Chapter 4

The Elf Programming Language

Elf, thou lovest best, I think,
The time to sit in a cave and drink.

— William Allingham
In Fairy Land [All75]

In Chapter 2 we have seen how deductive systems can be used systematically to specify aspects of the semantics of programming languages. In later chapters, we will see many more examples of this kind, including some examples from logic. In Chapter 3 we explored the logical framework LF as a formal meta-language for the representation of programming languages, their semantics, and their meta-theory. An important motivation behind the development of LF has been to provide a formal basis for the implementation of proof-checking and theorem proving tools, independently of any particular logic or deductive system. Note that search in the context of LF is the dual of type-checking: given a type A, find a closed object M of type A. If such an object M exists we refer to A as *inhabited*. Since types represent judgments and objects represent deductions, this is a natural formulation of the search for a deduction of a judgment via its representation in LF. Unlike type-checking, of course, the question whether a closed object of a given type exists is in general undecidable. The question of general search procedures for LF has been studied by Elliott, Pym and Wallen [Ell89, Ell90, Pym90, PW90, PW91], including the question of unification of LF objects modulo $\beta\eta$-conversion.

In the context of the study of programming languages, we encounter problems that are different from general proof search. For example, once a type system has been *specified* as a deductive system, how can we *implement* a type-checker or a type inference procedure for the language? Another natural question concerns the operational semantics: once specified as a deductive system, how can we take advantage of this specification to obtain an interpreter for the language? In both of these cases we are in a situation where algorithms are known and need to be implemented. The problem of proof search can also be phrased in these terms: given a logical system, implement algorithms for proof search that are appropriate to the system at hand.

Our approach to the implementation of algorithms is inspired by logic programming: specifications and programs are written in the same language. In traditional logic programming, the common basis for specifications and implementations has been the logic of Horn clauses; here, the common basis will be the logical framework LF. We would like to emphasize that specifications and programs are generally *not* the same: many specifications are not useful if interpreted as programs, and many programs would not normally be considered specifications. In the logic programming paradigm, execution is the search for a proof of some instance of a query. The operational semantics of the logic programming language specifies precisely how this search will be performed, given a list of Horn clauses which constitute the program. Thus, if one understands this operational reading of Horn clauses, one can induce the desired execution behavior by giving an appropriate presentation of a predicate. For example, the three Prolog programs

```
nat(zero).
nat(succ(X)) :- nat(X).
```

and

```
nat(succ(X)) :- nat(X).
nat(zero).
```

and

```
nat(X) :- nat(X).
nat(zero).
nat(succ(X)) :- nat(X).
```

specify exactly the same predicate `nat` which is true for all terms of the form `succ(...succ(zero)...)`, but only the first is useful as a program to enumerate the natural numbers, and only the first two are useful as programs to check if a given term represents a natural number.[1]

Exploration of the idea of program execution as proof search within a logical framework has led to the programming language Elf. We briefly sketch the operational intuition behind Elf in analogy to Prolog before going into greater detail

[1] In Prolog, `:-` is an implication written in reverse, and each rule is to be interpreted schematically, that is, stands for all its instances. For more on Prolog and traditional logic programming, see Sterling and Shapiro's introductory textbook [SS86] or some material in Chapter 8.

and examples in the remainder of this chapter. As explained above, proof search is modeled in LF by the search for a closed object of a given type. Thus a type plays the role of a query in Prolog and may contain free variables. A signature constitutes a program, and search is performed by trying each constant in a signature in turn in order to construct an object of the query type. First-order unification as it is familiar from Prolog is insufficient: instead we employ a constraint solver for typed equations over LF objects. Subgoals are solved in the order they are encountered, and search backtracks to the most recent choice point whenever a goal fails. This means that the search for a closed object is performed in a depth-first fashion as in Prolog.

Elf is a strongly typed language, since it is directly based on LF. The Elf interpreter must thus perform type reconstruction on programs and queries before executing them. Because of the complex type system of LF, this is a non-trivial task. In fact, it has been shown by Dowek [Dow93] that the general type inference problem for LF is undecidable, and thus not all types may be omitted from Elf programs. The algorithm for type reconstruction which is used in the implementation [Pfe91b, Pfe94] is based on the same constraint solving algorithm employed during execution.

4.1 Concrete Syntax

The concrete syntax of Elf is very simple, since we only have to model the relatively few constructs of LF. While LF is stratified into the level of kinds, families, and objects, the syntax is overloaded in that, for example, the symbol Π constructs dependent function types and dependent kinds. Similarly, juxtaposition is concrete syntax for instantiation of a type family and application of objects. We maintain this overloading in the concrete syntax for Elf and refer to expressions from any of the three levels collectively as *terms*. A signature is given as a sequence of *declarations*.[2]

[2] We describe here only the core language; the design and implementation of a module system for the structured presentation of signatures is currently in progress. The preliminary design is described in [HP99].

Terms	*term*	::=	| *id*	a or c or x
			| `{`*id*`:`*term*$_1$`}`*term*$_2$	$\Pi x{:}A_1.\ A_2$ or $\Pi x{:}A.\ K$
			| `[`*id*`:`*term*$_1$`]`*term*$_2$	$\lambda x{:}A.\ M$
			| *term*$_1$ *term*$_2$	$A\,M$ or $M_1\,M_2$
			| `type`	type
			| *term*$_1$ `->` *term*$_2$	$A_1 \to A_2$
			| *term*$_1$ `<-` *term*$_2$	$A_2 \to A_1$
			| `{`*id*`}`*term* | `[`*id*`]`*term* | `_`	omitted terms
			| *term*$_1$`:`*term*$_2$	cast
			| `(`*term*`)`	grouping
Declarations	*decl*	::=	*id* `:` *term*`.`	$a{:}K$ or $c{:}A$

The terminal *id* stands either for a bound variable, a free variable, or a constant at the level of families or objects. Bound variables and constants in Elf can be arbitrary identifiers, but free variables in a declaration or query must begin with an uppercase letter (a free, undeclared lowercase identifier is flagged as an undeclared constant). An uppercase identifier is one which begins with an underscore `_` or a letter in the range `A` through `Z`; all others are considered lowercase, including numerals. Identifiers may contain all characters except `(){}[]:.%` and whitespace. In particular, `A->B` would be a single identifier, while `A -> B` denotes a function type. The left-pointing arrow as in `B <- A` is a syntactic variant and parsed into the same representation as `A -> B`. It improves the readability of some Elf programs. Recall that `A -> B` is just an abbreviation for `{x:A} B` where `x` does not occur in `B`.

