follows by the rule val_lam in the definition of the value judgment. In the rule for pairing, we simply form the derivation that $\langle \alpha_1, \alpha_2 \rangle$ is a value by applying the inference rule val_pair to the derivations that α_1 and α_2 are values, and which must exist by induction hypothesis.

The cases of destructor functions require one small additional insight. Consider the rule for `fst` e,

eval_fst
$$\frac{\vdash e \Rightarrow \langle \alpha_1, \alpha_2 \rangle}{\vdash \mathtt{fst}\ e \Rightarrow \alpha_1}$$

By induction hypothesis we know that $\langle \alpha_1, \alpha_2 \rangle$ is a value. From this we need to conclude that α_1 is a value. But this follows by exhaustive analysis: the *only* way the derivation of $\vdash \langle \alpha_1, \alpha_2 \rangle\ \mathit{VALUE}$ can end is in an application of the val_pair rule. But then we have the derivation of $\vdash \alpha_1\ \mathit{VALUE}$ as a premiss and that is exactly what was needed. □

What we now seek is an internalization of this argument in the form of a relation. There are a number of ways this can be done—we will discuss two alternatives. It is important to remember, though, that in either case the induction cannot be internalized, and neither can the formal statement of the theorem itself. The goal must be to define a relation in such a way that the proof of the the theorem is "obvious" from its definition. We will have more to say on this point in each example.

5.1 Value Deductions in Elf

The implementation of the value judgment is straightforward.

```
value       : exp -> type.

val_t       : value true.
val_f       : value false.

val_z       : value z.
val_s       : value s.
val_pred    : value pred.
```

```
val_zerop    : value zerop.

val_pair     : value E1 -> value E2 -> value (pair E1 E2).

val_lam      : value (lam E).
val_app_s    : value E -> value (app s E).
```

As an example, consider the deduction

```
(val_pair val_t (val_app_s val_z)) : value (pair true (app s z)).
```

which represents a derivation that $\vdash$ `< true, s z >` *VALUE*.

5.2 Transformation of Evaluations to Value Deductions

The basic idea in this first approach to the partial internalization of Theorem 1 is that the core of the proof construction is a *total* function, transforming evaluation deductions (or traces) into value deductions. This function, while not itself representable in Elf, can be written out as a relation (which, as usual, is implemented as a type family). From the definition of the relation it can be checked easily that the function is in fact total, which yields the proof of the desired theorem.

We begin by defining the type of this desired relation between deductions. It is important, and a unique feature of the LF type system, that we can express that the result of evaluation is the same Mini-ML expression which we deduce to be a value. Here, the power of dependent types is fully exploited.

```
vp        : neval E V -> value V -> type.
```

`E` and `V` are implicitly quantified, and the full type of `vp` would be

```
vp : {E:exp} {V:exp} neval E V -> value V -> type.
```

For practical purposes, it is extremely helpful that the first two arguments (`E` and `V`) can remain implicit. They are, of course, fully determined by the latter two arguments and would thus not add any new information to the clauses defining `vp`, only bulk to the input which quickly becomes unmanageable.

First we consider a base case in the inductive proof, say neval_lam. As a reminder, we have from before

```
neval_lam        : neval (lam E) (lam E).
val_lam          : value (lam E).
```

We now simply write

```
vp_lam           : vp (neval_lam) (val_lam).
```

As one can see, the types work out: the second argument of the type of `neval_lam` and the argument of the type of `val_lam` agree. It also directly expresses our previous informal induction case.

As the second example, let us consider pairing—a case where we make use of the induction hypothesis. Here we have

```
neval_pair      : neval (pair E1 E2) (pair V1 V2)
                        <- neval E1 V1
                        <- neval E2 V2.
val_pair        : value E1 -> value E2 -> value (pair E1 E2).
```

Again, then, this case is not difficult: we combine the value deductions obtained from the recursive calls to vp using the val_pair inference rule:

```
vp_pair   : vp (neval_pair P2 P1) (val_pair VP1 VP2)
               <- vp P1 VP1
               <- vp P2 VP2.
```

Note that because we used the left-arrow notation for neval_pair, the arguments to this constructor appear in reverse of what might be expected. If one rewrites the declaration for neval_pair by reversing the arrow

```
neval_pair : neval E2 V2
             -> neval E1 V1
             -> neval (pair E1 E2) (pair V1 V2).
```

one can see that the deduction VP2 of neval E2 V2 is indeed expected as the first explicit argument to neval_pair. If we tried to repair this by writing

```
neval_pair      : neval (pair E1 E2) (pair V1 V2)
                        <- neval E2 V2
                        <- neval E1 V1.
```

we would obtain a different operational behavior (E2 would be evaluated first). This change in operational behavior would be irrelevant in this case, but, of course, not acceptable in general.

Finally, consider one of the critical cases of destructor functions. Here it will be more difficult to convince oneself that vp does in fact define a function. First, as a reminder, the evaluation rule for fst:

```
neval_fst : neval (fst E) V1 <- neval E (pair V1 V2).
```

We must extract the deduction that V1 is a value from the deduction that (pair V1 V2) is a value. This is done through matching, a technique familiar from other logic programming languages.

```
vp_fst    : vp (neval_fst P) VP1 <- vp P (val_pair VP1 VP2).
```

If we want to check that vp describes a function of its first argument, we have to check that subgoals of the form vp P (val_pair VP1 VP2) will always succeed. This could only fail if the value deduction of the subgoal could end in an inference different from val_pair. By inspection of the rules for value, we can see that this is not possible.

In this manner the definition of the relation can be easily completed (see Appendix D.2 for a complete listing). The computational content of the informal proof is fully present in our implementation, while the induction is currently left to manual inspection. One might imagine, however, that a simple check as required here could be automated.

5.3 Modified Evaluation to Generate Value Deductions

Another approach to the partial internalization of the informal proof is to define a modified evaluation relation, say `veval`, which produces not only a value `V`, but also a deduction that `V` is a value. The desired theorem then follows again by inspection, where we need to check that the new relation succeeds for a given expression `E` iff evaluation of `E` succeeds, and that the value produced would be the same. The complete code for this subsection can be found in Appendix D.3.

The type of our new `veval` relation needs to be something like

```
veval' : exp -> exp -> value E -> type.
```

so that the third argument is required to be a deduction that the expression `E`, from the second argument, is a value. However, as it is the type does not give enough information about `value E`: namely which argument it is a value-deduction of. To make this relation explicit, we use dependent types again. Elf allows us to give an argument a name *when we declare a relation*, in addition to doing so when we define the relation, so we have some way to refer to it when we specify properties of that argument. We replace the `exp ->` component with a corresponding `{ E:exp }` component, obtaining

```
veval : exp -> {E:exp} value E -> type.
```

That is, the first argument is an expression, the second argument is the result of evaluating that expression, and the third is a deduction that the second is a value.

Note that the first form is by no means incorrect, but the fact that our Elf program (to be written down below) is type-correct, would mean much less. With the refined declaration, the Elf type checker is required to check that the relationship between an expression and a deduction that it is a value is in some sense upheld by the program.

Now let us return to the definition of `veval`. The base cases are simple, in that they just return the (atomic) deduction that the base structure is a value. For example

```
veval_t       : veval true true val_t.
```

The deduction for pairs is a straightforward pairing of the deductions for the respective components.

```
veval_pair    : veval (pair E1 E2) (pair V1 V2) (val_pair P1 P2)
                      <- veval E1 V1 P1
                      <- veval E2 V2 P2.
veval_fst     : veval (fst E) V1 P1
                      <- veval E (pair V1 V2) (val_pair P1 P2).
```

In the second case of `fst`, we again use matching to extract a deduction `P1` that `V1` is a value.

