Chapter 3

Formalization in a Logical Framework

> We can look at the current field of problem solving by computers as a series of ideas about how to present a problem. If a problem can be cast into one of these representations in a natural way, then it is possible to manipulate it and stand some chance of solving it.
>
> — Allen Newell,
> *Limitations of the Current Stock of Ideas for Problem Solving* [New65]

In the previous chapter we have seen a typical application of deductive systems to specify and prove properties of programming languages. In this chapter we present techniques for the formalization of the languages and deductive systems involved. In the next chapter we show how these formalization techniques can lead to implementations.

The logical framework we use in these notes is called LF and sometimes ELF (for Edinburgh Logical Framework), not to be confused with Elf, which is the programming language based on the LF logical framework we introduce in Chapter **??**. LF was introduced by Harper, Honsell, and Plotkin [HHP93]. It has its roots in similar languages used in the project Automath [dB68, NGdV94]. LF has been explicitly designed as a meta-language for high-level specification of languages in logic and computer science and thus provides natural support for many of the techniques we have seen in the preceding chapter. For example, it can capture the convention that expressions that differ only in the names of bound variables are identified. Similarly, contexts and variable lookup as they arise in the typing judgment can be modelled concisely. The fact that these techniques are directly supported by the logical framework is not just a matter of engineering an implementation of the deductive systems in question, but it will be a crucial factor for the succinct implementation of proofs of meta-theorems such as type preservation.

By codifying formalization techniques into a meta-language, a logical framework also provides insight into principles of language presentation. Just as it is interesting to know if a mathematical proof depends on the axiom of choice or the law of excluded middle, a logical framework can be used to gauge the properties of the systems we are investigating.

The formalization task ahead of us consists of three common stages. The first stage is the representation of *abstract syntax* of the object language under investigation. For example, we need to specify the languages of expressions and types of Mini-ML. The second stage is the representation of the language *semantics*. This includes the static semantics (for example, the notion of value and the type system) and the dynamic semantics (for example, the operational semantics). The third stage is the representation of *meta-theory* of the language (for example, the proof of type preservation). Each of these uses its own set of techniques, some of which are explained in this chapter using the example of Mini-ML from the preceding chapter.

In the remainder of this chapter we introduce the framework in stages, always motivating new features using our example. The final summary of the system is given in Section 3.8 at the end of this chapter.

3.1 The Simply-Typed Fragment of LF

For the representation of the abstract syntax of a language, the simply-typed λ-calculus ($\lambda^{\rightarrow}$) is usually adequate. When we tackle the task of representing inference rules, we will have to refine the type system by adding dependent types. The reader should bear in mind that $\lambda^{\rightarrow}$ should *not* be considered as a functional programming language, but only as a representation language. In particular, the absence of recursion will be crucial in order to guarantee adequacy of representations. Our formulation of the simply-typed λ-calculus has two levels: the level of *types* and the level of *objects*, where types classify objects. Furthermore, we have *signatures* which declare type and object constants, and *contexts* which assign types to variables. Unlike Mini-ML, the presentation is given in the style of Church: every object will have a unique type. This requires that types appear in the syntax of objects to resolve the inherent ambiguity of certain functions, for example, the identity function. We let a range over type constants, c over object constants, and x over variables.

$$
\begin{array}{llcl}
\text{Types} & A & ::= & a \mid A_1 \rightarrow A_2 \\
\text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}\mathsf{type} \mid \Sigma, c{:}A \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}
$$

We make the general restriction that constants and variables can occur at most once in a signature or context, respectively. We will write $\Sigma(c) = A$ if $c{:}A$ occurs in Σ and $\Sigma(a) = \mathsf{type}$ if $a{:}\mathsf{type}$ occurs in Σ. Similarly $\Gamma(x) = A$ if $x{:}A$ occurs in Γ. We will use A and B to range over types, and M and N to range over objects. We refer to type constants a as *atomic types* and types of the form $A \to B$ as *function types*.

The judgments defining $\lambda^{\to}$ are

$\vdash_\Sigma A : \mathsf{type}$	A is a valid type
$\Gamma \vdash_\Sigma M : A$	M is a valid object of type A in context Γ
$\vdash \Sigma\ \mathit{Sig}$	Σ is a valid signature
$\vdash_\Sigma \Gamma\ \mathit{Ctx}$	Γ is a valid context

They are defined via the following inference rules.

$$\frac{\Sigma(c) = A}{\Gamma \vdash_\Sigma c : A}\,\mathsf{con} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash_\Sigma x : A}\,\mathsf{var}$$

$$\frac{\vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M : A \to B}\,\mathsf{lam}$$

$$\frac{\Gamma \vdash_\Sigma M : A \to B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : B}\,\mathsf{app}$$

$$\frac{\Sigma(a) = \mathsf{type}}{\vdash_\Sigma a : \mathsf{type}}\,\mathsf{tcon} \qquad \frac{\vdash_\Sigma A : \mathsf{type} \qquad \vdash_\Sigma B : \mathsf{type}}{\vdash_\Sigma A \to B : \mathsf{type}}\,\mathsf{arrow}$$

$$\frac{}{\vdash \cdot\ \mathit{Sig}}\,\mathsf{esig} \qquad \frac{\vdash \Sigma\ \mathit{Sig}}{\vdash \Sigma, a{:}\mathsf{type}\ \mathit{Sig}}\,\mathsf{tconsig}$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \qquad \vdash_\Sigma A : \mathsf{type}}{\vdash \Sigma, c{:}A\ \mathit{Sig}}\,\mathsf{consig}$$

$$\frac{}{\vdash_\Sigma \cdot\ \mathit{Ctx}}\,\mathsf{ectx} \qquad \frac{\vdash_\Sigma \Gamma\ \mathit{Ctx} \qquad \vdash_\Sigma A : \mathsf{type}}{\vdash_\Sigma \Gamma, x{:}A\ \mathit{Ctx}}\,\mathsf{varctx}$$

The rules for valid objects are somewhat non-standard in that they contain no check whether the signature Σ or the context Γ are valid. These are often added to the base cases, that is, the cases for variables and constants. We can separate the validity of signatures, since the signature Σ does not change in the rules for valid types and objects, Furthermore, the rules guarantee that if we have a derivation $\mathcal{D} :: \Gamma \vdash_{\Sigma} M : A$ and Γ is valid, then every context appearing in $\mathcal{D}$ is also valid. This is because the type A in the lam rule is checked for validity as it is added to the context. For an alternative formulation see Exercise 3.1.

Our formulation of the simply-typed λ-calculus above is parameterized by a signature in which new constants can be declared. In contrast, our formulation of Mini-ML has only a fixed set of constants and constructors. So far, we have left the dynamic semantics of $\lambda^{\rightarrow}$ unspecified. We later consider *canonical forms* as an analogue to Mini-ML values and conversion to canonical form as an analogue to evaluation. However, every well-typed $\lambda^{\rightarrow}$ object has a canonical form, while not every well-typed Mini-ML expression evaluates to a value. Moreover, we will start with a notion of *definitional equality* rather than an operational semantics. These differences illustrate that the similarity between Mini-ML as a programming language and $\lambda^{\rightarrow}$ as a representation language are rather superficial.