The right-pointing arrow `->` is right associative, while the left-pointing arrow `<-` is left associative. Juxtaposition binds tighter than the arrows and is left associative. The scope of quantifications `{`x : A`}` and abstractions `[`x : A`]` extends to the next closing parenthesis, bracket, brace or to the end of the term. Term reconstruction fills in the omitted types in quantifications `{`x`}` and abstractions `[`x`]` and omitted types or objects indicated by an underscore `_` (see Section 4.2). In case of essential ambiguity a warning or error message results.

Single-line comments begin with `%` and extend through the end of the line. A delimited comment begins with `%{` and ends with the matching `}%`, that is, delimited comments may be properly nested. The parser for Elf also supports infix, prefix, and postfix declarations similar to the ones available in Prolog.

4.2 Type and Term Reconstruction

A crucial element in a practical implementation of LF must be an algorithm for type reconstruction. We will illustrate type reconstruction with the Mini-ML examples from the previous chapter. First, the straightforward signature defining Mini-ML expressions which is summarized on page 46.

```
exp  : type.  %name exp E

z    : exp.
s    : exp -> exp.
case : exp -> exp -> (exp -> exp) -> exp.
pair : exp -> exp -> exp.
fst  : exp -> exp.
snd  : exp -> exp.
lam  : (exp -> exp) -> exp.
app  : exp -> exp -> exp.
letv  : exp -> (exp -> exp) -> exp.
letn  : exp -> (exp -> exp) -> exp.
fix  : (exp -> exp) -> exp.
```

The pragma `%name exp E` indicates to Elf that fresh variables of type `exp` which are created during type reconstruction or search should be named `E`, `E1`, `E2`, *etc.*

Next, we turn to the signature defining evaluations. Here are three declarations as they appear on page 56.

$$
\begin{array}{lll}
\mathrm{eval} & : & \mathrm{exp} \to \mathrm{exp} \to \mathrm{type} \\
\mathrm{ev\_z} & : & \mathrm{eval\ z\ z} \\
\mathrm{ev\_s} & : & \Pi E{:}\mathrm{exp}.\ \Pi V{:}\mathrm{exp}.\ \mathrm{eval}\ E\ V \to \mathrm{eval}\ (\mathrm{s}\ E)\ (\mathrm{s}\ V) \\
\mathrm{ev\_case\_z} & : & \Pi E_1{:}\mathrm{exp}.\ \Pi E_2{:}\mathrm{exp}.\ \Pi E_3{:}\mathrm{exp} \to \mathrm{exp}.\ \Pi V{:}\mathrm{exp}. \\
& & \quad \mathrm{eval}\ E_1\ \mathrm{z} \to \mathrm{eval}\ E_2\ V \to \mathrm{eval}\ (\mathrm{case}\ E_1\ E_2\ E_3)\ V
\end{array}
$$

In Elf's concrete syntax these would be written as

```
eval : exp -> exp -> type.
ev_z : eval z z.
ev_s : {E:exp} {V:exp} eval E V -> eval (s E) (s V).
ev_case_z :
   {E1:exp} {E2:exp} {E3:exp -> exp} {V:exp}
      eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V.
```

A simple deduction, such as

$$
\dfrac{\dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z} \qquad \dfrac{\dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z}}{\mathbf{s}\ \mathbf{z} \hookrightarrow \mathbf{s}\ \mathbf{z}}\,\mathsf{ev\_s}}{\mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{s}\ \mathbf{z} \mid \mathbf{s}\ x \Rightarrow \mathbf{z}}\,\mathsf{ev\_case\_z}
$$

is represented in Elf as[3]

```
ev_case_z z (s z) ([x:exp] z) (s z) (ev_z) (ev_s z z (ev_z)).
```

The top-level of Elf can perform type checking and reconstruction; later we will see how the user can also initiate search. In order to check the the object above represents a derivation of **case z of z** $\Rightarrow$ **s z** | **s** $x \Rightarrow$ **z**, we can pose the query

```
?- ev_case_z z (s z) ([x:exp] z) (s z) (ev_z) (ev_s z z (ev_z))
     : eval (case z (s z) ([x:exp] z)) (s z).
solved
```

The interpreter responds with the message `solved` which indicates that the given judgment holds, that is, the object to the left of the colon has type type to the right of the colon in the *current signature*. The current signature is embodied in the state of the top-level; please see the Elf User's Manual for details.[4]

We now reconsider the declaration of `ev_case_z`. The types of `E1`, `E2`, `E3`, and `V` are unambiguously determined by the kind of `eval` and the type of `case`. For example, `E1` must have type `exp`, since the first argument of `eval` must have type `exp`. This means, the declaration of `ev_case_z` could be replaced by

```
ev_case_z :
   {E1} {E2} {E3} {V}
      eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V.
```

It will frequently be the case that the types of the variables in a declaration can be determined from the context they appear in. To abbreviate declarations further we allow the omission of the explicit Π-quantifiers.[5] That is, the declaration above can be given even more succinctly as

```
ev_case_z : eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V.
```

This second step introduces a potential problem: the order of the quantifiers is not determined by the abbreviated declaration. Consequently, we do not know which argument to `ev_case_z` stands for `E1`, which for `E2`, *etc.* Fortunately, these arguments (which are objects) can be determined from the context in which `ev_case_z` occurs. Let `E1`, `E2`, `E3`, `V`, `E'` and `V'` stand for objects yet to be determined and consider the incomplete object

```
ev_case_z E1 E2 E3 V (ev_z) (ev_s E' V' (ev_z)).
```

The typing judgment

[3]The parentheses surrounding `ev_z` are unnecessary and inserted only for stylistic reasons.

