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 [Ell89, Ell90] and Pym and Wallen [PW90, Pym90, PW91, Pym92], 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 derivation 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 inference rules that constitute the program. Thus, if one understands this operational reading of inference rules, the programmer can obtain the desired execution behavior by defining judgments appropriately. We explain this in more detail in Section **??** and investigate it more formally in Chapter **??**.

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 [Pfe91a, Pfe94] is based on the same constraint solving algorithm employed during execution.

The current implementation of Elf is within the Twelf system [PS99]. The reader should consult an up-to-date version of the User's Guide for further information regarding the language, its implementation, and its use. Sources, binaries for various architectures, examples, and other materials are available from the Twelf home page [Twe98].

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 levels 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*. We describe here only the core language which corresponds very closely to LF. The main addition is a form of declaration $id\ \texttt{:}\ term_1\ \texttt{=}\ term_2$ that introduces an abbreviation *id* for $term_2$.

Terms	*term*	::=	id	a or c or x
		|	$\texttt{\{}id\texttt{:}term_1\texttt{\}}term_2$	$\Pi x{:}A_1.\ A_2$ or $\Pi x{:}A.\ K$
		|	$\texttt{[}id\texttt{:}term_1\texttt{]}\,term_2$	$\lambda x{:}A.\ M$
		|	$term_1\ term_2$	$A\ M$ or $M_1\ M_2$
		|	`type`	`type`
		|	$term_1$ `->` $term_2$	$A_1 \to A_2$
		|	$term_1$ `<-` $term_2$	$A_2 \to A_1$
		|	$\texttt{\{}id\texttt{\}}term \mid \texttt{[}id\texttt{]}\,term \mid \texttt{\_}$	omitted terms
		|	$term_1\texttt{:}term_2$	type ascription
		|	$\texttt{(}term\texttt{)}$	grouping
Declarations	*decl*	::=	$id\ \texttt{:}\ term\texttt{.}$	$a{:}K$ or $c{:}A$
		|	$id\ \texttt{:}\ term_1\ \texttt{=}\ term_2\texttt{.}$	$c{:}A = M$

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 $\texttt{\{}x\ \texttt{:}\ A\texttt{\}}$ and abstractions $\texttt{[}x\ \texttt{:}\ A\texttt{]}$ extends to the next closing parenthesis, bracket, brace or to the end of the term. Term reconstruction fills in the omitted types in quantifications $\texttt{\{}x\texttt{\}}$ and abstractions $\texttt{[}x\texttt{]}$ 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.

4.2 Type and Term Reconstruction

A crucial element in a practical implementation of LF is 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 x.

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 declaration `%name exp E x.` 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.

eval	:	$\mathsf{exp} \to \mathsf{exp} \to \mathsf{type}$
ev_z	:	$\mathsf{eval}\ \mathsf{z}\ \mathsf{z}$
ev_s	:	$\Pi E{:}\mathsf{exp}.\ \Pi V{:}\mathsf{exp}.\ \mathsf{eval}\ E\ V \to \mathsf{eval}\ (\mathsf{s}\ E)\ (\mathsf{s}\ V)$
ev_case_z	:	$\Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}.\ \Pi E_3{:}\mathsf{exp} \to \mathsf{exp}.\ \Pi V{:}\mathsf{exp}.$ $\mathsf{eval}\ E_1\ \mathsf{z} \to \mathsf{eval}\ E_2\ V \to \mathsf{eval}\ (\mathsf{case}\ E_1\ E_2\ E_3)\ V$

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

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

The Elf implementation performs type checking and reconstruction; later we will see how the user can also initiate search. In order to check that the object above represents a derivation of **case z of z** $\Rightarrow$ **s z** | **s** $x \Rightarrow$ **z**, we construct an *anonymous definition*

```
_ = 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).
```

The interpreter re-prints the declaration, 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 Twelf system and comprises all loaded files. Please see the Twelf User's Guide for details.