The notion of *definitional equality* for objects in $\lambda^{\rightarrow}$, written as $M \equiv N$, can be based on three conversions. The first is *α-conversion*: two objects are considered identical if they differ only in the names of their bound variables. The second is *β-conversion*: $(\lambda x{:}A.\ M)\ N \equiv [N/x]M$. It employs substitution $[N/x]M$ which renames bound variables to avoid variable capture. The third is derived from an extensionality principle. Roughly, two objects of functional type should be equal if applying them to equal arguments yields equal results. This can be incorporated by the rule of *η-conversion*: $(\lambda x{:}A.\ M\ x) \equiv M$ provided x does not occur free in M. The conversion rules can be applied to any subobject of an object M to obtain an object M' that is definitionally equal to M. Furthermore the relation of definitional equality is assumed to be an equivalence relation. We define the conversion judgment more formally in Section 3.8, once we have seen which role it plays in the logical framework.

3.2 Higher-Order Abstract Syntax

The first task in the formalization of a language in a logical framework is the representation of its expressions. We base the representation on abstract (rather than concrete) syntax in order to expose the essential structure of the object language so we can concentrate on semantics and meta-theory, rather than details of lexical analysis and parsing. The representation technique we use is called *higher-order abstract syntax*. It is supported by the simply-typed fragment $\lambda^{\rightarrow}$ of the logical framework LF. The idea of higher-order abstract syntax goes back to Church [Chu40] and has since been employed in a number of different contexts and guises. Church observed that once λ-notation is introduced into a language, all constructs that bind variables can be reduced to λ-abstraction. If we apply this principle in a setting where we distinguish a meta-language (the logical framework) from an object language (Mini-ML, in this example) then variables in the object language are represented by variables in the meta-language. Variables bound in the object language (by constructs such as **case**, **lam**, **let**, and **fix**) will be bound by λ in the meta-language. This has numerous advantages and a few disadvantages over the more immediate technique of representing variables by strings; some of the trade-offs are discussed in Section 3.10.

In the development below it is important not to confuse the typing of Mini-ML expressions with the type system employed by the logical framework, even though some overloading of notation is unavoidable. For example, ":" is used in both systems. For each (abstract) syntactic category of the object language we introduce a new type constant in the meta-language via a declaration of the form a:type. Thus, in order to represent Mini-ML expressions we declare a type exp in the meta-language. Since the representation techniques do not change when we generalize from the simply-typed λ-calculus to LF, we refer to the meta-language as LF throughout.

$$\mathsf{exp} \quad : \quad \mathsf{type}$$

We intend that every LF object M of type exp represents a Mini-ML expression and *vice versa*. The Mini-ML constant $\mathbf{z}$ is now represented by an LF constant z declared in the meta-language to be of type exp.

$$\mathsf{z} \quad : \quad \mathsf{exp}$$

The successor $\mathbf{s}$ is an expression constructor. It is represented by a constant of functional type that maps expressions to expressions so that, for example, $\mathsf{s}\ \mathsf{z}$ has type exp.

$$\mathsf{s} \quad : \quad \mathsf{exp} \to \mathsf{exp}$$

We now introduce the function $\ulcorner\cdot\urcorner$ which maps Mini-ML expressions to their representation in the logical framework. Later we will use $\ulcorner\cdot\urcorner$ generically for representation functions. So far we have

$$\begin{array}{rcl} \ulcorner \mathbf{z} \urcorner & = & \mathsf{z} \\ \ulcorner \mathbf{s}\ e \urcorner & = & \mathsf{s}\ \ulcorner e \urcorner. \end{array}$$

We would like to achieve that $\ulcorner e \urcorner$ has type exp in LF, given appropriate declarations for constants representing Mini-ML expression constructors. The constructs that do not introduce bound variables can be treated in a straightforward manner.

$$
\begin{array}{rcllcl}
\ulcorner \mathbf{z} \urcorner & = & \mathsf{z} & \mathsf{z} & : & \mathsf{exp} \\
\ulcorner \mathbf{s}\; e \urcorner & = & \mathsf{s}\, \ulcorner e \urcorner & \mathsf{s} & : & \mathsf{exp} \to \mathsf{exp} \\
\ulcorner \langle e_1, e_2 \rangle \urcorner & = & \mathsf{pair}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner & \mathsf{pair} & : & \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \\
\ulcorner \mathbf{fst}\; e \urcorner & = & \mathsf{fst}\, \ulcorner e \urcorner & \mathsf{fst} & : & \mathsf{exp} \to \mathsf{exp} \\
\ulcorner \mathbf{snd}\; e \urcorner & = & \mathsf{snd}\, \ulcorner e \urcorner & \mathsf{snd} & : & \mathsf{exp} \to \mathsf{exp} \\
\ulcorner e_1\; e_2 \urcorner & = & \mathsf{app}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner & \mathsf{app} & : & \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}
\end{array}
$$

Traditionally, one might now represent **lam** $x.\ e$ by $\mathsf{lam}\, \ulcorner x \urcorner\, \ulcorner e \urcorner$, where $\ulcorner x \urcorner$ may be a string or have some abstract type of identifier. This approach leads to a relatively low-level representation, since renaming of bound variables, capture-avoiding substitution, *etc.* as given in Section 2.2 now need to be axiomatized explicitly. Using higher-order abstract syntax means that variables of the object language (the language for which we are designing a representation) are represented by variables in the meta-language (the logical framework). Variables bound in the object language must then be bound correspondingly in the meta-language. As a first and immediate benefit, expressions which differ only in the names of their bound variables will be α-convertible in the meta-language. This leads to the representation

$$
\begin{array}{rclcll}
\ulcorner x \urcorner & = & x & & & \\
\ulcorner \mathbf{lam}\; x.\ e \urcorner & = & \mathsf{lam}\; (\lambda x{:}\mathsf{exp}.\ \ulcorner e \urcorner) & \mathsf{lam} & : & (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}.
\end{array}
$$

Recall that LF requires explicit types wherever variables are bound by λ, and free variables must be assigned a type in a context. Note also that the two occurrences of x in the first line above represent two variables with the same name in different languages, Mini-ML and LF. One can allow explicit renaming in the translation, but it complicates the presentation unnecessarily. The four remaining Mini-ML constructs, **case**, **let val**, **let name**, and **fix**, also introduce binding operators. Their representation follows the scheme for **lam**, taking care that variables bound in Mini-ML are also bound at the meta-level and have proper scope. For example, the representation of **let val** $x = e_1$ **in** e_2 reflects that x is bound in e_2 but not in e_1.