[4][*just kidding*]

[5]Since the logical counterpart of the dependent function type constructor is the universal quantifier, we will often refer to Π as a quantifier.

```
ev_case_z E1 E2 E3 V
  : eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V
```

holds for all valid objects `E1`, `E2`, `E3`, and `V` of appropriate type. The next argument, `(ev_z)` has type `eval z z`. For the object to be well-typed we must thus have

```
eval E1 z = eval z z
```

where `=` represents definitional equality. Thus `E1 = z`. We can similarly determine that `E2 = s z`, `V = s z`, `E' = z`, and `V' = z`. However, `E3` is as yet undetermined. But if we also know the type of the whole object, namely

```
eval (case z (s z) ([x:exp] z)) (s z),
```

then `E3 = [x:exp] z` also follows. Since it will generally be possible to determine these arguments (up to conversion), we omit them in the input. We observe a strict correspondence between implicit quantifiers in a constant declaration and implicit arguments wherever the constant is used. This solves the problem that the order of implicit arguments is unspecified. With the abbreviated declarations

```
eval : exp -> exp -> type.
ev_z : eval z z.
ev_s : eval E V -> eval (s E) (s V).
ev_case_z :
      eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V.
```

the derivation above is concisely represented by

```
ev_case_z (ev_z) (ev_s (ev_z))
  : eval (case z (s z) ([x:exp] z)) (s z).
```

While arguments to a object of truly dependent function type ($\Pi x{:}A.\ B$ where x occurs free in B) are often redundant, there are examples where arguments cannot be reconstructed unambiguously. It is a matter of practical experience that the great majority of arguments to dependently typed functions do not need to be explicitly given, but can be reconstructed from context. The Elf type reconstruction algorithm will give a warning when an implicit quantifier in a constant declaration is likely to lead to essential ambiguity later.

For debugging purposes it is sometimes useful to know the values of reconstructed types and objects. The front-end of the Elf implementation can thus print the internal and fully explicit form of all the declarations if desired. One can also use the Elf interpreter to reconstruct types through the use of free variables in a query. For example

```
?- ev_case_z (ev_z) (ev_s (ev_z) : A) : B.

B = eval (case z (s z) E) (s z),
A = eval (s z) (s z).
```

After typing the first line, the interpreter responds with the second which contains the *answer substitution* for `A` and `B`. In this example, the substitution term for `B` itself contains a free variable `E`. This means that all instances of the given typing judgment are valid. This reflects that in our informal analysis `E3` (called `E` above) remained undetermined. In effect, the object `ev_case_z (ev_z) (ev_s (ev_z))` represents a whole class of derivations, namely all instances of

$$\dfrac{\dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\textsf{ev\_z} \qquad \dfrac{\dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\textsf{ev\_z}}{\mathbf{s}\ \mathbf{z} \hookrightarrow \mathbf{s}\ \mathbf{z}}\,\textsf{ev\_s}}{\mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{s}\ \mathbf{z} \mid \mathbf{s}\ x \Rightarrow e \hookrightarrow \mathbf{s}\ \mathbf{z}}\,\textsf{ev\_case\_z}$$

where e is a meta-variable that may contain free occurrences of x. We call derivations with free (meta-)variables *schematic derivations*. As we have seen, schematic derivations can be represented naturally in LF and Elf.

Type reconstruction is discussed in further detail in the documentation of the implementation. For the remainder of this chapter, the main feature to keep in mind is the duality between implicit quantifiers and implicit arguments.

4.3 A Mini-ML Interpreter in Elf

Let us recap the signature EV defining evaluation as developed so far in the previous section.

```
eval : exp -> exp -> type.
ev_z : eval z z.
ev_s : eval E V -> eval (s E) (s V).
ev_case_z :
      eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V.
```

One can now follow follow the path of Section 3.6 and translate the LF signature into Elf syntax. Our main concern in this section, however, will be to implement an executable interpreter for Mini-ML in Elf. In logic programming languages like Prolog, computation is search for a proof of a query according to a particular operational interpretation Horn clauses. In Elf, computation is search for a derivation of a judgment according to a particular operational interpretation of inference rules. In the terminology of the LF type theory, this translates to the search for an object of a given type over a particular signature.

To consider a concrete example, assume we are given a Mini-ML expression e. We would like to find an object `V` and a closed object `D` of type `eval` $\ulcorner e \urcorner$ `V`. Thus, we are looking simultaneously for a closed instance of a type, `eval` $\ulcorner e \urcorner$ `V`, and a closed object of this instance of the type. How would this search proceed? As an example, consider $e = \mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{s}\ \mathbf{z} \mid \mathbf{s}\ x \Rightarrow \mathbf{z}$. The query would have the form

```
?- D : eval (case z (s z) ([x:exp] z)) V.
```

where `V` is a free variable. Now the Elf interpreter will attempt to use each of the constants in the given signature in turn in order to construct a canonical object of this type. Neither `ev_z` nor `ev_s` are appropriate, since the types do not match. However, there is an instance of the last declaration

```
ev_case_z : eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V.
```

whose conclusion `eval (case E1 E2 E3) V` matches the current query by instantiating `E1 = z`, `E2 = (s z)`, `E3 = ([x:exp] z)`, and `V = V`. Thus, solutions to the *subgoals*

```
?- D2 : eval (s z) V.
?- D1 : eval z z.
```

would provide a solution `D = ev_case_z D1 D2` to the original query. At this point during the search, the incomplete derivation in mathematical notation would be

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ \mathbf{z} \hookrightarrow \mathbf{z}\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \mathbf{s}\ \mathbf{z} \hookrightarrow v\end{array}}{(\mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{s}\ \mathbf{z} \mid \mathbf{s}\ x \Rightarrow \mathbf{z}) \hookrightarrow v}\ \textsf{ev\_case\_z}$$

where $\mathcal{D}_1$, $\mathcal{D}_2$, and v are still to be filled in. Thus computation in Elf corresponds to bottom-up search for a derivation of a judgment. The subgoal `D2` can be matched against the type of `ev_s`, leading to the further subgoal

```
?- D3 : eval z V1.
```

while instantiating `V` to `s V1` for a new variable `V1` and `D2` to `ev_s D3`. In mathematical notation, the current state of search would be the partial derivation