The subcases of application are straightforward either because they produce a base deduction,

```
veval_app_zerop_t : veval (app E1 E2) true val_t
                          <- veval E1 zerop P1
                          <- veval E2 z P2.
```

or take the deduction obtained from normalization

```
veval_app_lam    : veval (app E1 E2) V P
                          <- veval E1 (lam E1') P1
                          <- veval E2 V2 P2
                          <- veval (E1' V2) V P.
```

Likewise, the rules for `let` and `letrec` just return the deduction obtained by normalizing the body, discarding the deduction obtained by normalizing the bound term.

```
veval_let     : veval (let E1 E2) V P2
                      <- veval E1 V' P1
                      <- veval (E2 V') V P2.

veval_letrec  : veval (letrec E1 E2) V P2
                      <- veval (fix E1) V' P1
                      <- veval (E2 V') V P2.
```

Now we need to consider the connection between the relation `veval` as defined here and Theorem 1. We should start by formalizing what we claim the relation does:

Proposition 4 *For all Elf terms E and V that represent well-typed Mini-ML expressions, there exists a P such that $\vdash_{LF} P$: `neval` $E\ V$ if and only if there exist VP and Q such that $\vdash_{LF} VP$: `veval` $E\ V\ Q$ and $\vdash_{LF} Q$: `value` V .*

Proof: The right–to–left direction is obtained trivially by erasing the third argument of occurrences of `veval`, making them occurrences of `neval`. The left–to–right direction is analogous to the induction argument in the proof of the previous theorem. □

Now that we have some formal understanding of what is achieved by this style of programming in Elf, we also need to obtain a pragmatic, or methodological, understanding of this programming style. In particular, we need to know just what is guaranteed when an Elf program like the above is found to be well-typed. Essentially, each rule corresponds to one case in the case analysis of an inductive correctness proof. Elf type-correctness is, at a certain level, a guarantee that these individual cases are correct. However, it does not guarantee that the induction argument holds together. Pragmatically, this means that if the correctness argument embedded in a single rule like those above is incorrect, Elf type inference will return an error message. Let us consider some examples.

The rule `veval_t` that we saw earlier asserts that that `val_t` really *is* a deduction of `(value true)`. On the other hand, `veval_pair` asserts that an expression of the form `(pair E1 E2)`, if it is indeed a value, will indeed have a value deduction of the form `(val_pair P1 P2)`, where `P1` is the deduction that `E1` is a value and `P2` is the deduction that `E2` is a value, if the evaluations of `E1` and `E2` terminate. Now consider adding some incorrect rules. For example, adding

```
veval_wrong : veval X X of_t.
```

will result in an Elf type error. This is because `X` would have to be the constant `true` for the rule to be well-typed, but `X` is implicitly universally (not existentially!) quantified. However, if the rule were

```
veval_wrong : veval X _ of_t.
```

the Elf front-end would have been free to replace the underscore with `true`, and there would not have been an error, resulting instead in

```
veval_wrong : {X:exp} veval X true of_t.
```

The check that evaluation traces can be reconstructed from `veval` deductions would now fail, since there is no corresponding rule of type `neval X true`. This cannot be discovered by the Elf type checker, however. Similarly, the rule

```
veval_bogus : veval X X Y.
```

will become, after type reconstruction, the rule

```
veval_bogus : {X : exp} {Y : value X} veval X X Y.
```

and there is no type error. This is because the Elf front-end is at liberty to make the necessary restrictions on the types of quantified variables. Of course, the rule is still quite wrong, one problem being the the generated value deduction will not always be ground. If we were to run the so-augmented `veval` relation on something that was not a value, such as

```
?- veval (app (lam [x] x) z) V P.
```

the behavior would depend on the way `value` was interpreted. If it was defined as a *dynamic* relation (see Section 8) these rules would be used to check whether the argument was a value, and if appropriate, would cause backtracking, and so the program would (accidentally) run correctly. However, if it were defined as a *static* type family the rules would not be used operationally at runtime, and a deduction obligation would remain until the end of the computation, and be printed as part of the answer.

This tells us that it is worthwhile to look at the rules output by Elf upon parsing—type-checking alone does not guarantee program correctness, though the properties which can be guaranteed go well beyond what is possible in languages without dependent types.

6 An Equivalent Algorithmic Semantics

Here we give an equivalent formulation of the Mini-ML operational semantics which avoids backtracking, and hence is referred to as "algorithmic". We then discuss how this would affect the material of the previous section, and show that the two formulations of the semantics are equivalent.

6.1 Algorithmic Operational Semantics

While this formulation may be semantically somewhat obscure, being deterministic it is more efficient and closer to a practical interpreter. For this reason it is more useful to be able to verify its properties, and also it is complicated enough that it has interesting properties to verify.

The basic idea is that for those situations where the previous versions required (deep) backtracking based on the result of an evaluation, we compute the result in advance and then call another relation to handle that case, using only shallow backtracking. The full program appears in Appendix E.

In addition to the basic evaluation relation, here `aeval`, we need to declare two new relations for conditionals and applications:

```
aeval    : exp -> exp -> type.
doif     : exp -> exp -> exp -> exp -> type.
app_vals : exp -> exp -> exp -> type.
```

so that when the appropriate branch is evaluated the deduction that the result is a value may be obtained as well. Of course only those `eval` rules that relate to conditionals or application are affected.

For application it suffices to evaluate both sides, and pass both the values and the deductions that they are values into `avapp_vals`:

```
aveval_app      : aveval (app E1 E2) V P
                          <- aveval E1 V1 P1
                          <- aveval E2 V2 P2
                          <- avapp_vals V1 P1 V2 P2 V P.
avapp_v_s       : avapp_vals s val_s V2 P2 (app s V2) (val_app_s P2).
```

Conditionals are similar, although the deduction for the boolean term is discarded. The definition of `doif` merely needs to return the deduction that is produced by the call to `aveval`.

```
aveval_if       : aveval (if E1 E2 E3) V P
                          <- aveval E1 V1 P1
                          <- avdoif V1 E2 E3 V P.

avdoif_t        : avdoif true E2 E3 V P <- aveval E2 V P.
avdoif_f        : avdoif false E2 E3 V P <- aveval E3 V P.
```

The other cases are obvious incarnations of ideas previously discussed. We leave it to the interested reader to verify that this program achieves as much as the natural version.

6.3 Equivalence of Natural and Algorithmic Evaluation

So far we have claimed that the two formulations of evaluation are equivalent. This is expressed in the following proposition.

Proposition 5 *For all Elf terms E and V that represent well-typed Mini-ML expressions, there exists a P such that $\vdash_{LF} P$:* `neval` *E V if and only if there exists a Q such that $\vdash_{LF} Q$:* `aeval` *E V*

Proof: In each direction, by induction on the structure of deductions. □

We can show the equivalence by defining an Elf equality relation `na` thus:

```
na : neval E V -> aeval E V -> type.
```

where `neval` is the natural version of the evaluation relation, and `aeval` is the algorithmic version. That is, the first argument is a deduction corresponding to the natural semantics, and the second is a deduction corresponding to the algorithmic semantics. Dependent types are used to express that both deductions must end in the equivalent judgment of evaluation. The program we define will be executable in both directions. That is, it will be able to translate either kind of deduction into the other.

Most of the rules are very simple, with base cases like

```
na_t            : na neval_t aeval_t.
```

and simple recursive rules like

```
na_pair        : na (neval_pair P2 P1) (aeval_pair Q2 Q1)
                    <- na P1 Q1
                    <- na P2 Q2.
```

but the rules that correspond to differences between the two semantics are slightly more interesting. For example, the rules dealing with application, such as

```
na_app_lam     : na (neval_app_lam P3 P2 P1)
                    (aeval_app (app_v_lam Q3) Q2 Q1)
                    <- na P1 Q1
                    <- na P2 Q2
                    <- na P3 Q3.
```

are all similar, as are those dealing with conditionals, such as

```
na_if_t        : na (neval_if_t P2 P1) (aeval_if (doif_t Q2) Q1)
                    <- na P1 Q1
                    <- na P2 Q2.
```

Essentially, these rules just need to include the extra step taken in the algorithmic semantics. It is obvious that such a translation is total on deductions for the natural semantics. It should also be clear that it is total for the algorithmic case because the particular compositions of rules are the only ones possible. The full program appears in Appendix F.