$$
\begin{array}{rcl}
\ulcorner \mathbf{case}\; e_1\; \mathbf{of}\; \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\; x \Rightarrow e_3 \urcorner & = & \mathsf{case}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner\, (\lambda x{:}\mathsf{exp}.\ \ulcorner e_3 \urcorner) \\
\ulcorner \mathbf{let\ val}\; x = e_1\; \mathbf{in}\; e_2 \urcorner & = & \mathsf{letv}\, \ulcorner e_1 \urcorner\, (\lambda x{:}\mathsf{exp}.\ \ulcorner e_2 \urcorner) \\
\ulcorner \mathbf{let\ name}\; x = e_1\; \mathbf{in}\; e_2 \urcorner & = & \mathsf{letn}\, \ulcorner e_1 \urcorner\, (\lambda x{:}\mathsf{exp}.\ \ulcorner e_2 \urcorner) \\
\ulcorner \mathbf{fix}\; x.\ e \urcorner & = & \mathsf{fix}\, (\lambda x{:}\mathsf{exp}.\ \ulcorner e \urcorner)
\end{array}
$$

Hence we have

$$
\begin{array}{lcl}
\mathsf{case} & : & \mathsf{exp} \to \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \\
\mathsf{letv} & : & \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \\
\mathsf{letn} & : & \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \\
\mathsf{fix} & : & (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}.
\end{array}
$$

As an example, consider the program *double* from page 17.

$$\begin{array}{l}\ulcorner \mathbf{fix}\ f.\ \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (f\ x'))\urcorner \\ \quad = \mathsf{fix}\ (\lambda f{:}\mathsf{exp}.\ \mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \mathsf{case}\ x\ \mathsf{z}\ (\lambda x'{:}\mathsf{exp}.\ \mathsf{s}\ (\mathsf{s}\ (\mathsf{app}\ f\ x')))))\end{array}$$

One can easily see that the object on the right-hand side is valid and has type exp, given the constant declarations above.

The next step will be to formulate (and later prove) what this representation accomplishes, namely that every expression has a representation, and every LF object of type exp constructed with constants from the signature above represents an expression. In practice we want a stronger property, namely that the representation function is a *compositional bijection*, something we will return to later in this chapter in Section 3.3.

Recall that $\vdash_\Sigma$ is the typing judgment of LF. We fix the signature E to contain all declarations above starting with $\mathsf{exp}{:}\mathsf{type}$ through $\mathsf{fix}{:}(\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$. At first it might appear that we should be able to prove:

1. For any Mini-ML expression e, $\vdash_E \ulcorner e \urcorner : \mathsf{exp}$.
2. For any LF object M such that $\vdash_E M : \mathsf{exp}$, there is a Mini-ML expression e such that $\ulcorner e \urcorner = M$.

As stated, neither of these two propositions is true. The first one fails due to the presence of free variables in e and therefore in $\ulcorner e \urcorner$ (recall that object-language variables are represented as meta-language variables). The second property fails because there are many objects M of type exp that are not in the image of $\ulcorner \cdot \urcorner$. Consider, for example, $(\lambda x{:}\mathsf{exp}.\ x)\ \mathsf{z}$ for which it is easy to show that

$$\vdash_E (\lambda x{:}\mathsf{exp}.\ x)\ \mathsf{z} : \mathsf{exp}.$$

Examining the representation function reveals that the resulting LF objects contain no β-redices, that is, no objects of the form $(\lambda x{:}A.\ M)\ N$.

A more precise analysis later yields the related notion of *canonical form*. Taking into account free variables and restricting ourselves to canonical forms (yet to be defined), we can reformulate the proposition expressing the correctness of the representation.

1. Let e be a Mini-ML expression with free variables among $x_1, \ldots, x_n$. Then $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_E \ulcorner e \urcorner : \mathsf{exp}$, and $\ulcorner e \urcorner$ is in canonical form.
2. For any canonical form M such that $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_E M : \mathsf{exp}$ there is a Mini-ML expression e with free variables among $x_1, \ldots, x_n$ such that $\ulcorner e \urcorner = M$.

It is a deep property of LF that every valid object is definitionally equal to a unique canonical form. Thus, if we want to answer the question which Mini-ML expression is represented by a non-canonical object M of type exp, we convert it to canonical form M' and determine the expression e represented directly by M'.

The definition of canonical form is based on two observations regarding the inverse of the representation function. The first is that if we are considering an LF object M of type exp we can read off the top-level constructor (the alternative in the definition of Mini-ML expressions) if the term has the form $c\ M_1 \ldots M_n$, where c is one of the LF constants in the signature defining Mini-ML expressions. For example, if M has the form $(\mathsf{s}\ M_1)$ we know that M represents an expression of the form $\mathbf{s}\ e_1$, where M_1 is the representation of e_1.

The second observation is less obvious. Let us consider an LF object of type $\mathsf{exp} \to \mathsf{exp}$. Such objects arise in the representation, for example, in the second argument to letv, which has type $\mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$. For example,

$$\ulcorner \mathbf{let\ val}\ x = \mathbf{s\ z\ in}\ \langle x, x\rangle \urcorner = \mathsf{letv}\ (\mathsf{s\ z})\ (\lambda x{:}\mathsf{exp}.\ \mathsf{pair}\ x\ x).$$

The argument $(\lambda x{:}\mathsf{exp}.\ \mathsf{pair}\ x\ x)$ represents the body of the **let**-expression, abstracted over the **let**-bound variable x. Since we model the scope of a bound variable in the object language by the scope of a corresponding λ-abstraction in the meta-language, we always expect an object of type $\mathsf{exp} \to \mathsf{exp}$ to be a λ-abstraction. As a counterexample consider the object

$$\mathsf{letv}\ (\mathsf{pair}\ (\mathsf{s\ z})\ \mathsf{z})\ \mathsf{fst}$$

which is certainly well-typed in LF and has type exp, since $\mathsf{fst} : \mathsf{exp} \to \mathsf{exp}$. This object is not the image of any expression e under the representation function $\ulcorner\cdot\urcorner$. However, there is an η-equivalent object, namely

$$\mathsf{letv}\ (\mathsf{pair}\ (\mathsf{s\ z})\ \mathsf{z})\ (\lambda x{:}\mathsf{exp}.\ \mathsf{fst}\ x)$$

which represents $\mathbf{let\ val}\ x = \langle \mathbf{s\ z}, \mathbf{z}\rangle\ \mathbf{in\ fst}\ x$.

We can summarize these two observations as the following statement constraining our definition of canonical forms.

1. A canonical object of type exp should either be a variable or have the form $c\ M_1\ \ldots M_n$, where $M_1, \ldots, M_n$ are again canonical; and

2. a canonical object of type $\mathsf{exp} \to \mathsf{exp}$ should have the form $\lambda x{:}\mathsf{exp}.\ M_1$, where M_1 is again canonical.

Returning to an earlier counterexample, $((\lambda x{:}\mathsf{exp}.\ x)\ \mathsf{z})$, we notice that it is not canonical, since it is of atomic type (exp), but does not have the form of a constant applied to some arguments. In this case, there is a β-equivalent object which is canonical form, namely z. In general each valid object has a $\beta\eta$-equivalent object in canonical form, but this is a rather deep theorem about LF.