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ \mathbf{z} \hookrightarrow \mathbf{z}\end{array} \qquad \dfrac{\begin{array}{c}\mathcal{D}_3\\ \mathbf{z} \hookrightarrow v_1\end{array}}{\mathbf{s}\ \mathbf{z} \hookrightarrow s\ v_1}\ \textsf{ev\_s}}{(\mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{s}\ \mathbf{z} \mid \mathbf{s}\ x \Rightarrow \mathbf{z}) \hookrightarrow \mathbf{s}\ v_1}\ \textsf{ev\_case\_z}.$$

The subgoals `D3` can be solved directly by `ev_z`, instantiating `V1` to `z`. The subgoal `D1` can also be solved directly and we obtain the instantiations

```
D = ev_case_z D1 D2
D2 = ev_s D3,
V = s V1,
D3 = ev_z,
V1 = z,
D1 = ev_z.
```

Substituting for some the intermediate variables we obtain the same answer that Elf would return.

```
?- D : eval (case z (s z) ([x:exp] z)) V.

V = s z,
D = ev_case_z ev_z (ev_s ev_z).
```

One can see that the matching process which is required for this search procedure must allow instantiation of the query as well as the declarations. The problem of finding a common instance of two terms is called *unification*. A unification algorithm for terms in first-order logic was first sketched by Herbrand [Her30]. The first full description of an efficient algorithm for unification was given by Robinson [Rob65], which has henceforth been a central building block for automated theorem proving procedures and logic programming languages. In Elf, Robinson's algorithm is not directly applicable because of the presence of types and λ-abstraction. Huet showed that unification in the simply-typed λ-calculus is undecidable [Hue73], a result later sharpened by Goldfarb [Gol81]. The main difficulty stems from the notion of definitional equality, which can be taken as β or $\beta\eta$-convertibility. Of course, the simply-typed λ-calculus is a subsystem of the LF type theory, and thus unifiability is undecidable for LF as well. A practical semi-decision procedure for unifiability in the simply-typed λ-calculus has been proposed by Huet [Hue75] and used in a number of implementations of theorem provers and logic programming languages [AINP88, EP89, PN90]. However, the procedure has the drawback that it may not only diverge but also branch, which is difficult to control in logic programming. Thus, in Elf, we have adopted the approach of constraint logic programming languages first proposed by Jaffar and Lassez [JL87], whereby difficult unification problems are postponed and carried along as constraints during execution. We will say more about the exact nature of the constraint solving algorithm employed in Elf in Section **??**. In this chapter, all unification problems encountered will be essentially first-order.

We have not payed close attention to the order of various operations during computation. In the first approximation, the operational semantics of Elf can be described as follows. Assume we are given a list of goals $A_1, \ldots, A_n$ with some free variables. Each type of an object-level constant c in a signature has the form $\Pi y_1{:}B_1 \ldots \Pi y_m{:}B_m.\ C$, where C is an atomic type. We call C the *target type* of c.

Also, in analogy to logic programming, we call c a *clause*, C the *head* of the clause c. Recall, that some of these quantifiers may remain implicit in Elf, and that $A \to B$ is only an abbreviation for $\Pi x{:}A.\ B$ where x does not occur in B. We instantiate $y_1, \ldots, y_m$ with fresh variables $Y_1, \ldots, Y_n$ and unify the resulting instance of C' with A_1, trying each constant in the signature in turn until unification succeeds. Unification may require further instantiation, leading to types $B_1', \ldots, B_n'$. We now set these up as subgoals, that is, we obtain the new list of goals $B_m', \ldots, B_1', A_2, \ldots, A_n$. The (closed) object we were looking for will be $c\ M_1 \ldots M_m$, where $M_1, \ldots, M_m$ are the objects of type $B_1', \ldots, B_m'$, respectively, yet to be determined. We say that the goal A_1 has been *resolved* with the clause c and refer to the process as *back-chaining*. Note that the subgoals will be solved "from the inside out," that is, B_m' is the first one to be considered. If unification should fail and no further constants are available in the signature, we *backtrack*, that is, we return to the most recent point where a goal unified with a clause head (*i.e.*, a target type of a constant declaration in a signature) and further choices were available. If there are no such choice points, the overall goal fails.

Logic programming tradition suggests writing the (atomic) target type C *first* in a declaration, since it makes is visually much easier to read a program. We follow the same convention here, although the reader should keep in mind that `A -> B` and `B <- A` are parsed to the same representation: the direction of the arrow has no semantic significance. The logical reading of `B <- A` is "`B` *if* `A`," although strictly speaking it should be "`B` *is derivable if* `A` *is derivable*." The left-pointing arrow is left associative so that `C <- B <- A`, `(C <- B) <- A`, `A -> (B -> C)`, and `A -> B -> C` are all syntactically different representations for the same type. Since we solve innermost subgoals first, the operational interpretation of the clause

```
ev_case_z :
    eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V.
```

would be: "*To solve a goal of the form* `eval (case E1 E2 E3) V`*, solve* `eval E2 V` *and, if successful, solve* `eval E1 z`." On the other hand, the clause

```
ev_case_z  : eval (case E1 E2 E3) V
             <- eval E1 z
             <- eval E2 V.
```

reads as: "*to solve a goal of the form* `eval (case E1 E2 E3) V`*, solve* `eval E1 z` *and, if successful,* `eval E2 V`." Clearly this latter interpretation is desirable from the operational point of view, even though the argument order to `ev_case_z` is reversed when compared to the LF encoding of the inference rules we have used so far. This serves to illustrate that a signature that is adequate as a specification of a deductive system is not necessarily adequate for search. We need to pay close attention to the order of the declarations in a signature (since they will be tried in succession) and the order of the subgoals (since they will be solved from the inside out).