7 The Subject Reduction Property

In this section we take on a more challenging task: verifying the subject reduction property for Mini-ML. In this context, the subject reduction property (sometimes referred to as soundness of the type system with respect to an operational semantics) says that if a well-typed expression e of type τ evaluates to α then α also has type τ.

Again, there are at least two possible approaches to the internalization of this theorem. We will discuss them in turn, after sketching the informal proof of this theorem.

Theorem 2 *Let e and α be Mini-ML expressions and τ a Mini-ML type such that $\vdash e \in \tau$ and $\vdash e \Rightarrow \alpha$. Then $\vdash \alpha \in \tau$.*

Proof: By induction on the structure of the derivation that e evaluates to α. For each case we have to construct a derivation that α has type τ, employing the induction hypothesis applied to the premisses of the derivation of $\vdash e \Rightarrow \alpha$. We consider a few typical cases.

eval_t

$$\frac{}{\vdash \texttt{true} \Rightarrow \texttt{true}}$$

In this case we have to show that for every type τ and derivation of $\vdash \texttt{true} \in \tau$, there exists a derivation of $\vdash \texttt{true} \in \tau$. Of course, we can either use the same derivation, or use the initial derivation of_t. These two alternatives will lead to two different, but equivalent programs in Elf.

eval_pair
$$\frac{\vdash e_1 \Rightarrow \alpha_1 \qquad \vdash e_2 \Rightarrow \alpha_2}{\vdash \langle e_1, e_2 \rangle \Rightarrow \langle \alpha_1, \alpha_2 \rangle}$$

From the induction hypothesis we conclude that every type of e_1 is also a type of α_1, and every type of e_2 is also a type of α_2. A type deduction of $\langle e_1, e_2 \rangle$ must end in the of_pair rule, where the premisses show that e_1 has type τ_1 and e_2 has type τ_2. Hence α_1 also has type τ_1 and α_2 has type τ_2 and we can apply the typing rule of_pair to construct a deduction of $\vdash \langle \alpha_1, \alpha_2 \rangle \in \tau_1 * \tau_2$, which is what we needed to show in this case.

The crucial and most difficult case is the `let`. This is because in the operational semantics rule

eval_let
$$\frac{\vdash e_1 \Rightarrow \alpha_1 \qquad \vdash [\alpha_1 / x] e_2 \Rightarrow \alpha_2}{\vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Rightarrow \alpha_2}$$

e_1 is evaluated only once, while in the type derivation (which must end in the of_let rule) types for e_1 might be calculated many times.

of_let
$$\frac{\Pi \vdash e_1 \in \tau_1 \qquad \Pi \vdash [e_1 / x] e_2 \in \tau_2}{\Pi \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \in \tau_2}$$

Thus, in order to construct the derivation that $[\alpha_1 / x] e_2$ has type τ_2 (which is necessary to apply the induction hypothesis to the rightmost premiss of the application of eval_let), we need to transform the deduction of $[e_1 / x] e_2$ and substitute a fresh derivation that α_1 has whatever type was inferred for e_1 at each occurrence of x in e_2. This is straightforward, but tedious. □

To see that the complicated substitution in the case of `let` is really necessary, consider the Mini-ML expression

```
let id = (lambda x . x) in < id true , id zero >
```

Here we need to assign two different types to `x` at the two different occurrences of `id` in the body of the `let`: `Bool` at the first occurrence, and `Nat` at the second occurrence. In the absence of type schemas, the only way this can be achieved is by explicitly type checking `(lambda x . x)` twice.

Note that in the proof above we strongly rely on the fact that for each expression constructor, there is exactly one typing rule which applies. This is so, even though a type of an expression is not unique (for example, `lambda` x `.` x has type `Nat -> Nat`, `Bool -> Bool`, and infinitely many others).

7.1 Transformation of Evaluation Traces and Type Deductions

In the first partial internalization of this proof, we define a relation `sr` implementing the function which transforms an evaluation trace of $\vdash e \Rightarrow \alpha$ and a type derivation of $\vdash e \in \tau$ to a derivation of $\vdash \alpha \in \tau$. The complete code for this relation can be found in Appendix G.1. From our intended functionality above, we can directly read off the appropriate type declaration for `sr`.

```
sr : neval E V -> of E A -> of V A -> type.
```

The base case of the informal proof above is easy. In fact, there are two different realizations, depending on which proof variant we prefer.

```
sr_t          : sr (neval_t) (of_t) (of_t).
sr_t'         : sr (neval_t) D D.
```

The inductive case for pairing is similarly straightforward

```
sr_neval_pair : sr (neval_pair P2 P1) (of_pair D2 D1) (of_pair C2 C1)
                   <- sr P1 D1 C1
                   <- sr P2 D2 C2.
```

Note that this describes a total function only because there is only one typing rule for an expression which is a pair, namely of_pair (recall that the first two arguments to `sr_neval_pair` are inputs).

The rule for `fix` is interesting, because of the way the type derivation for the recursive call is constructed:

```
sr_neval_fix     : sr (neval_fix P) (of_fix D) C
                       <- sr P (D (fix E) (of_fix D)) C.
```

As in the informal proof, the case for `let` is the hardest. Recall, that our formulation of the rule for the typing of `let` differed from the rule used in the presentation in Section 2.

```
of_let     : of (let E1 E2) A2
                <- of E1 A1
                <- {x:exp} ({A:tp} of x A <- of E1 A)
                              -> of (E2 x) A2.
```

This will now help us in defining the auxiliary substitution predicate which performs the operation described in the informal proof above. This auxiliary prediate must also be directly applicable in the case of `letrec`. We have

```
sbst : ({x:exp} ({A:tp} of E1 A -> of x A) -> of (E2 x) A2)
       -> neval E1 V1
       -> of (E2 V1) A2
       -> type.

sr_neval_let  : sr (neval_let P2 P1) (of_let D2 D1) C
                   <- sbst D2 P1 C2
                   <- sr P2 C2 C.
```

Assume we are given an expression `let` $x = e_1$ `in` e_2, represented as `let E1 E2`, with `E2` a function from expressions to expressions. The first argument of `sbst` is a type derivation `D` of `E2` under the appropriate assumption that any type of `E1` can be used as a type of the bound variable `x`. The second argument is the evaluation trace for `E1`. The third argument (the output) is a derivation showing that the result of substituting `V1` for `x` in `E2` has the same type as `E2`. This is achieved by copying `D`, except where the assumption about the possible types of `x` was used. In each of those places we recursively convert the evaluation trace of `E1` to a deduction showing that `V1` also must have type `A2`. In Elf, this last and critical case is expressed by

```
sbst_var   : sbst ([x:exp] [d:{A:tp} of E A -> of x A] d A D) P C
               <- sr P D C.
```

All other cases are mere congruence cases, descending into the subcomponents where the necessary substitutions might eventually be performed. An exposition of this general technique can be found in [13] in the context of the simply-typed λ-calculus. These ideas can be applied here, too. We discuss a couple of cases.

```
sbst_t     : sbst ([x] [d] of_t) P of_t.
```

In the base case, the assumption `d` is not used, and we simply return the deduction `of_t` unchanged. In the cases not involving binding operators, we descend into the components and reconstruct a deduction with the same final inference rule.

```
sbst_if    : sbst ([x] [d] of_if (D3 x d) (D2 x d) (D1 x d)) P
               (of_if C3 C2 C1)
             <- sbst D1 P C1
             <- sbst D2 P C2
             <- sbst D3 P C3.
```

Any of the components might contain uses of the assumption `d`, which means that each subdeduction must explicitly be allowed to depend on `x` and `d`. This also makes the recursion possible (since the first argument to `sbst` must have this functional type).