For the representation of more complicated languages, we have to generalize the observations above and allow an arbitrary number of type constants (rather than just exp) and allow arguments to variables. We write the general judgment as

$$\Gamma \vdash_\Sigma M \Uparrow A \qquad\qquad M \text{ is canonical of type } A.$$

This judgment is defined by the following inference rules. Recall that a stands for constants at the level of types.

$$\frac{\vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M \Uparrow B}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M \Uparrow A \to B}\,\mathsf{carrow}$$

$$\frac{\Sigma(c) = A_1 \to \cdots \to A_n \to a \qquad \Gamma \vdash_\Sigma M_1 \Uparrow A_1 \quad \ldots \quad \Gamma \vdash_\Sigma M_n \Uparrow A_n}{\Gamma \vdash_\Sigma c\, M_1 \ldots M_n \Uparrow a}\,\mathsf{conapp}$$

$$\frac{\Gamma(x) = A_1 \to \cdots \to A_n \to a \qquad \Gamma \vdash_\Sigma M_1 \Uparrow A_1 \quad \ldots \quad \Gamma \vdash_\Sigma M_n \Uparrow A_n}{\Gamma \vdash_\Sigma x\, M_1 \ldots M_n \Uparrow a}\,\mathsf{varapp}$$

This judgment singles out certain valid objects, as the following theorem shows.

Theorem 3.1 (Validity of Canonical Objects) *Let Σ be a valid signature and Γ a context valid in Σ. If $\Gamma \vdash_\Sigma M \Uparrow A$ then $\Gamma \vdash_\Sigma M : A$.*

Proof: See Exercise 3.2 and Section 3.9. □

The simply-typed λ-calculus we have introduced so far has some important properties. In particular, type-checking is decidable, that is, it is decidable if a given object is valid. It is also decidable if a given object is in canonical form, and every well-typed object can effectively be converted to a unique canonical form. Further discussion and proof of these and other properties can be found in Section **??**.

3.3 Representing Mini-ML Expressions

In order to obtain a better understanding of the representation techniques, it is worthwile to state in full detail and carry out the proofs that the representation of Mini-ML introduced in this chapter is correct. First, we summarize the representation function and the signature E defining the abstract syntax of Mini-ML.

$$
\begin{array}{rcl}
\ulcorner \mathbf{z} \urcorner & = & \mathsf{z} \\
\ulcorner \mathbf{s}\, e \urcorner & = & \mathsf{s}\, \ulcorner e \urcorner \\
\ulcorner \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\, x \Rightarrow e_3 \urcorner & = & \mathsf{case}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner\, (\lambda x{:}\mathsf{exp}.\, \ulcorner e_3 \urcorner) \\
\ulcorner \langle e_1, e_2 \rangle \urcorner & = & \mathsf{pair}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner \\
\ulcorner \mathbf{fst}\, e \urcorner & = & \mathsf{fst}\, \ulcorner e \urcorner \\
\ulcorner \mathbf{snd}\, e \urcorner & = & \mathsf{snd}\, \ulcorner e \urcorner \\
\ulcorner \mathbf{lam}\, x.\, e \urcorner & = & \mathsf{lam}\, (\lambda x{:}\mathsf{exp}.\, \ulcorner e \urcorner) \\
\ulcorner e_1\, e_2 \urcorner & = & \mathsf{app}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner \\
\ulcorner \mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 \urcorner & = & \mathsf{letv}\, \ulcorner e_1 \urcorner\, (\lambda x{:}\mathsf{exp}.\, \ulcorner e_2 \urcorner) \\
\ulcorner \mathbf{let\ name}\ x = e_1\ \mathbf{in}\ e_2 \urcorner & = & \mathsf{letn}\, \ulcorner e_1 \urcorner\, (\lambda x{:}\mathsf{exp}.\, \ulcorner e_2 \urcorner) \\
\ulcorner \mathbf{fix}\, x.\, e \urcorner & = & \mathsf{fix}\, (\lambda x{:}\mathsf{exp}.\, \ulcorner e \urcorner) \\
\ulcorner x \urcorner & = & x
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{exp} & : & \mathsf{type} \\
\mathsf{z} & : & \mathsf{exp} \\
\mathsf{s} & : & \mathsf{exp} \to \mathsf{exp} \\
\mathsf{case} & : & \mathsf{exp} \to \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \\
\mathsf{pair} & : & \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \\
\mathsf{fst} & : & \mathsf{exp} \to \mathsf{exp} \\
\mathsf{snd} & : & \mathsf{exp} \to \mathsf{exp} \\
\mathsf{lam} & : & (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \\
\mathsf{app} & : & \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \\
\mathsf{let} & : & \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \\
\mathsf{fix} & : & (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}
\end{array}
$$

Lemma 3.2 (Validity of Representation) *For any context $\Gamma = x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}$ and Mini-ML expression e with free variables among $x_1, \ldots, x_n$,*

$$\Gamma \vdash_E \ulcorner e \urcorner \Uparrow \mathsf{exp}$$

Proof: The proof is a simple induction on the structure of e. We show three representative cases—the others follow similarly.

Case: $e = \mathbf{z}$. Then $\ulcorner \mathbf{z} \urcorner = \mathsf{z}$ and $\Gamma \vdash_E \mathsf{z} \Uparrow \mathsf{exp}$.

Case: $e = e_1\ e_2$. Then $\ulcorner e \urcorner = \mathsf{app}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner$. By induction hypothesis there are derivations

$$
\begin{array}{lcl}
\mathcal{D}_1 & :: & \Gamma \vdash_E \ulcorner e_1 \urcorner \Uparrow \mathsf{exp}, \text{ and} \\
\mathcal{D}_2 & :: & \Gamma \vdash_E \ulcorner e_2 \urcorner \Uparrow \mathsf{exp}.
\end{array}
$$

Since $E(\mathsf{app}) = \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$ we can apply rule conapp from the definition of canonical forms to $\mathcal{D}_1$ and $\mathcal{D}_2$ to conclude that

$$\Gamma \vdash_E \mathsf{app}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner \Uparrow \mathsf{exp}$$

is derivable.

Case: $e = (\mathbf{let\,val}\ x = e_1\ \mathbf{in}\ e_2)$. Then $\ulcorner e \urcorner = \mathsf{let}\ \ulcorner e_1 \urcorner\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_2 \urcorner)$. Note that if e has free variables among $x_1, \ldots, x_n$, then e_2 has free variables among $x_1, \ldots, x_n, x$. Hence, by induction hypothesis, we have derivations

$$\begin{array}{lll} \mathcal{D}_1 & :: & \Gamma \vdash_E \ulcorner e_1 \urcorner \Uparrow \mathsf{exp}, \text{ and} \\ \mathcal{D}_2 & :: & \Gamma, x{:}\mathsf{exp} \vdash_E \ulcorner e_2 \urcorner \Uparrow \mathsf{exp}. \end{array}$$

Applying rule carrow yields the derivation

$$\dfrac{\dfrac{E(\mathsf{exp}) = \mathsf{type}}{\vdash_E \mathsf{exp} : \mathsf{type}}\,\mathsf{con} \qquad \begin{array}{c}\mathcal{D}_2 \\ \Gamma, x{:}\mathsf{exp} \vdash_E \ulcorner e_2 \urcorner \Uparrow \mathsf{exp}\end{array}}{\Gamma \vdash_E \lambda x{:}\mathsf{exp}.\ \ulcorner e_2 \urcorner \Uparrow \mathsf{exp} \to \mathsf{exp}}\,\mathsf{carrow}$$

Using this derivation, $E(\mathsf{let}) = \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$, derivation $\mathcal{D}_1$ and rule conapp yields a derivation of

$$\Gamma \vdash_E \mathsf{let}\ \ulcorner e_1 \urcorner\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_2 \urcorner) \Uparrow \mathsf{exp},$$

which is what we needed to show.