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. Consequently, 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. Therefore, 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

```
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 an 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. 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 in general computation is search for a derivation. 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 $e = \textbf{case z of z} \Rightarrow \textbf{s z} \mid \textbf{s}\ x \Rightarrow \textbf{z}$. 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

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

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. We solve the currently unsolved subgoals going through the partial deduction in a depth-first, left-to-right manner. So the next step would be to solve

```
?- D1 : eval z z.
```

We see that only one inference rule can apply, namely `ev_z`, instantiating `D1` to `ev_z`. Now 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

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

The subgoals `D3` can be solved directly by `ev_z`, instantiating `V1` to `z`. We obtain the following cumulative substitution:

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

Eliminating 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, Pfe91a, Pau94]. 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 $\Pi y_1{:}B_1 \ldots \Pi y_m{:}B_m.\ C_1 \to \cdots \to C_k \to C$, 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. We instantiate $y_1, \ldots, y_m$ with fresh variables $Y_1, \ldots, Y_m$ and unify the resulting instance of C' with A_1, trying each constant in the signature in turn until unification succeeds. Unification may instantiate $C_1, \ldots, C_k$ to $C'_1, \ldots, C'_k$. We now set these up as subgoals, that is, we obtain the new list of goals $C'_k, \ldots, C'_1, A_2, \ldots, A_n$. The object we were looking for will be $c\, Y_1 \ldots Y_n\ M_1 \ldots M_k$, where $M_1, \ldots, M_k$ are the objects of type $C'_1, \ldots, C'_k$, 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, C'_k 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 (that is, 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:

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

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

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 **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 **lam** $x.\ e_1'$ is represented as in LF as $\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)$, and the variable `E1' : exp -> exp` will be instantiated to `([x:exp]` $\ulcorner e_1' \urcorner$`)`. 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 **??**)—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 (**lam** $x.\ x$) **z** which is given by the deduction

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

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.
```

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 17, **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 17) that there is no v such that the judgment **fix** $x.\, x \hookrightarrow v$ is derivable.

It is also possible that evaluation fails finitely, although in our formulation of the language this is only possible for Mini-ML expressions that are not well-typed according to the Mini-ML typing rules. 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 17. 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.

```
%query 1 *
D : eval (app
            (fix [f:exp] lam [x:exp]
               (case x z ([x':exp] s (s (app f x')))))
            (s z))
    V.
```

This generates the following answer:

```
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).
```

The example above exhibits another feature of the Elf implementation. We can pose query in the form `%query` n k A`.` which solves the query A and verifies that it produces precisely n solutions after k tries. Here `*` as either n or k represents infinity.

One can also enter queries interactively after typing `top` in the Twelf server. Then, after after displaying the first solution for `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)).
Empty Substitution.
More? n
?- value (fst (pair z (s z))).
no
?- value (lam [x] (fst x)).
Empty Substitution.
More? y
No more solutions
```

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 can optimize 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.

$$\mathsf{vs} \quad : \quad \Pi E{:}\mathsf{exp}.\ \Pi V{:}\mathsf{exp}.\ \mathsf{eval}\ E\ V \rightarrow \mathsf{value}\ V \rightarrow \mathsf{type}.$$

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:

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

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,

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

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.

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

Its representation in LF is given by

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

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 corresponding 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 by using the declaration `%solve c : A`. This will solve the query `A` obtain the first solution `A'` with proof term `M` and then making the definition `c : A' = M`. Later queries can then refer to `c`. For example,

```
%solve d0 : eval (case z (s z) ([x:exp] z) (s z)).
%query 1 * vs d0 P.
```

will construct `d0` and then transform it to a value derivation using the higher-level judgment `vs` that implements value soundness.

4.5 Input and Output Modes

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 an object `M` of some instance of `A`. Therefore in some way Elf unifies the concepts of goal and logic variable. On the other hand, the intuition underlying the operational interpretation of judgments makes a clear distinction between construction of a derivation and unification: unification is employed only to see if an inference rule can be applied to reduce a goal to subgoals.

In Elf, the distinction between subgoals and logic variables is made based on the presence or absence of a true dependency. Recall that the only distinction between $\Pi x{:}A.\ B$ and $A \to B$ is that x may occur in B in the first form, but not in the second. For the purposes of the operational semantics of Elf, the truly dependent function type $\Pi x{:}A.\ B$ where x does in fact occur somewhere in B is treated by substituting a new logic variable for x which is subject to unification. The non-dependent function type $A \to B$ is treated by introducing A as a subgoal necessary for the solution of B.

Therefore, a typical constant declaration which has the form

$$c : \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ B_1 \to \cdots \to B_m.\ C$$

for an atomic type C, introduces logic variables $X_1, \ldots, X_n$, then finds most general common instance between the goal G and C (possibly instantiating $X_1, \ldots, X_n$ and then solves $B_m, \ldots, B_1$ as subgoals, in that order. Note that in practical programs, the quantifiers on $x_1, \ldots, x_n$ are often implicit.

When writing a program it is important to kept this interpretation in mind. In order to illustrate it, we write some simple programs.

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)`.