In the cases where a binding operator is involved, we need to make an assumption which indicates how to substitute for the bound variable. In this particular application of this programming technique, bound variables are not affected by the `sbst` predicate, which means that they copy to themselves, expressed by the assumption `(sbst ([x] [d] d') P d')` below.

```
sbst_lam   : sbst ([x] [d] of_lam (D x d)) P (of_lam C)
               <- {x':exp} {d':of x' A}
                    sbst ([x] [d] d') P d'
                    -> sbst ([x] [d] D x d x' d')
                            P (C x' d').
```

Another point which is easily missed is that the resulting deduction `C` may also depend on the bound variables (`d'`, in this case) and in fact does to the same extend that `D` depends on `d'`. Therefore we must abstract `C` over `x'` and `d'` when the computation of `C` from `D` is complete.

Again the question arises to what extend this does encapsulate the informal subject reduction proof. First of all, the operational content of the informal proof is fully preserved: we can use the predicate to compute a type deduction of the value α. Secondly, the overall induction is missing as before, and we need to convince ourselves that we do indeed describe a function of the first two arguments. This is straightforward on the first argument (one case for each inference rule for evaluation). For the second argument, one has to observe that for each possible expression constructor, there is only one possible typing rule which could be applied. This pleasant property of the Mini-ML typing rules, as we presented them, is also an important reason for avoiding explicit type schemas in the typing rules.

7.2 Evaluation of Type Deductions

An alternative expression of the computational contents of the subject reduction proof is to take advantage of the fact that expressions and type deductions stand (almost) in one-to-one correspondence and evaluate type derivations instead of expressions. However, the correspondence is not close enough to let this go through easily: even though expressions and the type deductions correspond structurally, the types which occur in them may differ (since types are *not* uniquely determined by an expression). Therefore, we cannot circumvent the explicit substitution we needed above—it only appears in a slightly different guise.

The two predicates we need are `sr_eval` and `esubst`:

```
sr_eval     : of E A -> of V A -> type.
esubst      : ({x:exp} ({A:tp} of E1 A -> of x A) -> of (E2 x) A2)
              -> of (E2 V) A2
              -> type.
```

Now the relationship between this and the predicate `sr` also is clear: `sr` has one more argument than `sr_eval`, namely a guiding evaluation trace of `E`. Here we just perform the evaluation, and leave the relationship to an evaluation trace for `E` implicit.

So `sr_eval` expresses the relationship between a deduction and a normalized deduction, and `esubst` uses the same method as before to translate one deduction to another. The intent is that there is a closed term `SR` such that

```
SR : sr_eval (P:of E A) (Q:of V A)
```

iff there is a closed term `NE` such that

```
NE : neval E V
```

Interestingly, this relationship could again be made explicit in Elf! This is the first application of relations between relations among deductions we know of. Since it would lead us too far afield, let us just write out the declaration of such a predicate, since it introduces a useful technique. We would like to write something like

```
srne : sr_eval P Q -> neval E V -> type.
```

but we lose too much information: `P` should be a type deduction of `E` and `Q` should be a type deduction of `V`. These arguments to `sr_eval` are implicit, but can be recovered by using explicit type annotation. Thus we can write

```
srne : sr_eval (P:of E A) (Q:of V A) -> neval E V -> type.
```

and now all the relationships are expressed through the type.

Now let us only briefly consider how `sr_eval` is to be defined. Because of the similarity of the techniques to previously presented material, we restrict ourselves to simply stating a few typical cases. The complete code can be found in Appendix G.2.

```
sr_eval_t          : sr_eval of_t of_t.
sr_eval_if_f       : sr_eval (of_if P3 P2 P1) Q3
                             <- sr_eval P1 of_f
```

```
                                    <- sr_eval P3 Q3.

sr_eval_let         : sr_eval (of_let P2 P1) Q
                                    <- sr_eval P1 Q1
                                    <- esubst P2 Q2
                                    <- sr_eval Q2 Q.
```

Then `esubst`, the auxiliary substitution predicate, is almost identical in form to the `subst` predicate in the first formulation of subject reduction. But instead of passing along an evaluation trace to be converted in the base case, we simply evaluate the type derivation which we find at the places where the bound variable occurs in the body of the `let` expression:

```
esubst_var       : esubst ([x] [p] p A P1) Q1
                       <- sr_eval P1 Q1.
```

8 Example Queries

As discussed in Section 3, a type can play different roles. One role is as a constraint on the instantiation of variables during unification. In our example, `exp` might play this role. The other is that of a goal, for which we are trying to find a deduction. The type `(of (pair z z) A)` from the query `?- of (pair z z) A` is an example. In order to obtain a reasonable operational semantics for Elf, the interpreter needs to know which interpretation we would like to assign to each type and type family. We call the types and type families which are interpreted as predicates and thus give rise to goals *dynamic*. Thus, in our example, `of` would be *dynamic*, while `nat` and `exp` would be *static*.

The distinction of *static* and *dynamic* families is not a property of the declaration itself, but indicates how we would like to use a declaration. For example, we could declare the type `tp` as *dynamic* and then pose the query

```
?- tp.
```

and obtain the first answer `bool`. The declarations

```
bool  : tp.
nat   : tp.
cross : tp -> tp -> tp.
arrow : tp -> tp -> tp.
```

can now be understood as a named implementation of the Prolog clauses

```
tp.
tp.
tp :- tp, tp.
tp :- tp, tp.
```

for the predicate `tp`. Since we can exploit the explicit representation of *deductions*, this is not completely trivial or meaningless. Under this interpretation `bool` would be considered a deduction of `tp`.

Finally, let us look at example queries for the programs presented here. Because we restrict ourselves to the core language in this paper, we need a mechanism for staged evaluation of queries. We use a built-in special predicate `sigma` for this purpose, which is inspired by the use of Σ types in various extensions of the LF type theory. Solving a query of the form

```
?- sigma [X:A] (B X)
```

will first solve `A` by finding an appropriate instantiation for `X` and then solve `(B X)`. The corresponding proof object is a pair consisting of a witness (the instantiation term for `X`) and the term of type `(B X)`.

The addition function was used as an example in Section 4. Here is the concrete version of this function applied to the representations of 2 and 1:

```
(app (app (letrec
            ([add] (lam [x] (lam [y]
                      (if (app zerop x)
                          y
                          (app s (app (app add (app pred x)) y))))))
            ([add] add))
          (app s (app s z)))
      (app s z))
```

A query to synthesize its type would be formulated as

```
?- of (app (app (letrec
                  ([add] (lam [x] (lam [y]
                            (if (app zerop x)
                                y
                                (app s (app (app add (app pred x)) y))))))
                  ([add] add))
                (app s (app s z)))
          (app s z))
      A.
```

and instantiates `A` to `nat` as its only solution. Evaluation can be achieved with

```
?- aeval (app (app (letrec
                     ([add] (lam [x] (lam [y]
                               (if (app zerop x)
                                   y
                                   (app s (app (app add (app pred x)) y))))))
                     ([add] add))
                   (app s (app s z)))
             (app s z))
        V.
```

(the "natural" version `neval` is not recommended for this example).