□

Next we define the inverse of the representation function, $\llcorner \cdot \lrcorner$. We need to keep in mind that it only needs to be defined on canonical forms of type exp.

$$\begin{array}{rcl}
\llcorner \mathsf{z} \lrcorner & = & \mathbf{z} \\
\llcorner \mathsf{s}\ M \lrcorner & = & \mathbf{s}\ \llcorner M \lrcorner \\
\llcorner \mathsf{case}\ M_1\ M_2\ (\lambda x{:}\mathsf{exp}.\ M_3) \lrcorner & = & \mathbf{case}\ \llcorner M_1 \lrcorner\ \mathbf{of}\ \mathbf{z} \Rightarrow \llcorner M_2 \lrcorner \mid \mathbf{s}\ x \Rightarrow \llcorner M_3 \lrcorner \\
\llcorner \mathsf{pair}\ M_1\ M_2 \lrcorner & = & \langle \llcorner M_1 \lrcorner, \llcorner M_2 \lrcorner \rangle \\
\llcorner \mathsf{fst}\ M \lrcorner & = & \mathbf{fst}\ \llcorner M \lrcorner \\
\llcorner \mathsf{snd}\ M \lrcorner & = & \mathbf{snd}\ \llcorner M \lrcorner \\
\llcorner \mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ M) \lrcorner & = & \mathbf{lam}\ x.\ \llcorner M \lrcorner \\
\llcorner \mathsf{app}\ M_1\ M_2 \lrcorner & = & \llcorner M_1 \lrcorner\ \llcorner M_2 \lrcorner \\
\llcorner \mathsf{letv}\ M_1\ (\lambda x{:}\mathsf{exp}.\ M_2) \lrcorner & = & \mathbf{let\,val}\ x = \llcorner M_1 \lrcorner\ \mathbf{in}\ \llcorner M_2 \lrcorner \\
\llcorner \mathsf{letn}\ M_1\ (\lambda x{:}\mathsf{exp}.\ M_2) \lrcorner & = & \mathbf{let\,name}\ x = \llcorner M_1 \lrcorner\ \mathbf{in}\ \llcorner M_2 \lrcorner \\
\llcorner \mathsf{fix}\ (\lambda x{:}\mathsf{exp}.\ M) \lrcorner & = & \mathbf{fix}\ x.\ \llcorner M \lrcorner \\
\llcorner x \lrcorner & = & x
\end{array}$$

Lemma 3.3 *For any* $\Gamma = x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}$ *and* M *such that* $\Gamma \vdash_E M \Uparrow \mathsf{exp}$, $\llcorner M \lrcorner$ *is defined and yields a Mini-ML expression such that* $\ulcorner \llcorner M \lrcorner \urcorner = M$.

Proof: The proof is by induction on the structure of the derivation $\mathcal{D}$ of $\Gamma \vdash_E M \Uparrow \mathsf{exp}$. Note that $\mathcal{D}$ cannot end with an application of the carrow rule, since exp is atomic.

Case: $\mathcal{D}$ ends in varapp. From the form of Γ we know that $x = x_i$ for some i and x has no arguments. Hence $\llcorner M \lrcorner = \llcorner x \lrcorner = x$ is defined.

Case: $\mathcal{D}$ ends in conapp. Then c must be one of the constants in E. We now further distinguish subcases, depending on c. We only show three subcases; the others follow similarly.

Subcase: $c = \mathsf{z}$. Then c has no arguments and $\llcorner M \lrcorner = \llcorner \mathsf{z} \lrcorner = \mathbf{z}$, which is a Mini-ML expression. Furthermore, $\ulcorner \mathbf{z} \urcorner = \mathsf{z}$.

Subcase: $c = \mathsf{app}$. Then c has two arguments, $\llcorner M \lrcorner = \llcorner \mathsf{app}\ M_1\ M_2 \lrcorner = \llcorner M_1 \lrcorner\ \llcorner M_2 \lrcorner$, and, suppressing the premise $E(\mathsf{app}) = \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$, $\mathcal{D}$ has the form

$$\dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \vdash_E M_1 \Uparrow \mathsf{exp}\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \Gamma \vdash_E M_2 \Uparrow \mathsf{exp}\end{array}}{\Gamma \vdash_E \mathsf{app}\ M_1\ M_2 \Uparrow \mathsf{exp}}\ \mathsf{conapp}$$

By the induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2$, $\llcorner M_1 \lrcorner$ and $\llcorner M_2 \lrcorner$ are defined and therefore $\llcorner M \lrcorner = \llcorner M_1 \lrcorner\ \llcorner M_2 \lrcorner$ is also defined. Furthermore, $\ulcorner \llcorner M \lrcorner \urcorner = \ulcorner \llcorner M_1 \lrcorner\ \llcorner M_2 \lrcorner \urcorner = \mathsf{app}\ \ulcorner \llcorner M_1 \lrcorner \urcorner\ \ulcorner \llcorner M_2 \lrcorner \urcorner = \mathsf{app}\ M_1\ M_2$, where the last equality follows by the induction hypothesis on M_1 and M_2.

Subcase: $c = \mathsf{letv}$. Then c has two arguments and, suppressing the premise $E(\mathsf{letv}) = \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$, $\mathcal{D}$ has the form

$$\dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \vdash_E M_1 \Uparrow \mathsf{exp}\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \Gamma \vdash_E M_2 \Uparrow \mathsf{exp} \to \mathsf{exp}\end{array}}{\Gamma \vdash_E \mathsf{letv}\ M_1\ M_2 \Uparrow \mathsf{exp}}\ \mathsf{conapp}$$

There is only one inference rule which could have been used as the last inference in $\mathcal{D}_2$, namely carrow. Hence, by inversion, $\mathcal{D}_2$ must have the form

$$\dfrac{\begin{array}{c}\mathcal{D}_2' \\ \Gamma, x{:}\mathsf{exp} \vdash_E M_2' \Uparrow \mathsf{exp}\end{array}}{\Gamma \vdash_E \lambda x{:}\mathsf{exp}.\ M_2' \Uparrow (\mathsf{exp} \to \mathsf{exp})}\ \mathsf{carrow}$$

where $M_2 = \lambda x{:}\mathsf{exp}.\ M_2'$. Then

$$\llcorner M \lrcorner = \llcorner \mathsf{letv}\ M_1\ (\lambda x{:}\mathsf{exp}.\ M_2') \lrcorner = (\mathbf{let\ val}\ x = \llcorner M_1 \lrcorner\ \mathbf{in}\ \llcorner M_2' \lrcorner)$$

which is a Mini-ML expression by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2'$. We reason as in the previous cases that here, too, $\ulcorner \llcorner M \lrcorner \urcorner = M$.