We now complete the signature describing the interpreter for Mini-ML in Elf. It differs from the LF signature in Section 3.6 only in the order of the arguments to the constants. First the complete rules concerning natural numbers.

```
ev_z       : eval z z.
ev_s       : eval (s E) (s V)
              <- eval E V.
ev_case_z  : eval (case E1 E2 E3) V
              <- eval E1 z
              <- eval E2 V.
ev_case_s  : eval (case E1 E2 E3) V
              <- eval E1 (s V1')
              <- eval (E3 V1') V.
```

Recall that the application `(E3 V1')` was used to implement substitution in the object language. We discuss how this is handled operationally below when considering `ev_app`. Pairs are straightforward.

```
ev_pair : eval (pair E1 E2) (pair V1 V2)
           <- eval E1 V1
           <- eval E2 V2.
ev_fst  : eval (fst E) V1
           <- eval E (pair V1 V2).
ev_snd  : eval (snd E) V2
           <- eval E (pair V1 V2).
```

Abstraction and function application employ the notion of substitution. Recall the inference rule and its representation in LF:

$$\frac{e_1 \hookrightarrow \mathbf{lam}\, x.\, e_1' \qquad e_2 \hookrightarrow v_2 \qquad [v_2/x]e_1' \hookrightarrow v}{e_1\; e_2 \hookrightarrow v}\,\mathsf{ev\_app}$$

$$\begin{array}{lll} \text{ev\_app} & : & \Pi E_1{:}\text{exp}.\ \Pi E_2{:}\text{exp}.\ \Pi E_1'{:}\text{exp} \to \text{exp}.\ \Pi V_2{:}\text{exp}.\ \Pi V{:}\text{exp}. \\ & & \quad \text{eval}\ E_1\ (\text{lam}\ E_1') \\ & & \quad \to \text{eval}\ E_2\ V_2 \\ & & \quad \to \text{eval}\ (E_1'\ V_2)\ V \\ & & \quad \to \text{eval}\ (\text{app}\ E_1\ E_2)\ V. \end{array}$$

As before, we transcribe this (and the trivial rule for evaluating λ-expressions) into Elf.

```
ev_lam  : eval (lam E) (lam E).

ev_app  : eval (app E1 E2) V
             <- eval E1 (lam E1')
             <- eval E2 V2
             <- eval (E1' V2) V.
```

The operational reading of the `ev_app` rule is as follows. In order to evaluate an application $e_1\ e_2$ we evaluate e_1 and match the result against $\mathbf{lam}\ x.\ e_1'$. If this succeeds we evaluate e_2 to the value v_2. Then we evaluate the result of substituting v_2 for x in e_1'. The Mini-ML expression $\mathbf{lam}\ x.\ e_1'$ is represented as in LF as lam $(\lambda x{:}\mathrm{exp}.\ \ulcorner e_1' \urcorner)$, and the variable `E1' : exp -> exp` will be instantiated to `([x:exp]` $\ulcorner e_1' \urcorner$`)`.[6] In the operational semantics of Elf, an application which is not in canonical form (such as `(E1' V2)` after instantiation of `E1'` and `V2`) will be reduced until it is in head-normal form (see Section 8.2)—in this case this means performing the substitution of `V2` for the top-level bound variable in `E1'`. As an example, consider the evaluation of $(\mathbf{lam}\ x.\ x)\ \mathbf{z}$ which is given by the deduction

$$\dfrac{\dfrac{}{\mathbf{lam}\ x.\ x \hookrightarrow \mathbf{lam}\ x.\ x}\,\textsf{ev\_lam} \qquad \dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\textsf{ev\_z} \qquad \dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\textsf{ev\_z}}{(\mathbf{lam}\ x.\ x)\ \mathbf{z} \hookrightarrow \mathbf{z}}\,\textsf{ev\_app}.$$

The first goal is

```
?- D : eval (app (lam [x:exp] x) z) V.
```

This is resolved with the clause `ev_app`, yielding the subgoals

```
?- D1 : eval (lam [x:exp] x) (lam E1').
?- D2 : eval z V2.
?- D3 : eval (E1' V2) V.
```

The first subgoal will be resolved with the clause `ev_lam`, instantiating `E1'` to `([x:exp] x)`. The second subgoal will be resolved with the clause `ev_z`, instantiating `V2` to `z`. Thus, by the time the third subgoal is considered, it has been instantiated to

```
?- D3 : eval (([x:exp] x) z) V.
```

When this goal is unified with the clauses in the signature, `(([x:exp] x) z)` is reduced to `z`. It thus unifies with the head of the clause `ev_z`, and `V` is instantiated to `z` to yield the answer

```
V = z.
D = ev_app ev_z ev_z ev_lam.
```

[6] Recall that `[`$x{:}A$`]`B is Elf's concrete syntax for $\lambda x{:}A.\ B$.

Note that because of the subgoal ordering, ev_lam is the last argument to ev_app.

Evaluation of **let**-expressions follows the same schema as function application, and we again take advantage of meta-level β-reduction in order to model object-level substitution.

```
ev_letv : eval (letv E1 E2) V
     <- eval E1 V1
     <- eval (E2 V1) V.

ev_letn : eval (letn E1 E2) V
     <- eval (E2 E1) V.
```

The Elf declaration for evaluating a fixpoint construct is again a direct transcription of the corresponding LF declaration. Recall the rule

$$\frac{[\mathbf{fix}\ x.\ e/x]e \hookrightarrow v}{\mathbf{fix}\ x.\ e \hookrightarrow v}\,\textsf{ev\_fix}$$

```
ev_fix  : eval (fix E) V
            <- eval (E (fix E)) V.
```

This declaration introduces non-terminating computations into the interpreter. Reconsider the example from page 16, $\mathbf{fix}\ x.\ x$. Its representation in Elf is given by `fix ([x:exp] x)`. Attempting to evaluate this expression leads to the following sequence of goals.

```
?- D : eval (fix ([x:exp] x)) V.
?- D1 : eval (([x:exp] x) (fix ([x:exp] x))) V.
?- D1 : eval (fix ([x:exp] x)) V.
```

The step from the original goal to the first subgoal is simply the back-chaining step, instantiating `E` to `[x:exp] x`. The second is a β-reduction required to transform the goal into canonical form, relying on the rule of type conversion. The third goal is then a renaming of the first one, and computation will diverge. This corresponds to the earlier observation (see page 16) that there is no v such that the judgment $\mathbf{fix}\ x.\ x \hookrightarrow v$ is derivable.

It is also possible that evaluation fails finitely, although in our formulation of the language only for Mini-ML expressions that are not well-typed (in Mini-ML, not LF). For example,

```
?- D : eval (fst z) V.
no
```

The only subgoal considered is `D' : eval z (pair V V2)` after resolution with the clause `ev_fst`. This subgoal fails, since there is no rule that would permit a conclusion of this form, that is, no clause head unifies with `eval z (pair V V2)`.