To put all these together consider the following sequence: we generate a type deduction `D` for the expression above (call it `E`) and also evaluate `E` algorithmically to obtain `Q`. This is then translated to a natural evaluation trace `P`. Then we apply subject reduction to `D` and `P` to obtain a type deduction for the result of evaluation (which is `(app s (app s (app s z)))` and has type `nat`). This whole sequence can be executed with the query

```
?- sigma [D:of (app (app (letrec
                     ([add] (lam [x] (lam [y]
```

```
                                (if (app zerop x)
                                    y
                                    (app s (app (app add (app pred x)) y))))))
                      ([add] add))
                      (app s (app s z)))
                   (app s z)) A]
      sigma [Q:aeval _ V]
         sigma [NA:na P Q]
            sr P D C.
```

The underscore in the type of `Q` is determined by type reconstruction to be the same expression as the one whose type we determine. This query indeed generated the expected substitutions which can be found in full in Appendix H. Here we abbreviate some large subterms by `%%`.

```
C <- of_app (of_app (of_app of_z of_s) of_s) of_s ,
P <- neval_app_lam %% %% %% ,
V <- app s (app s (app s z)) ,
A <- nat .
```

9 Concluding Remarks

We have demonstrated a methodology for applying Elf to expressing and reasoning about the natural semantics of a programming language — in this case Mini-ML. In doing so, we have extended the now well-known methodology for representing operational semantics. All the examples in the paper have been tested with the current prototype implementation of Elf.

Many tasks remain to be done to complete this line of research. In [8], Hannan and Miller systematically transform a high-level description of a language related to Mini-ML in terms of inference rules into two low-level abstract machines. We believe that most, if not all of these transformations should be representable in Elf as judgments on deductions, and this should be investigated.

More fundamental is the question how to achieve the *complete* formalization of the meta-theoretic properties of languages implemented in Elf. We have begun to investigate such possibilities based on the preliminary design of a module system for Elf [11].

Acknowledgements

We are grateful to Bob Harper for commenting on some preliminary drafts. We also wish to acknowledge the Max-Planck-Institute for Computer Science in Saarbrücken, where the second author completed the paper during an extended visit.

A Expressions and Types of Mini-ML

```
exp    : type.

true   : exp.
false  : exp.
if     : exp -> exp -> exp -> exp.
```

```
z       : exp.
s       : exp.
pred    : exp.
zerop   : exp.

pair    : exp -> exp -> exp.
fst     : exp -> exp.
snd     : exp -> exp.

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

let     : exp -> (exp -> exp) -> exp.

letrec  : (exp -> exp) -> (exp -> exp) -> exp.
fix     : (exp -> exp) -> exp.

tp      : type.

bool    : tp.
nat     : tp.
cross   : tp -> tp -> tp.
arrow   : tp -> tp -> tp.
```

B Typing Rules

```
of  : exp -> tp -> type.

of_t        : of true bool.
of_f        : of false bool.
of_if       : of (if E1 E2 E3) A
                  <- of E1 bool
                  <- of E2 A
                  <- of E3 A.

of_z        : of z nat.
of_s        : of s (arrow nat nat).
of_pred     : of pred (arrow nat nat).

of_zerop    : of zerop (arrow nat bool).

of_pair     : of (pair E1 E2) (cross A1 A2)
                  <- of E1 A1
                  <- of E2 A2.
of_fst      : of (fst E) A1
                  <- of E (cross A1 A2).
```

```
of_snd     : of (snd E) A2
               <- of E (cross A1 A2).

of_lam     : of (lam E) (arrow A1 A2)
               <- {x:exp} of x A1 -> of (E x) A2.
of_app     : of (app E1 E2) A1
               <- of E1 (arrow A2 A1)
               <- of E2 A2.

of_let     : of (let E1 E2) A2
               <- of E1 A1
               <- {x:exp} ({A:tp} of x A <- of E1 A)
                            -> of (E2 x) A2.

of_letrec  : of (letrec E1 E2) A2
               <- of (fix E1) A1
               <- {x:exp} ({A:tp} of x A <- of (fix E1) A)
                            -> of (E2 x) A2.

of_fix     : of (fix E) A
               <- {x:exp} of x A -> of (E x) A.
```

C Natural Operational Semantics

```
neval      : exp -> exp -> type.

neval_t          : neval true true.
neval_f          : neval false false.
neval_if_t       : neval (if E1 E2 E3) V
                        <- neval E1 true
                        <- neval E2 V.
neval_if_f       : neval (if E1 E2 E3) V
                        <- neval E1 false
                        <- neval E3 V.

neval_z          : neval z z.
neval_s          : neval s s.
neval_pred       : neval pred pred.

neval_zerop      : neval zerop zerop.

neval_pair       : neval (pair E1 E2) (pair V1 V2)
                        <- neval E1 V1
                        <- neval E2 V2.
neval_fst        : neval (fst E) V1
                        <- neval E (pair V1 V2).
neval_snd        : neval (snd E) V2
```

```
                              <- neval E (pair V1 V2).

neval_lam          : neval (lam E) (lam E).

neval_app_lam      : neval (app E1 E2) V
                              <- neval E1 (lam E1')
                              <- neval E2 V2
                              <- neval (E1' V2) V.
neval_app_s        : neval (app E1 E2) (app s V)
                              <- neval E1 s
                              <- neval E2 V.
neval_app_pred_s   : neval (app E1 E2) V
                              <- neval E1 pred
                              <- neval E2 (app s V).
neval_app_zerop_t  : neval (app E1 E2) true
                              <- neval E1 zerop
                              <- neval E2 z.
neval_app_zerop_f  : neval (app E1 E2) false
                              <- neval E1 zerop
                              <- neval E2 (app s V).

neval_let          : neval (let E1 E2) V2
                              <- neval E1 V1
                              <- neval (E2 V1) V2.

neval_letrec       : neval (letrec E1 E2) V
                              <- neval (fix E1) V1
                              <- neval (E2 V1) V2.

neval_fix          : neval (fix E) V <- neval (E (fix E)) V.
```

D The Value Property and Evaluation

D.1 The Value Property

```
value          : exp -> type.

val_t          : value true.
val_f          : value false.

val_z          : value z.
val_s          : value s.
val_pred       : value pred.

val_zerop      : value zerop.

val_pair       : value E1 -> value E2 -> value (pair E1 E2).
```

```
val_lam      : value (lam E).
val_app_s    : value E -> value (app s E).
```

D.2 Transformation of Evaluations to Value Deductions

```
vp         : neval E V -> value V -> type.

vp_t       : vp (neval_t) val_t.
vp_f       : vp (neval_f) val_f.
vp_if_t    : vp (neval_if_t P2 P1) VP2 <- vp P2 VP2.
vp_if_f    : vp (neval_if_f P3 P1) VP3 <- vp P3 VP3.

vp_z       : vp (neval_z) val_z.
vp_s       : vp (neval_s) val_s.
vp_pred    : vp (neval_pred) val_pred.

vp_zerop   : vp (neval_zerop) val_zerop.

vp_pair    : vp (neval_pair P2 P1) (val_pair VP1 VP2)
               <- vp P1 VP1
               <- vp P2 VP2.

vp_fst     : vp (neval_fst P) VP1 <- vp P (val_pair VP1 VP2).
vp_snd     : vp (neval_snd P) VP2 <- vp P (val_pair VP1 VP2).

vp_lam     : vp (neval_lam) val_lam.

vp_app_lam     : vp (neval_app_lam P3 P2 P1) VP3
                   <- vp P3 VP3.
vp_app_s       : vp (neval_app_s P2 P1) (val_app_s VP2)
                   <- vp P2 VP2.

vp_app_pred_s : vp (neval_app_pred_s P2 P1) VP0
                   <- vp P2 (val_app_s VP0).

vp_app_zerop_t : vp (neval_app_zerop_t P2 P1) val_t.
vp_app_zerop_f : vp (neval_app_zerop_f P2 P1) val_f.

vp_let     : vp (neval_let P2 P1) VP <- vp P2 VP.

vp_letrec : vp (neval_letrec P2 P1) VP <- vp P2 VP.

vp_fix     : vp (neval_fix P) VP <- vp P VP.
```