□

Lemma 3.4 *For any Mini-ML expression e, $\llcorner \ulcorner e \urcorner \lrcorner = e$.*

Proof: The proof is a simple induction over the structure of e (see Exercise 3.3). □

The final lemma of this section asserts *compositionality* of the representation function, connecting meta-level substitution with object-level substitution. We only state this lemma for substitution of a single variable, but other, more general variants are possible. This lemma gives a formal expression to the statement that the representation of a compound expression is constructed from the representations of its immediate constituents. Note that in the statement of the lemma, the substitution on the left-hand side of the equation is substitution in the Mini-ML language as defined in Section 2.2, while on the right-hand side we have substitution at the level of the framework.

Lemma 3.5 (Compositionality) $\ulcorner [e_1/x]e_2 \urcorner = [\ulcorner e_1 \urcorner / x]\ulcorner e_2 \urcorner$.

Proof: The proof is by induction on the structure of e_2. We show three cases—the remaining ones follow the same pattern.

Case: $e_2 = x$. Then

$$\ulcorner [e_1/x]e_2 \urcorner = \ulcorner [e_1/x]x \urcorner = \ulcorner e_1 \urcorner = [\ulcorner e_1 \urcorner / x]x = [\ulcorner e_1 \urcorner / x]\ulcorner e_2 \urcorner.$$

Case: $e_2 = y$ and $y \neq x$. Then

$$\ulcorner [e_1/x]e_2 \urcorner = \ulcorner [e_1/x]y \urcorner = y = [\ulcorner e_1 \urcorner / x]y = [\ulcorner e_1 \urcorner / x]\ulcorner e_2 \urcorner.$$

Case: $e_2 = (\textbf{let val}\ y = e_2'\ \textbf{in}\ e_2'')$, where $y \neq x$ and y is not free in e_1. Note that this condition can always be achieved via renaming of the bound variable y. Then

$$\begin{array}{ll}
= & \ulcorner [e_1/x]e_2 \urcorner \\
= & \ulcorner [e_1/x](\textbf{let val}\ y = e_2'\ \textbf{in}\ e_2'') \urcorner \\
= & \ulcorner \textbf{let val}\ y = [e_1/x]e_2'\ \textbf{in}\ [e_1/x]e_2'' \urcorner \\
= & \mathsf{letv}\ \ulcorner [e_1/x]e_2' \urcorner\ (\lambda y{:}\mathsf{exp}.\ \ulcorner [e_1/x]e_2'' \urcorner) \\
= & \mathsf{letv}\ ([\ulcorner e_1 \urcorner/x]\ulcorner e_2' \urcorner)\ (\lambda y{:}\mathsf{exp}.\ [\ulcorner e_1 \urcorner/x]\ulcorner e_2'' \urcorner) \quad \text{by induction hypothesis} \\
= & [\ulcorner e_1 \urcorner/x](\mathsf{letv}\ \ulcorner e_2' \urcorner\ (\lambda y{:}\mathsf{exp}.\ \ulcorner e_2'' \urcorner)) \\
= & [\ulcorner e_1 \urcorner/x]\ulcorner \textbf{let val}\ y = e_2'\ \textbf{in}\ e_2'' \urcorner \\
= & [\ulcorner e_1 \urcorner/x]\ulcorner e_2 \urcorner.
\end{array}$$

□

We usually summarize Lemmas 3.2, 3.3, 3.4, and 3.5 into a single *adequacy theorem*, whose proof is is immediate from the preceding lemmas.

Theorem 3.6 (Adequacy) *There is a bijection* $\ulcorner\cdot\urcorner$ *between Mini-ML expressions with free variables among* $x_1, \ldots, x_n$ *and (canonical) LF objects* M *such that*

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_E M \Uparrow \mathsf{exp}$$

is derivable. The bijection is compositional in the sense that

$$\ulcorner[e_1/x]e_2\urcorner = [\ulcorner e_1\urcorner/x]\ulcorner e_2\urcorner.$$

3.4 Judgments as Types

So far, we have only discussed the representation of the abstract syntax of a language, taking advantage of the expressive power of the simply-typed λ-calculus. The next step is the representation of deductions. The general approach is to represent deductions as objects and judgments as types. For example, given closed expressions e and v and a deduction

$$\begin{array}{c}\mathcal{D}\\ e \hookrightarrow v\end{array}$$

we would like to establish that

$$\vdash_{EV} \ulcorner\mathcal{D}\urcorner \Uparrow \ulcorner e \hookrightarrow v\urcorner,$$

where $\ulcorner\cdot\urcorner$ is again a representation function and EV is an LF signature from which the constants in $\ulcorner\mathcal{D}\urcorner$ are drawn. That is, the representation of $\mathcal{D}$ is a canonical object of type $\ulcorner e \hookrightarrow v\urcorner$. The main difficulty will be achieving the converse, namely that if

$$\vdash_{EV} M \Uparrow \ulcorner e \hookrightarrow v\urcorner$$

then there is a deduction $\mathcal{D}$ such that $\ulcorner\mathcal{D}\urcorner = M$.

As a first approximation, assume we declare a type eval of evaluations, similar to the way we declared a type exp of Mini-ML expressions.

$$\mathsf{eval} \quad : \quad \mathsf{type}$$

An axiom would simply be represented as a constant of type eval. An inference rule can be viewed as a constructor which, given deductions of the premises, yields a deduction of the conclusion. For example, the rules

$$\frac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z} \qquad \frac{e \hookrightarrow v}{\mathbf{s}\ e \hookrightarrow \mathbf{s}\ v}\,\mathsf{ev\_s}$$

$$\frac{e_1 \hookrightarrow \mathbf{z} \qquad e_2 \hookrightarrow v}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \hookrightarrow v}\,\mathsf{ev\_case\_z}$$

would be represented by

$$\begin{array}{lcl} \mathsf{ev\_z} & : & \mathsf{eval} \\ \mathsf{ev\_s} & : & \mathsf{eval} \to \mathsf{eval} \\ \mathsf{ev\_case\_z} & : & \mathsf{eval} \to \mathsf{eval} \to \mathsf{eval}. \end{array}$$

One can easily see that this representation is not faithful: the declaration of a constant in the signature contains much less information than the statement of the inference rule. For example,

$$\vdash_{EV} \mathsf{ev\_case\_z}\ (\mathsf{ev\_s}\ \mathsf{ev\_z})\ \mathsf{ev\_z} \Uparrow \mathsf{eval}$$

would be derivable, but the object above does not represent a valid evaluation. The problem is that the first premise of the rule $\mathsf{ev\_case\_z}$ must be an evaluation yielding $\mathbf{z}$, while the corresponding argument to $\mathsf{ev\_case\_z}$, namely $(\mathsf{ev\_s}\ \mathsf{ev\_z})$, represents an evaluation yielding $\mathbf{s}\ \mathbf{z}$.