As a somewhat larger example, we reconsider the evaluation which doubles the natural number 1, as given on page 16. Reading the justifications of the lines 1–17 from the bottom-up yields the same sequence of inference rules as reading the object `D` below from left to right.

```
?- D : eval (app
               (fix [f:exp] lam [x:exp]
                  (case x z ([x':exp] s (s (app f x')))))
               (s z))
        V.

V = s (s z),
D =
   ev_app
      (ev_case_s
          (ev_s (ev_s (ev_app (ev_case_z ev_z ev_z)
                          ev_z (ev_fix ev_lam))))
          (ev_s ev_z))
      (ev_s ev_z) (ev_fix ev_lam).
;
no more solutions
```

The example above exhibits another feature of the interactive top-level of Elf. After displaying the first solution fir `V` and `D` the Elf interpreter pauses. If one simply inputs a newline then Elf prompts again with `?-` , waiting for another query. If the user types a semi-colon, then the interpreter backtracks as if the most recent subgoal had failed, and tries to find another solution. This can be a useful debugging device. We know that evaluation of Mini-ML expressions should be deterministic in two ways: there should be only one value (see Theorem 2.6) and there should also be at most one deduction of every evaluation judgment. Thus backtracking should never result in another value or another deduction of the same value. Fortunately, the interpreter confirms this property in this particular example.

As a second example for an Elf program, we repeat the definition of value

$$\text{Values} \quad v \quad ::= \quad \mathbf{z} \mid \mathbf{s}\, v \mid \langle v_1, v_2 \rangle \mid \mathbf{lam}\, x.\, e$$

which was presented as a judgment on page 18 and as an LF signature on page 62.

```
value : exp -> type.  %name value P

val_z     : value z.
val_s     : value (s E) <- value E.
val_pair  : value (pair E1 E2) <- value E1 <- value E2.
val_lam   : value (lam E).
```

This signature can be used as a program to decide if a given expression is a value. For example,

```
?- value (pair z (s z)).
solved

yes
?- value (fst (pair z (s z))).
no
?- value (lam [x] (fst x)).
solved
```

Here we use a special query form that consists only of a type A, rather than a typing judgment $M : A$. Such a query is interpreted as $X : A$ for a new free variable X whose instantiation will not be shown with in the answer substitution. In many cases this query form is substantially more efficient than the form $M : A$, since the interpreter optimizes such queries and does not construct the potentially large object M.

4.4 An Implementation of Value Soundness

We now return to the proof of value soundness which was first given in Section 2.4 and formalized in Section 3.7. The theorem states that evaluation always returns a value. The proof of the theorem proceeds by induction over the structure of the derivation $\mathcal{D}$ of the judgment $e \hookrightarrow v$, that is, the evaluation of e. The first step in the formalization of this proof is to formulate a judgment between deductions,

$$\begin{array}{c}\mathcal{D}\\ e \hookrightarrow v\end{array} \Longrightarrow \begin{array}{c}\mathcal{P}\\ v\ \mathit{Value}\end{array}$$

which relates every $\mathcal{D}$ to some $\mathcal{P}$ and whose definition is based on the structure of $\mathcal{D}$. This judgment is then represented in LF as a type family vs, following the judgments-as-types principle.

$$\text{vs} \quad : \quad \Pi E{:}\text{exp}.\ \Pi V{:}\text{exp}.\ \text{eval}\ E\ V \to \text{value}\ V \to \text{type}.$$

Each of the various cases in the induction proof gives rise to one inference rule for the $\Longrightarrow$ judgment, and each such inference rule is represented by a constant declaration in LF. We illustrate the Elf implementation with the case where $\mathcal{D}$ ends in the rule ev_fst and then present the remainder of the signature in Elf more tersely.

Case:

$$\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}' \\ e \hookrightarrow \langle v_1, v_2 \rangle\end{matrix}}{\mathbf{fst}\ e \hookrightarrow v_1}\ \mathsf{ev\_fst}.$$

Then the induction hypothesis applied to $\mathcal{D}'$ yields a deduction $\mathcal{P}'$ of the judgment $\langle v_1, v_2 \rangle$ *Value*. By examining the inference rules we can see that $\mathcal{P}'$ must end in an application of the val_pair rule, that is,

$$\mathcal{P}' = \dfrac{\begin{matrix}\mathcal{P}_1 \\ v_1\ \mathit{Value}\end{matrix} \qquad \begin{matrix}\mathcal{P}_2 \\ v_2\ \mathit{Value}\end{matrix}}{\langle v_1, v_2 \rangle\ \mathit{Value}}\ \mathsf{val\_pair}$$

for some $\mathcal{P}_1$ and $\mathcal{P}_2$. Hence v_1 *Value* must be derivable, which is what we needed to show.

This is represented by the following inference rule for the $\Longrightarrow$ judgment.

$$\dfrac{\begin{matrix}\mathcal{D}' \\ e \hookrightarrow \langle v_1, v_2 \rangle\end{matrix} \Longrightarrow \dfrac{\begin{matrix}\mathcal{P}_1 \\ v_1\ \mathit{Value}\end{matrix} \qquad \begin{matrix}\mathcal{P}_2 \\ v_2\ \mathit{Value}\end{matrix}}{\langle v_1, v_2 \rangle\ \mathit{Value}}\ \mathsf{val\_pair}}{\dfrac{\begin{matrix}\mathcal{D}' \\ e \hookrightarrow \langle v_1, v_2 \rangle\end{matrix}}{\mathbf{fst}\ e \hookrightarrow v_1}\ \mathsf{ev\_fst} \Longrightarrow \begin{matrix}\mathcal{P}_1 \\ v_1\ \mathit{Value}\end{matrix}}\ \mathsf{vs\_fst}$$

It representation in LF is given by

$$\begin{array}{lll} \text{vs\_fst} & : & \Pi E_1\text{:exp. } \Pi V_1\text{:exp. } \Pi V_2\text{:exp.} \\ & & \Pi D'\text{:eval } E \text{ (pair } V_1\ V_2\text{). } \Pi P_1\text{:value } V_1.\ \Pi P_2\text{:value } V_2. \\ & & \quad \text{vs } E \text{ (pair } V_1\ V_2\text{) } D' \text{ (val\_pair } V_1\ V_2\ P_1\ P_2\text{)} \\ & & \quad \to \text{vs (fst } E\text{) } V_1 \text{ (ev\_fst } E\ V_1\ V_2\ D'\text{) } P_1 \end{array}$$