D.3 Modified Evaluation to Generate Value Deductions

```
veval : exp -> {E:exp} value E -> type.

veval_t       : veval true true val_t.
veval_f       : veval false false val_f.
veval_if_t    : veval (if E1 E2 E3) V P2
                        <- veval E1 true P1
                        <- veval E2 V  P2.
veval_if_f    : veval (if E1 E2 E3) V P3
                        <- veval E1 false P1
                        <- veval E3 V  P3.

veval_z       : veval z z val_z.
veval_s       : veval s s val_s.
veval_pred    : veval pred pred val_pred.

veval_zerop   : veval zerop zerop val_zerop.

veval_pair    : veval (pair E1 E2) (pair V1 V2) (val_pair P1 P2)
                        <- veval E1 V1 P1
                        <- veval E2 V2 P2.
veval_fst     : veval (fst E) V1 P1
                        <- veval E (pair V1 V2) (val_pair P1 P2).
veval_snd     : veval (snd E) V2 P2
                        <- veval E (pair V1 V2) (val_pair P1 P2).

veval_lam     : veval (lam E) (lam E) val_lam.

veval_app_lam    : veval (app E1 E2) V P
                        <- veval E1 (lam E1') P1
                        <- veval E2 V2 P2
                        <- veval (E1' V2) V P.
veval_app_s      : veval (app E1 E2) (app s V) (val_app_s P2)
                        <- veval E1 s P1
                        <- veval E2 V P2.
veval_app_pred_s : veval (app E1 E2) V P2
                        <- veval E1 pred P1
                        <- veval E2 (app s V) (val_app_s P2).
veval_app_zerop_t : veval (app E1 E2) true val_t
                        <- veval E1 zerop P1
                        <- veval E2 z P2.
veval_app_zerop_f : veval (app E1 E2) false val_f
                        <- veval E1 zerop P1
                        <- veval E2 (app s V) P2.

veval_let     : veval (let E1 E2) V P2
```

```
                          <- veval E1 V' P1
                          <- veval (E2 V') V P2.

veval_letrec  : veval (letrec E1 E2) V P2
                          <- veval (fix E1) V' P1
                          <- veval (E2 V') V P2.

veval_fix     : veval (fix E) V P
                          <- veval (E (fix E)) V P.
```

D.4 Modified Algorithmic Evaluation to Generate Value Deductions

```
aveval : exp -> {E:exp} value E -> type.
avapp_vals : {X:exp} value X
             -> {Y:exp} value Y -> {Z:exp} value Z -> type.
avdoif : exp -> exp -> exp -> {E:exp} value E -> type.

aveval_t       : aveval true true val_t.
aveval_f       : aveval false false val_f.
aveval_if      : aveval (if E1 E2 E3) V P
                          <- aveval E1 V1 P1
                          <- avdoif V1 E2 E3 V P.

aveval_z       : aveval z z val_z.
aveval_s       : aveval s s val_s.
aveval_pred    : aveval pred pred val_pred.

aveval_zerop   : aveval zerop zerop val_zerop.

aveval_pair    : aveval (pair E1 E2) (pair V1 V2) (val_pair P1 P2)
                          <- aveval E1 V1 P1
                          <- aveval E2 V2 P2.
aveval_fst     : aveval (fst E) V1 P1
                          <- aveval E (pair V1 V2) (val_pair P1 P2).
aveval_snd     : aveval (snd E) V2 P2
                          <- aveval E (pair V1 V2) (val_pair P1 P2).

aveval_lam     : aveval (lam E) (lam E) val_lam.

aveval_app     : aveval (app E1 E2) V P
                          <- aveval E1 V1 P1
                          <- aveval E2 V2 P2
                          <- avapp_vals V1 P1 V2 P2 V P.

aveval_let     : aveval (let E1 E2) V P2
                          <- aveval E1 V' P1
                          <- aveval (E2 V') V P2.
```

```
aveval_letrec  : aveval (letrec E1 E2) V P2
                       <- aveval (fix E1) V' P1
                       <- aveval (E2 V') V P2.

aveval_fix     : aveval (fix E) V P
                       <- aveval (E (fix E)) V P.

avdoif_t       : avdoif true E2 E3 V P <- aveval E2 V P.
avdoif_f       : avdoif false E2 E3 V P <- aveval E3 V P.

avapp_v_lam     : avapp_vals (lam E1) val_lam V2 P2 V P
                       <- aveval (E1 V2) V P.
avapp_v_s       : avapp_vals s val_s V2 P2 (app s V2) (val_app_s P2).
avapp_v_zerop_t : avapp_vals zerop val_zerop z val_z true val_t.
avapp_v_zerop_f : avapp_vals zerop val_zerop (app s V2) (val_app_s P2)
                             false val_f.
avapp_v_pred_s  : avapp_vals pred val_pred (app s V2) (val_app_s P2)
                             V2 P2.
```

E Algorithmic Operational Semantics

```
aeval    : exp -> exp -> type.
doif     : exp -> exp -> exp -> exp -> type.
app_vals : exp -> exp -> exp -> type.

aeval_t          : aeval true true.
aeval_f          : aeval false false.
aeval_if         : aeval (if E1 E2 E3) V
                       <- aeval E1 V1
                       <- doif V1 E2 E3 V.

aeval_z          : aeval z z.
aeval_s          : aeval s s.
aeval_pred       : aeval pred pred.

aeval_zerop      : aeval zerop zerop.

aeval_pair       : aeval (pair E1 E2) (pair V1 V2)
                       <- aeval E1 V1
                       <- aeval E2 V2.
aeval_fst        : aeval (fst E) V1
                       <- aeval E (pair V1 V2).
aeval_snd        : aeval (snd E) V2
                       <- aeval P (pair V1 V2).

aeval_lam        : aeval (lam E) (lam E).
```

```
aeval_app        : aeval (app E1 E2) V
                       <- aeval E1 V1
                       <- aeval E2 V2
                       <- app_vals V1 V2 V.

aeval_let        : aeval (let E1 E2) V2
                       <- aeval E1 V1
                       <- aeval (E2 V1) V2.

aeval_letrec     : aeval (letrec E1 E2) V2
                       <- aeval (fix E1) V1
                       <- aeval (E2 V1) V2.

aeval_fix        : aeval (fix E) V <- aeval (E (fix E)) V.

doif_t           : doif true E2 E3 V <- aeval E2 V.
doif_f           : doif false E2 E3 V <- aeval E3 V.

app_v_lam        : app_vals (lam E) V' V <- aeval (E V') V.
app_v_s          : app_vals s V (app s V).
app_v_zerop_t    : app_vals zerop z true.
app_v_zerop_f    : app_vals zerop (app s V) false.
app_v_pred_s     : app_vals pred (app s V) V.
```

F Equivalence of Natural and Algorithmic Semantics

```
na : neval E V -> aeval E V -> type.

na_t             : na neval_t aeval_t.
na_f             : na neval_f aeval_f.
na_if_t          : na (neval_if_t P2 P1) (aeval_if (doif_t Q2) Q1)
                       <- na P1 Q1
                       <- na P2 Q2.
na_if_f          : na (neval_if_f P2 P1) (aeval_if (doif_f Q2) Q1)
                       <- na P1 Q1
                       <- na P2 Q2.

na_z             : na neval_z aeval_z.
na_s             : na neval_s aeval_s.
na_pred          : na neval_pred aeval_pred.

na_zerop         : na neval_zerop aeval_zerop.

na_pair          : na (neval_pair P2 P1) (aeval_pair Q2 Q1)
                       <- na P1 Q1
                       <- na P2 Q2.
```

```
na_fst          : na (neval_fst P) (aeval_fst Q).
na_snd          : na (neval_snd P) (aeval_snd Q).

na_lam          : na neval_lam aeval_lam.

na_app_lam      : na (neval_app_lam P3 P2 P1)
                     (aeval_app (app_v_lam Q3) Q2 Q1)
                         <- na P1 Q1
                         <- na P2 Q2
                         <- na P3 Q3.
na_app_s        : na (neval_app_s P2 P1)
                     (aeval_app app_v_s Q2 Q1)
                         <- na P1 Q1
                         <- na P2 Q2.
na_app_zerop_t  : na (neval_app_zerop_t P2 P1)
                     (aeval_app app_v_zerop_t Q2 Q1)
                         <- na P1 Q1
                         <- na P2 Q2.
na_app_zerop_f  : na (neval_app_zerop_f P2 P1)
                     (aeval_app app_v_zerop_f Q2 Q1)
                         <- na P1 Q1
                         <- na P2 Q2.
na_app_pred_s   : na (neval_app_pred_s P2 P1)
                     (aeval_app app_v_pred_s Q2 Q1)
                         <- na P1 Q1
                         <- na P2 Q2.
na_let          : na (neval_let P2 P1)
                     (aeval_let Q2 Q1)
                         <- na P1 Q1
                         <- na P2 Q2.
na_letrec       : na (neval_letrec P2 P1)
                     (aeval_letrec Q2 Q1)
                         <- na P1 Q1
                         <- na P2 Q2.
na_fix          : na (neval_fix P) (aeval_fix Q)
                         <- na P Q.
```