```
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.
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.
%mode append +L +K -M.

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

The mode declaration

```
%mode append +L +K -M.
```

specifies that, for the operational reading, we should consider the first two argument `L` and `K` as given input, while the third argument `M` is to be constructed by the program as output. To make this more precise, we define that a term is *ground* if it does not contain any logic variables. With the above mode declaration we specify that first two arguments l and k to `append` should always be ground when a goal of the form `append` l k m is to be solved. Secondly, it expresses that upon success, the third argument m should always be ground. The Elf compiler verifies this property as each declaration is read and issues an appropriate error message if it is violated.

This mode verification proceeds as follows. We first consider

```
ap_nil : append nil K K.
```

We may assume that the first two arguments are ground, that is, `nil` and `K` will be ground when `append` is invoked. Therefore, if this rule succeeds, the third argument `K` will indeed be ground.

Next we consider the second clause.

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

We may assume that the first two arguments are ground when `append` is invoked. Hence `X`, `L`, and `K` may be assumed to be ground. This is necessary to know that the recursive call `append L K M` is well-moded: `L` and `K` are indeed ground. Inductively, we may now assume that `M` is ground if this subgoal succeeds, since the third argument to `M` was designated as an output argument. Since we also already know that `X` is ground, we thus conclude that `(X ; M)` is ground. Therefore the declaration is well-moded.

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.
```

We can also use the same implementation to split a list into two parts by posing a query of the form `append L K` m for a given list m. This query constitutes a use of `append` in the mode

```
%mode append -L -K +M.
```

The reader may wish to analyze `append` to see why `append` also satisfies this mode declaration. We can now use the `%query` construct to verify that the actual and expected number of solutions coincide.

```
%query 4 *
append L K (0 ; s 0 ; s s 0 ; nil).
---------- Solution 1 ----------
K = 0 ; s 0 ; s s 0 ; nil;
L = nil.
---------- Solution 2 ----------
K = s 0 ; s s 0 ; nil;
L = 0 ; nil.
---------- Solution 3 ----------
K = s s 0 ; nil;
L = 0 ; s 0 ; nil.
---------- Solution 4 ----------
K = nil;
L = 0 ; s 0 ; s s 0 ; nil.
```

Mode-checking is a valuable tool for the programmer to check the correct definition and use of predicate. Incorrect use often leads to non-termination. For example, consider the following definition of the even numbers.

```
even : nat -> type.
%mode even +N.
even_ss : even (s (s N)) <- even N.
even_0 : even 0.
```

The mode declaration indicates that it should be used only to verify if a given (ground) natural numbers is even. Indeed, the query

```
?- even N.
```

will fail to terminate without giving a single answer. It is not mode-correct, since `N` is a logic variable in an input position. To see why this fails to terminate, we step through the execution:

```
?- even N.
Solving...
% Goal 1:
even N.
% Resolved with clause even_ss
N = s s N1.
% Solving subgoal (1) of clause even_ss
% Goal 2:
even N1.
% Resolved with clause even_ss
N1 = s s N2.
% Solving subgoal (1) of clause even_ss
% Goal 3:
even N2.
...
```

If definition of `even` was intended to enumerate all even numbers instead, we would exchange the order of the two declarations `even_0` and `even_ss`. We call this new family `even*`.