This may seem unwieldy, but Elf's type reconstruction comes to our aid. In the declaration of vs, the quantifiers on E and V can remain implicit:

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

The correspoinding arguments to `vs` now also remain implicit. We also repeat the declarations for the inference rules involved in the deduction above.

```
ev_fst  : eval (fst E) V1 <- eval E (pair V1 V2).
val_pair  : value (pair E1 E2) <- value E1 <- value E2.
```

Here is the declaration of the vs_fst constant:

```
vs_fst  : vs (ev_fst D') P1 <- vs D' (val_pair P2 P1).
```

Note that this declaration only has to deal with deductions, not with expressions. Term reconstruction expands this into

```
vs_fst :
   {E:exp} {E1:exp} {E2:exp} {D':eval E (pair E1 E2)}
      {P2:value E2} {P1:value E1}
      vs E (pair E1 E2) D' (val_pair E2 E1 P2 P1)
         -> vs (fst E) E1 (ev_fst E E1 E2 D') P1.
```

Disregarding the order of quantifiers and the choice of names, this is the LF declaration given above. We show the complete signature which implements the proof of value soundness without further comment. The declarations can be derived from the material and the examples in Sections 2.4 and 3.7.

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

% Natural Numbers
vs_z      : vs (ev_z) (val_z).
vs_s      : vs (ev_s D1) (val_s P1)
             <- vs D1 P1.
vs_case_z : vs (ev_case_z D2 D1) P2
             <- vs D2 P2.
vs_case_s : vs (ev_case_s D3 D1) P3
             <- vs D3 P3.

% Pairs
vs_pair : vs (ev_pair D2 D1) (val_pair P2 P1)
            <- vs D1 P1
            <- vs D2 P2.
vs_fst  : vs (ev_fst D') P1
            <- vs D' (val_pair P2 P1).
vs_snd  : vs (ev_snd D') P2
            <- vs D' (val_pair P2 P1).

% Functions
vs_lam  : vs (ev_lam) (val_lam).
vs_app  : vs (ev_app D3 D2 D1) P3
```

```
            <- vs D3 P3.

% Definitions
vs_letv : vs (ev_letv D2 D1) P2
   <- vs D2 P2.
vs_letn : vs (ev_letn D2) P2
   <- vs D2 P2.

% Recursion
vs_fix : vs (ev_fix D1) P1
          <- vs D1 P1.
```

This signature can be used to transform evaluations into value deductions. For example, the evaluation of **case z of z** $\Rightarrow$ **s z** | **s** $x \Rightarrow$ **z** considered above is given by the Elf object

```
ev_case_z (ev_s ev_z) ev_z
```

of type

```
eval (case z (s z) ([x:exp] z)) (s z).
```

We can transform this evaluation into a derivation which shows that `s z` is a value:

```
?- vs (ev_case_z (ev_s ev_z) ev_z) P.

P = val_s val_z.
```

The sequence of subgoals considered is

```
?- vs (ev_case_z (ev_s ev_z) ev_z) P.
% Resolved with clause vs_case_z
?- vs (ev_s ev_z) P.
% Resolved with clause vs_s [with P = val_s P1]
?- vs ev_z P1.
% Resolved with clause vs_z [with P1 = val_z]
```

This approach to testing the meta-theory is feasible for this simple example. As evaluations become more complicated, however, we would like to use the program for evaluation to generate a appropriate derivations and then transform them. This form of sequencing of computation can be achieved in Elf, but only in a somewhat awkward fashion because of the order in which subgoals are solved. In order to properly stage the queries, we need an auxiliary family `find` whose only purpose is to explicitly construct an evaluation. Note that `find` is a level 2 type family.

```
find : {E:exp} eval E V -> type.
find_all : {E:exp} {D:eval E V} find E D.
```

The family tfm below now first constructs an evaluation D and then transforms it.

```
tfm : {E:exp} eval E V -> value V -> type.
tfm_all : tfm E D P <- find E D <- vs D P.
```

For example,

```
?- tfm (case z (s z) ([x:exp] z)) D P.

P = val_s val_z,
D = ev_case_z (ev_s ev_z) ev_z.
```

Since there is often a need for staging queries, the Elf implementation provides a special kind of query sigma $[x : A]$ B.[7] A query of this form first solves A and instantiates x to an object of the appropriate type. It then solves B under this instantiation. The sigma construct can be nested, but it is a special top-level form and cannot be embedded within signatures. Using this special construct, we can reformulate the query without special auxiliary families find and tfm.

```
?- sigma [D: eval (case z (s z) ([x:exp] z)) V] vs D P.

P = val_s val_z,
V = s z.
```

This is reminiscent of the use of Σ-types (or *strong sums*) in some type theories—thus the name sigma.

4.5 Dynamic and Static Constants

Via the judgments-as-types and deductions-as-object principles of representation, Elf unifies concepts which are ordinarily distinct in logic programming languages. For example, a goal is represented as a type in Elf. If we look a little deeper, Elf associates a variable M with each goal type A such that solving the goal requires finding a closed object M of some closed instance of A. That is, Elf unifies the concepts of goal and logic variable. This identification of concepts has not presented any problems so far, but in some circumstances it can lead to undesired operational behavior of Elf programs. One example of this problem is in the program for Mini-ML type inference in Section 5.5. Here we will use simple list manipulation programs as examples to illustrate the difficulties and the solution adopted in Elf.

First, the declaration of natural numbers. We declare s, the successor function, as a prefix operator so we can write 2, for example, as s s 0 without additional parentheses. Note that without the prefix declaration this term would be associated to the left and parsed incorrectly as ((s s) 0).

[7]The exact form of the support for staging may differ in future versions of Elf.

```
nat  : type.        %name nat N
0    : nat.
s    : nat -> nat.  %prefix 20 s
```

The prefix declaration has the general form `%prefix` *prec* $id_1 \ldots id_n$ and gives the constants $id_1, \ldots, id_n$ precedence *prec*. The second declaration introduces lists of natural numbers. We declare "`;`" as a right-associative infix constructor for lists.