G The Subject Reduction Property

G.1 Transformation of Evaluation Traces and Type Deductions

```
sr : neval E V -> of E A -> of V A -> type.

sbst : ({x:exp} ({A:tp} of E1 A -> of x A) -> of (E2 x) A2)
       -> neval E1 V1
       -> of (E2 V1) A2
       -> type.
```

```
sr_t          : sr (neval_t) (of_t) (of_t).
sr_f          : sr (neval_f) (of_f) (of_f).
sr_neval_if_t : sr (neval_if_t P2 P1) (of_if D3 D2 D1) C2
                   <- sr P2 D2 C2.
sr_neval_if_f : sr (neval_if_f P3 P1) (of_if D3 D2 D1) C3
                   <- sr P3 D3 C3.

sr_neval_z    : sr (neval_z) (of_z) (of_z).
sr_neval_s    : sr (neval_s) (of_s) (of_s).
sr_neval_pred : sr (neval_pred) (of_pred) (of_pred).

sr_neval_zerop   : sr (neval_zerop) (of_zerop) (of_zerop).

sr_neval_pair : sr (neval_pair P2 P1) (of_pair D2 D1) (of_pair C2 C1)
                   <- sr P1 D1 C1
                   <- sr P2 D2 C2.

sr_neval_fst  : sr (neval_fst P) (of_fst D) C1
                   <- sr P D (of_pair C1 C2).
sr_neval_snd  : sr (neval_snd P) (of_snd D) C2
                   <- sr P D (of_pair C1 C2).

sr_neval_lam  : sr (neval_lam) (of_lam D) (of_lam D).

sr_neval_app_lam : sr (neval_app_lam P3 P2 P1) (of_app D2 D1) C
                    <- sr P1 D1 (of_lam C1)
                    <- sr P2 D2 C2
                    <- sr P3 (C1 _ C2) C.

sr_neval_app_s   : sr (neval_app_s P2 P1)
                      (of_app D2 D1) (of_app C2 (of_s))
                    <- sr P1 D1 (of_s)
                    <- sr P2 D2 C2.

sr_neval_app_pred_s : sr (neval_app_pred_s P2 P1) (of_app D2 D1) C2
                       <- sr P1 D1 (of_pred)
                       <- sr P2 D2 (of_app C2 (of_s)).

sr_neval_app_zerop_t : sr (neval_app_zerop_t P2 P1)
                          (of_app D2 D1) (of_t)
                        <- sr P1 D1 (of_zerop)
                        <- sr P2 D2 (of_z).

sr_neval_app_zerop_f : sr (neval_app_zerop_f P2 P1)
                          (of_app D2 D1) (of_f)
                        <- sr P1 D1 (of_zerop)
```

```
                        <- sr P2 D2 (of_app C2 (of_s)).

sr_neval_let  : sr (neval_let P2 P1) (of_let D2 D1) C
                  <- sbst D2 P1 C2
                  <- sr P2 C2 C.

sr_neval_letrec  : sr (neval_letrec P2 P1) (of_letrec D2 D1) C
                     <- sbst D2 P1 C2
                     <- sr P2 C2 C.

sr_neval_fix     : sr (neval_fix P) (of_fix D) C
                     <- sr P (D (fix E) (of_fix D)) C.


sbst_var   : sbst ([x:exp] [d:{A:tp} of E A -> of x A] d A D) P C
               <- sr P D C.

sbst_t     : sbst ([x] [d] of_t) P of_t.
sbst_f     : sbst ([x] [d] of_f) P of_f.

sbst_if    : sbst ([x] [d] of_if (D3 x d) (D2 x d) (D1 x d)) P
                  (of_if C3 C2 C1)
               <- sbst D1 P C1
               <- sbst D2 P C2
               <- sbst D3 P C3.

sbst_z     : sbst ([x] [d] of_z) P of_z.
sbst_s     : sbst ([x] [d] of_s) P of_s.
sbst_pred  : sbst ([x] [d] of_pred) P of_pred.

sbst_zerop : sbst ([x] [d] of_zerop) P of_zerop.

sbst_pair  : sbst ([x] [d] of_pair (D2 x d) (D1 x d)) P
                  (of_pair C2 C1)
               <- sbst D1 P C1
               <- sbst D2 P C2.
sbst_fst   : sbst ([x] [d] of_fst (D x d)) P (of_fst C)
               <- sbst D P C.
sbst_snd   : sbst ([x] [d] of_snd (D x d)) P (of_snd C)
               <- sbst D P C.

sbst_lam   : sbst ([x] [d] of_lam (D x d)) P (of_lam C)
               <- {x':exp} {d':of x' A}
                     sbst ([x] [d] d') P d'
                     -> sbst ([x] [d] D x d x' d')
                             P (C x' d').
```

```
sbst_app    : sbst ([x][d] of_app (D2 x d) (D1 x d)) P (of_app C2 C1)
                <- sbst D1 P C1
                <- sbst D2 P C2.

sbst_let    : sbst ([x][d] of_let (D2 x d) (D1 x d)) P (of_let C2 C1)
                <- sbst D1 P C1
                <- {x':exp} {d':{B':tp} of E1 B' -> of x' B'}
                     ({B':tp} {D': of E1 B'} sbst ([x] [d] d' B' D') P
                                                  (d' B' D'))
                     -> sbst ([x][d] D2 x d x' d') P (C x' d').

sbst_letrec : sbst ([x][d] of_letrec (D2 x d) (D1 x d)) P
                   (of_letrec C2 C1)
                <- sbst D1 P C1
                <- {x':exp} {d':{B':tp} of (fix E1) B' -> of x' B'}
                      ({B':tp} {D': of (fix E1) B'}
                          sbst ([x] [d] d' B' D') P (d' B' D'))
                      -> sbst ([x][d] D2 x d x' d') P (C2 x' d').

sbst_fix    : sbst ([x] [d] of_fix (D x d)) P (of_fix C)
                <- {x':exp} {d':of x' A}
                      sbst ([x] [d] d') P d'
                      -> sbst ([x] [d] D x d x' d') P (C x' d').
```