One solution to this representation problem is to introduce a validity predicate and define when a given object of type eval represents a valid deduction. This is, for example, the solution one would take in a framework such as higher-order Horn clauses or hereditary Harrop formulas. This approach is discussed in a number of papers [MNPS91, Pau86] and also is the basis for the logic programming language λProlog [NM99] and the theorem prover Isabelle [Pau94]. Here we take a different approach in that we refine the type system instead in such a way that only the representations of valid deductions (evaluations, in this example) will be well-typed in the meta-language. This has a number of methodological advantages. Perhaps the most important is that checking the validity of a deduction is reduced to a type-checking problem in the logical framework. Since LF type-checking is decidable, this means that checking deductions of the object language is automatically decidable, once a suitable representation in LF has been chosen.

But how do we refine the type system so that the counterexample above is rejected as ill-typed? It is clear that we have to subdivide the type of all evaluations into an infinite number of subtypes: for any expression e and value v there should be a type of deductions of $e \hookrightarrow v$. Of course, many of of these types should be empty. For example, there is no deduction of the judgment $\mathbf{s}\ \mathbf{z} \hookrightarrow \mathbf{z}$. These considerations lead to the view that eval is a *type family* indexed by representations of e and v.

Following our representation methodology, both of these will be LF objects of type exp. Thus we have *types*, such as (eval z z) which depend on *objects*, a situation which can easily lead to an undecidable type system. In the case of LF we can preserve decidability of type-checking (see Section 3.5). A first approximation to a revision of the representation for evaluations above would be

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

The declarations of ev_s and ev_case_z are schematic in the sense that they are intended to represent all instances with valid objects E, E_1, E_2, E_3, and V of appropriate type. With these declarations the object (ev_case_z (ev_s ev_z) ev_z) is no longer well-typed, since (ev_s ev_z) has type eval (s z) (s z), while the first argument to ev_case_z should have type eval E_1 z for some E_1.

Although it is not apparent in this example, allowing unrestricted schematic declarations would lead to an undecidable type-checking problem for LF, since it would require a form of higher-order unification. Instead we add E_1, E_2, E_3, and V as explicit arguments to ev_case_z. In practice this is often unnecessary and the Elf programming language allows schematic declarations in the form above and performs type reconstruction. A simple function type (formed by $\to$) is not expressive enough to capture the dependencies between the various arguments. For example,

$$
\begin{array}{lcl}
\mathsf{ev\_case\_z} & : & \mathsf{exp} \to \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \\
& & \quad \to \mathsf{eval}\ E_1\ \mathsf{z} \to \mathsf{eval}\ E_2\ V \to \mathsf{eval}\ (\mathsf{case}\ E_1\ E_2\ E_3)\ V
\end{array}
$$

does not express that the first argument is supposed to be E_1, the second argument E_2, *etc.* Thus we must explicitly label the first four arguments: this is what the *dependent function type* constructor Π achieves. Using dependent function types we write

$$
\begin{array}{lcl}
\mathsf{ev\_case\_z} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}.\ \Pi E_3{:}\mathsf{exp} \to \mathsf{exp}.\ \Pi V{:}\mathsf{exp}. \\
& & \quad \mathsf{eval}\ E_1\ \mathsf{z} \to \mathsf{eval}\ E_2\ V \to \mathsf{eval}\ (\mathsf{case}\ E_1\ E_2\ E_3)\ V.
\end{array}
$$

Note that the right-hand side is now a closed type since Π binds the variable it quantifies. The function ev_case_z is now a function of six arguments.

Before continuing the representation, we need to extend the simply-typed framework as presented in Section 3.1 to account for the two new phenomena we have encountered: type families indexed by objects and dependent function types.

3.5 Adding Dependent Types to the Framework

We now introduce type families and dependent function types into the simply-typed fragment, although at this point not in the full generality of LF.

The first change deals with type families: it is now more complicated to check if a given type is well-formed, since types depend on objects. Moreover, we must be able to declare the type of the indices of type families. This leads to the introduction of *kinds*, which form another level in the definition of the framework calculus.

$$\begin{array}{llll} \text{Kinds} & K & ::= & A_1 \to \ldots \to A_n \to \mathsf{type} \\ \text{Types} & A & ::= & a\ M_1 \ldots M_n \mid A_1 \to A_2 \mid \Pi x{:}A_1.\ A_2 \\ \text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \\ \text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \end{array}$$

Note that the level of objects has only changed insofar as the types occurring in λ-abstractions may now be more general. Indeed, all functions which can be expressed in this version of the framework could already be expressed in the simply-typed fragment. This highlights our motivation and intuition behind this extension: we *refine* the type system so that objects that do not represent deductions will be ill-typed. We are not interested in extending the language so that, for example, more functions would be representable.

Type families can be declared via $a{:}K$ in signatures and instantiated to types as $a\ M_1 \ldots M_n$. We refer to such types as *atomic types*, to types of the form $A_1 \to A_2$ as *simple function types*, and to types of the form $\Pi x{:}A_1.\ A_2$ as *dependent function types*. We also need to extend the inference rules for valid types and objects. We now have the basic judgments

$$\begin{array}{ll} \Gamma \vdash_\Sigma A : \mathsf{type} & A \text{ is a valid type} \\ \Gamma \vdash_\Sigma M : A & M \text{ is a valid object of type } A \end{array}$$

and auxiliary judgments

$$\begin{array}{ll} \vdash \Sigma\ \mathit{Sig} & \Sigma \text{ is a valid signature} \\ \vdash_\Sigma \Gamma\ \mathit{Ctx} & \Gamma \text{ is a valid context} \\ \Gamma \vdash_\Sigma K : \mathsf{kind} & K \text{ is a valid kind} \\ \Gamma \vdash_\Sigma M \equiv N : A & M \text{ is definitionally equal to } N \text{ at type } A \\ \Gamma \vdash_\Sigma A \equiv B : \mathsf{type} & A \text{ is definitionally equal to } B \end{array}$$

The judgments are now mutually dependent to a large degree. For example, in order to check that a type is valid, we have to check that the objects occuring in the indices of a type family are valid. The need for the convertibility judgments will be motivated below. Again, there are a variety of possibilities for defining these judgments. The one we give below is perhaps not the most convenient for the meta-theory of LF, but it reflects the process of type-checking fairly directly. We begin with the rules defining the valid types.

$$\frac{\Sigma(a) = A_1 \to \cdots \to A_n \to \mathsf{type} \qquad \Gamma \vdash_\Sigma M_1 : A_1 \quad \ldots \quad \Gamma \vdash_\Sigma M_n : A_n}{\Gamma \vdash_\Sigma a\ M_1 \ldots M_n : \mathsf{type}}\ \mathsf{atom}$$

$$\frac{\Gamma \vdash_\Sigma A : \mathsf{type} \qquad \Gamma \vdash_\Sigma B : \mathsf{type}}{\Gamma \vdash_\Sigma A \to B : \mathsf{type}}\ \mathsf{arrow}$$

$$\frac{\Gamma \vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma B : \mathsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.\ B : \mathsf{type}}\ \mathsf{pi}$$

The basic rules for valid objects are as before, except that we now have to allow for dependency. The typing rule for applying a function with a dependent type requires some thought. Recall, from the previous section,