```
even* : nat -> type.
%mode even* -N.

even*_0 : even* 0.
even*_ss : even* (s (s N)) <- even* N.
```

The mode declaration now indicates that the argument of `even*` is no longer an input, but an output. Since the declarations are tried in order, execution now succeeds infinitely many times, starting with `0` as the first answer. The query

```
%query * 10 even* N.
```

enumerates the first 10 answers and then stops.

It is also possible to declare variables to be neither input nor output by using the pattern $*X$ for an argument X. This kind of pattern is used, for example, in the implementation of type inference in Section 5.5.

4.6 Exercises

Exercise 4.1 Show the sequence of subgoals generated by the query which attempts to evaluate the Mini-ML expression (**lam** $x.\ x\ x$) (**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 **case** e_1 **of** **z** $\Rightarrow e_2 \mid$ **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 2.12 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.

Bibliography

[AINP88] Peter B. Andrews, Sunil Issar, Daniel Nesmith, and Frank Pfenning. The TPS theorem proving system. In Ewing Lusk and Russ Overbeek, editors, *9th International Conference on Automated Deduction*, pages 760–761, Argonne, Illinois, May 1988. Springer-Verlag LNCS 310. System abstract.

[All75] William Allingham. *In Fairy Land*. Longmans, Green, and Co., London, England, 1875.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[Chu32] A. Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

[Chu33] A. Church. A set of postulates for the foundation of logic II. *Annals of Mathematics*, 34:839–864, 1933.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

[dB68] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, December 1968. Springer-Verlag LNM 125.

[DFH$^+$93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Conference Record of the 9th ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM Press, 1982.

[Dow93] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 139–145, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[DP91] Scott Dietzen and Frank Pfenning. A declarative alternative to assert in logic programming. In Vijay Saraswat and Kazunori Ueda, editors, *International Logic Programming Symposium*, pages 372–386. MIT Press, October 1991.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, pages 121–136, Chapel Hill, North Carolina, April 1989. Springer-Verlag LNCS 355.

[Ell90] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, July 1992. Available as Technical Report CST-93-92.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[Geu92] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1992.

[Han91] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, January 1991. Available as Technical Report MS-CIS-91-09.

[Han93] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[Har90] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[HB34] David Hilbert and Paul Bernays. *Grundlagen der Mathematik*. Springer-Verlag, Berlin, 1934.

[Her30] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Lettres de Varsovic*, 33, 1930.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM89] John Hannan and Dale Miller. A meta-logic for functional programming. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 24, pages 453–476. MIT Press, 1989.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HP00] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University, July 2000.

[Hue73] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987.

[Lia95] Chuck Liang. *Object-Level Substitutions, Unification and Generalization in Meta Logic*. PhD thesis, University of Pennsylvania, December 1995.

[Mai92] H.G. Mairson. Quantifier elimination and parametric polymorphism in programming languages. *Journal of Functional Programming*, 2(2):213–226, April 1992.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*, 17:348–375, August 1978.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[ML85] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[New65] Allen Newell. Limitations of the current stock of ideas about problem solving. In A. Kent and O. E. Taulbee, editors, *Electronic Information Handling*, pages 195–208, Washington, D.C., 1965. Spartan Books.

[NGdV94] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1994.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[Pfe91a] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe91b] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming.* MIT Press, Cambridge, Massachusetts, 1992.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[PW90] David Pym and Lincoln Wallen. Investigations into proof-search in a system of first-order dependent function types. In M.E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 236–250, Kaiserslautern, Germany, July 1990. Springer-Verlag LNCS 449.

[PW91] David Pym and Lincoln A. Wallen. Proof search in the $\lambda\Pi$-calculus. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 309–340. Cambridge University Press, 1991.

[Pym90] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, University of Edinburgh, 1990. Available as CST-69-90, also published as ECS-LFCS-90-125.

[Pym92] David Pym. A unification algorithm for the $\lambda\Pi$-calculus. *International Journal of Foundations of Computer Science*, 3(3):333–378, September 1992.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[Sal90] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, France, May 1990.

[Sch00] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[SH84] Peter Schroeder-Heister. A natural extension of natural deduction. *The Journal of Symbolic Logic*, 49(4):1284–1300, December 1984.

[Twe98] Twelf home page. Available at `http://www.cs.cmu.edu/~twelf`, 1998.