G.2 Evaluation of Type Deductions

```
sr_eval     : of E A -> of V A -> type.

esubst      : ({x:exp} ({A:tp} of E1 A -> of x A) -> of (E2 x) A2)
              -> of (E2 V) A2
              -> type.

sr_eval_t          : sr_eval of_t of_t.
sr_eval_f          : sr_eval of_f of_f.
sr_eval_if_t       : sr_eval (of_if P3 P2 P1) Q2
                             <- sr_eval P1 of_t
                             <- sr_eval P2 Q2.
sr_eval_if_f       : sr_eval (of_if P3 P2 P1) Q3
                             <- sr_eval P1 of_f
                             <- sr_eval P3 Q3.

sr_eval_z          : sr_eval of_z of_z.
sr_eval_s          : sr_eval of_s of_s.
sr_eval_pred       : sr_eval of_pred of_pred.

sr_eval_zerop      : sr_eval of_zerop of_zerop.
```

```
sr_eval_pair       : sr_eval (of_pair P2 P1) (of_pair Q2 Q1)
                             <- sr_eval P1 Q1
                             <- sr_eval P2 Q2.
sr_eval_fst        : sr_eval (of_fst P) P1
                             <- sr_eval P (of_pair P2 P1).
sr_eval_snd        : sr_eval (of_snd P) P2
                             <- sr_eval P (of_pair P2 P1).

sr_eval_lam        : sr_eval (of_lam P) (of_lam P).

sr_eval_app_lam    : sr_eval (of_app P2 P1) P
                             <- sr_eval P1 (of_lam Q1)
                             <- sr_eval P2 Q2
                             <- sr_eval (Q1 _ Q2) P.
sr_eval_app_s      : sr_eval (of_app P2 P1) (of_app P of_s)
                             <- sr_eval P1 of_s
                             <- sr_eval P2 P.
sr_eval_app_pred_s : sr_eval (of_app P2 P1) Q3
                             <- sr_eval P1 of_pred
                             <- sr_eval P2 (of_app Q3 of_s).
sr_eval_app_zerop_t : sr_eval (of_app P2 P1) of_t
                             <- sr_eval P1 of_zerop
                             <- sr_eval P2 of_z.
sr_eval_app_zerop_f : sr_eval (of_app P2 P1) of_f
                             <- sr_eval P1 of_zerop
                             <- sr_eval P2 (of_app Q2 of_s).

sr_eval_let        : sr_eval (of_let P2 P1) Q
                             <- sr_eval P1 Q1
                             <- esubst P2 Q2
                             <- sr_eval Q2 Q.
sr_eval_letrec     : sr_eval (of_letrec P2 P1) Q
                             <- sr_eval P1 Q1
                             <- esubst P2 Q2
                             <- sr_eval Q2 Q.
sr_eval_fix        : sr_eval (of_fix P) Q
                             <- sr_eval (P _ (of_fix P)) Q.

esubst_var      : esubst ([x] [p] p A P1) Q1
                    <- sr_eval P1 Q1.

esubst_t        : esubst ([x] [p] of_t) of_t.
esubst_f        : esubst ([x] [p] of_f) of_f.
esubst_if       : esubst ([x] [p] of_if (P3 x p) (P2 x p) (P1 x p))
                         (of_if Q3 Q2 Q1)
                    <- esubst P1 Q1
                    <- esubst P2 Q2
```

```
                    <- esubst P3 Q3.

esubst_z        : esubst ([x] [p] of_z) of_z.
esubst_s        : esubst ([x] [p] of_s) of_s.
esubst_pred     : esubst ([x] [p] of_pred) of_pred.

esubst_zerop    : esubst ([x] [p] of_zerop) of_zerop.

esubst_pair     : esubst ([x] [p] of_pair (P2 x p) (P1 x p))
                         (of_pair Q2 Q1)
                    <- esubst P1 Q1
                    <- esubst P2 Q2.
esubst_fst      : esubst ([x] [p] of_fst (P x p)) (of_fst Q)
                    <- esubst P Q.
esubst_snd      : esubst ([x] [p] of_snd (P x p)) (of_snd Q)
                    <- esubst P Q.

esubst_lam      : esubst ([x] [p] of_lam (P x p)) (of_lam Q)
                    <- {x':exp} {p': of x' A'}
                           esubst ([x][p] p') p'
                           -> esubst ([x][p] P x p x' p') (Q x' p').

esubst_app      : esubst ([x] [p] of_app (P2 x p) (P1 x p))
                         (of_app Q2 Q1)
                    <- esubst P1 Q1
                    <- esubst P2 Q2.

esubst_let      :
   esubst ([x] [p] of_let (P2 x p) (P1 x p)) (of_let Q2 Q1)
      <- esubst P1 Q1
      <- {x':exp} {p': {A':tp} of E' A' -> of x' A'}
            ({A':tp} {P':of E' A'} esubst ([x] [p] p' A' P')
                                          (p' A' P'))
            -> esubst ([x] [p] P2 x p x' p') (Q2 x' p').

esubst_letrec   :
   esubst ([x] [p] of_letrec (P2 x p) (P1 x p)) (of_letrec Q2 Q1)
      <- esubst P1 Q1
      <- {x':exp} {p': {A':tp} of (fix E') A' -> of x' A'}
            ({A':tp} {P':of (fix E') A'} esubst ([x] [p] p' A' P')
                                                (p' A' P'))
            -> esubst ([x] [p] P2 x p x' p') (Q2 x' p').

esubst_fix      : esubst ([x] [p] of_fix (P x p)) (of_fix Q)
                    <- {x':exp} {p': of x' A'}
                           esubst ([x][p] p') p'
                           -> esubst ([x][p] P x p x' p') (Q x' p').
```

H Example Query and Answer Substitutions

```
 ?- sigma [D:of (app (app (letrec
                       ([add] (lam [x] (lam [y]
                                (if (app zerop x)
                                    y
                                    (app s (app (app add (app pred x)) y))))))
                       ([add] add))
                      (app s (app s z)))
                  (app s z)) A]
      sigma [Q:aeval _ V]
         sigma [NA:na P Q]
            sr P D C.

C <- of_app (of_app (of_app of_z of_s) of_s) of_s ,
P <-
neval_app_lam
   (neval_if_f
       (neval_app_s
           (neval_app_lam
               (neval_if_f
                   (neval_app_s
                       (neval_app_lam
                           (neval_if_t (neval_app_s neval_z neval_s)
                               (neval_app_zerop_t neval_z neval_zerop))
                           (neval_app_s neval_z neval_s)
                           (neval_app_lam neval_lam
                               (neval_app_pred_s (neval_app_s neval_z neval_s)
                                   neval_pred)
                               (neval_fix neval_lam)))
                       neval_s)
                   (neval_app_zerop_f (neval_app_s neval_z neval_s) neval_zerop))
               (neval_app_s neval_z neval_s)
               (neval_app_lam neval_lam
                   (neval_app_pred_s
                       (neval_app_s (neval_app_s neval_z neval_s) neval_s)
                       neval_pred)
                   (neval_fix neval_lam)))
           neval_s)
       (neval_app_zerop_f (neval_app_s (neval_app_s neval_z neval_s) neval_s)
           neval_zerop))
   (neval_app_s neval_z neval_s)
   (neval_app_lam neval_lam (neval_app_s (neval_app_s neval_z neval_s) neval_s)
       (neval_letrec neval_lam (neval_fix neval_lam))) ,
V <- app s (app s (app s z)) ,
A <- nat .
```

References

[1] Rod Burstall and Furio Honsell. A natural deduction treatment of operational semantics. In K. V. Nori and S. Kumar, editors, *8th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, December 1988.

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

[3] 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*. ACM Press, 1986.

[4] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM SIGPLAN/SIGACT, 1982.

[5] Scott Dietzen and Frank Pfenning. A declarative alternative to assert in logic programming. In Vijay Saraswat, editor, *International Logic Programming Symposium*. MIT Press, October 1991. To appear. Available as Ergo Report 90-094.

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

[7] John Hannan and Dale Miller. Enriching a meta-language with higher-order features. In John Lloyd, editor, *Proceedings of the Workshop on Meta-Programming in Logic Programming*, Bristol, England, June 1988. University of Bristol.

[8] 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.

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

[10] 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.

[11] Robert Harper and Frank Pfenning. Modularity in the LF logical framework. Talk given at the Second Workshop on Logical Frameworks, Edinburgh, Scotland, May 1991.

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

[13] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Proceedings of the Eigth International Conference on Logic Programming*. MIT Press, July 1991.

[14] Gopalan Nadathur and Dale 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.

[15] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, 1990.

[16] 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. Also available as Ergo Report 89–067, School of Computer Science, Carnegie Mellon University, Pittsburgh.

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

[18] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988. Available as Ergo Report 88–036, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[19] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.