```
list : type.        %name list L K M
nil  : list.
;    : nat -> list -> list.  %infix right 10 ;
```

For example, `(0 ; s 0 ; s s 0 ; nil)` denotes the list of the first three natural numbers; `(0 ; ((s 0) ; ((s (s 0)) ; nil)))` is its fully parenthesized version, and `(; 0 (; (s 0) (; (s (s 0)) nil)))` is the prefix form which would have to be used if no infix declarations had been supplied.

The definition of the `append` program is straightforward. It is implemented as a type family indexed by three lists, where the third list must be the result of appending the first two. This can easily be written as a judgment (see Exercise 4.6).

```
append  : list -> list -> list -> type.

ap_nil  : append nil K K.
ap_cons : append (X ; L) K (X ; M) <- append L K M.
```

This program exhibits the expected behavior when given ground lists as the first two arguments. It can also be used to split a list when the third argument is given the first two are variables. For example,

```
?- append (0 ; s 0 ; nil) (s s 0 ; nil) M.

M = 0 ; s 0 ; s s 0 ; nil.
yes
?- append L K (0 ; s 0 ; s s 0 ; nil).

K = 0 ; s 0 ; s s 0 ; nil,
L = nil.
;
K = s 0 ; s s 0 ; nil,
L = 0 ; nil.
;
K = s s 0 ; nil,
L = 0 ; s 0 ; nil.
;
```

```
K = nil,
L = 0 ; s 0 ; s s 0 ; nil.
;
no more solutions
```

The behavior of this program on a query such as `?- append nil K M.` may be a surprise at first. Recall that our first approximation to the full operational semantics calls for *closed* answers. Omitting deductions, the first four solutions are given below.

```
?- append nil K M.

M = nil,
K = nil.
;
M = 0 ; nil,
K = 0 ; nil.
;
M = s 0 ; nil,
K = s 0 ; nil.
;
M = s s 0 ; nil,
K = s s 0 ; nil.
```

The analogy to Prolog suggests that the goal `append nil K M` should instead be resolved with the clause `ap_nil`, instantiating `K` and `M` to some common variable. For example,

```
?- append nil K M.

M = L,
K = L.
;
no more solutions
```

This solution subsumes all the others given above, that is, any other solution will be an instance of this one. But it violates the principle that search proceeds by finding a closed object of a closed instance of the given type, since the answer contains the free variable `L`. In many circumstance like the one above it is desirable to permit free variables of certain types in answer substitutions. Thus we need a way to instruct the Elf interpreter not to search for closed objects of types `list` or `nat`. We label such families *static*, while families such as `append` are labelled *dynamic*. The Elf top-level provides ways to load signatures such that its declarations will be considered either static or dynamic.[8] In the example above, the signatures defining natural numbers and lists would normally be loaded as static, while the signature defining `append` would be loaded as dynamic. When all signatures of this example are loaded dynamically, no free variables are tolerated in the answer, and the Elf interpreter will enumerate lists of natural numbers as shown above.

Declaring too many signatures to be static can lead to unexpected behavior. For example, if all signatures above are loaded statically, then there will never be any search! Each query will immediately "succeed," and the object of the given query type remains a free variable. For example,

```
?- Q : append (0 ; nil) K M.

M = M,
K = K,
Q = Q.
```

The variable `Q` variable remains uninstantiated; we have "reduced" the problem of finding an object of type `append nil (0 ; nil) nil` to itself. In the usual situation, that is, when `append` is dynamic and all other types are static, we would obtain, for example,

```
?- Q : append (0 ; nil) K M.

M = 0 ; K1,
K = K1,
Q = ap_cons ap_nil.
```

The significance of static type families goes beyond the final answer to a query. In Mini-ML type inference, for example, it is crucial that Mini-ML types are static in order to avoid excessive backtracking and ensure termination (see Exercise 5.7).

Besides families, objects may also be either static or dynamic. An object which is static will never be used during search, and thus solutions may be overlooked. On the other hand, objects whose type would lead to undesirable search behavior can be excluded. In Elf programming practice it is usual to divide the signatures into the static and the dynamic part, situated in separate files. Each file contains declarations of families and objects.

The distinction between dynamic and static constants affects only search and not the underlying LF type theory or the process of type reconstruction. It reintroduces some aspects of the distinction between goals and logic variables that is present in Prolog. A formula in Prolog corresponds to a dynamic type in Elf, a predicate to a dynamic type family. A logic variable in Prolog is now a variable of static type, that is, a variable which is instantiated only by unification, but not by search.

[8]A more principled solution has been adopted in the module calculus for Elf [HP99].

4.6 Exercises

Exercise 4.1 Show the sequence of subgoals generated by the query which attempts to evaluate the Mini-ML expression $(\mathbf{lam}\ x.\ x\ x)\ (\mathbf{lam}\ x.\ x\ x)$. Also show that this expression is not well-typed in Mini-ML, although its representation is of course well-typed in LF.

Exercise 4.2 The Elf interpreter for Mini-ML contains some obvious redundancies. For example, while constructing an evaluation of $\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3$, the expression e_1 will be evaluated twice if its value is not zero. Write a program for evaluation of Mini-ML expressions in Elf that avoids this redundant computation and prove that the new interpreter and the natural semantics given here are equivalent. Implement this proof as a higher-level judgment relating derivations.

Exercise 4.3 Implement the optimized version of evaluation from Exercise 2.11 in which values that are substituted for variables during evaluation are not evaluated again. Based on the modified interpreter, implement *bounded evaluation* $e \overset{n}{\hookrightarrow} v$ with the intended meaning that e evaluates to v in at most n steps (for a natural number n). You may make the simplifying but unrealistic assumption that every inference rule represents one step in the evaluation.

Exercise 4.4 Write Elf programs to implement quicksort and insertion sort for lists of natural numbers, including all necessary auxiliary judgments.

Exercise 4.5 Write declarations to represent natural numbers in binary notation.

1. Implement a translation between binary and unary representations in Elf.
2. Formulate an appropriate representation theorem and prove it.
3. Implement the proof of the representation theorem in Elf.

Exercise 4.6 Give the definition of the judgment *append* as a deductive system. *append* $l_1\ l_2\ l_3$ should be derivable whenever l_3 is the result of appending l_1 and l_2.