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

The Π construct was introduced to express the dependency between the first argument and the type of the fifth argument. This means, for example, that we would expect

$$\begin{array}{ll} \vdash_{EV} & \mathsf{ev\_case\_z}\ \mathsf{z}\ \mathsf{z}\ (\lambda x{:}\mathsf{exp}.\ x)\ \mathsf{z} \\ & \quad : \mathsf{eval}\ \mathsf{z}\ \mathsf{z} \to \mathsf{eval}\ \mathsf{z}\ \mathsf{z} \to \mathsf{eval}\ (\mathsf{case}\ \mathsf{z}\ \mathsf{z}\ (\lambda x{:}\mathsf{exp}.\ x))\ \mathsf{z} \end{array}$$

to be derivable. We have instantiated E_1 with z, E_2 with z, E_3 with $(\lambda x{:}\mathsf{exp}.\ x)$ and V with z. Thus the typing rule

$$\frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.\ B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : [N/x]B}\ \mathsf{app}$$

emerges. In this rule we can see that the *type* (and not just the value) of an application of a function M to an argument N may *depend* on N. This is the reason why $\Pi x{:}A.\ B$ is called a *dependent function type*. For different reasons it is also sometimes referred to as the *dependent product*. The rule for λ-abstraction and the other rules do not change significantly.

$$\frac{\Sigma(c) = A}{\Gamma \vdash_\Sigma c : A}\,\mathsf{con} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash_\Sigma x : A}\,\mathsf{var}$$

$$\frac{\Gamma \vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M : \Pi x{:}A.\ B}\,\mathsf{lam}$$

The prior rules for functions of simple type are still valid, with the restriction that x may not occur free in B in the rule lam''. This restriction is necessary, since it is now possible for x to occur in B because objects (including variables) can appear inside types.

$$\frac{\Gamma \vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M : A \to B}\,\mathsf{lam}''$$

$$\frac{\Gamma \vdash_\Sigma M : A \to B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : B}\,\mathsf{app}''$$

The type system as given so far has a certain redundancy and is also no longer syntax-directed. That is, there are two rules for λ-abstraction (lam and lam'') and application. It is convenient to eliminate this redundancy by allowing $A \to B$ as a notation for $\Pi x{:}A.\ B$ whenever x does not occur in B. It is easy to see that under this convention, the rules lam'' and app'' are valid rules of inference, but are no longer necessary since any of their instances are also instances of lam and app.

The rules for valid signatures, contexts, and kinds are straightforward and left as Exercise 3.10. They are a special case of the rules for full LF given in Section 3.8.

One rule which is still missing is the rule of type conversion. Type conversion introduces a major complication into the type system and is difficult to motivate and illustrate with the example as we have developed it so far. We take a brief excursion and introduce another example to illustrate the necessity for the type conversion rule. Consider a potential application of dependent types in functional programming, where we would like to index the type of vectors of integers by the length of the vector. That is, vector is a type family, indexed by integers.

$$\begin{array}{lcl} \mathsf{int} & : & \mathsf{type} \\ \mathsf{plus} & : & \mathsf{int} \to \mathsf{int} \to \mathsf{int} \\ \mathsf{vector} & : & \mathsf{int} \to \mathsf{type} \end{array}$$

Furthermore, assume we can assign the following type to the function which concatenates two vectors:

$$\mathsf{concat} \quad : \quad \Pi n{:}\mathsf{int}.\ \Pi m{:}\mathsf{int}.\ \mathsf{vector}\ n \to \mathsf{vector}\ m \to \mathsf{vector}\ (\mathsf{plus}\ n\ m).$$

Then we would obtain the typings

$$\begin{array}{rcl} \mathsf{concat}\ 3\ 2\ \langle 1,2,3\rangle\ \langle 4,5\rangle & : & \mathsf{vector}\ (\mathsf{plus}\ 3\ 2) \\ \langle 1,2,3,4,5\rangle & : & \mathsf{vector}\ 5. \end{array}$$

But since the first expression presumably evaluates to the second, we would expect $\langle 1,2,3,4,5\rangle$ to have type $\mathsf{vector}\ (\mathsf{plus}\ 3\ 2)$, or the first expression to have type $\mathsf{vector}\ 5$—otherwise the language would not preserve types under evaluation.

This example illustrates two points. The first is that adding dependent types to functional languages almost invariably leads to an undecidable type-checking problem, since with the approach above one could easily encode arbitrary arithmetic equations. The second is that we need to allow conversion between equivalent types. In the example above, $\mathsf{vector}\ (\mathsf{plus}\ 3\ 2) \equiv \mathsf{vector}\ 5$. Thus we need a notion of definitional equality and add the rule of *type conversion* to the system we have considered so far.

$$\frac{\Gamma \vdash_\Sigma M : A \qquad \Gamma \vdash_\Sigma A \equiv B : \mathsf{type}}{\Gamma \vdash_\Sigma M : B}\,\mathsf{conv}$$

It is necessary to check the validity of B in the premise, since we have followed the standard technique of formulating definitional equality as an untyped judgment, and a valid type may be convertible to an invalid type. As hinted earlier, the notion of definitional equality that is most useful for our purposes is based on β- and η-conversion. We postpone the full definition until the need for these conversions is better motivated from the example.

3.6 Representing Evaluations

We summarize the signature for evaluations as we have developed it so far, taking advantage of type families and dependent types.

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

The representation function on derivations using these rules is defined inductively on the structure of the derivation.

$$\ulcorner \frac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z} \urcorner \quad = \quad \mathsf{ev\_z}$$

$$\left\ulcorner \dfrac{\begin{matrix}\mathcal{D}\\ e \hookrightarrow v\end{matrix}}{\mathbf{s}\ e \hookrightarrow \mathbf{s}\ v}\,\textsf{ev\_s} \right\urcorner \quad = \quad \textsf{ev\_s}\ \ulcorner e \urcorner\ \ulcorner v \urcorner\ \ulcorner \mathcal{D} \urcorner$$

$$\left\ulcorner \dfrac{\begin{matrix}\mathcal{D}_1\\ e_1 \hookrightarrow \mathbf{z}\end{matrix} \qquad \begin{matrix}\mathcal{D}_2\\ e_2 \hookrightarrow v\end{matrix}}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \hookrightarrow v}\,\textsf{ev\_case\_z} \right\urcorner$$

$$= \quad \textsf{ev\_case\_z}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner\ (\lambda x{:}\textsf{exp}.\ \ulcorner e_3 \urcorner)\ \ulcorner v \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner$$

The rules dealing with pairs are straightforward and introduce no new representation techniques. We leave them as Exercise 3.4. Next we consider the rule for evaluating a Mini-ML expression formed with **lam**. For this rule we will examine more closely why, for example, E_3 in the ev_case_z rule was assumed to be of type $\textsf{exp} \rightarrow \textsf{exp}$.

$$\dfrac{}{\mathbf{lam}\ x.\ e \hookrightarrow \mathbf{lam}\ x.\ e}\,\textsf{ev\_lam}$$

Recall that the representation function employs the idea of higher-order abstract syntax:

$$\ulcorner \mathbf{lam}\ x.\ e \urcorner = \textsf{lam}\ (\lambda x{:}\textsf{exp}.\ \ulcorner e \urcorner).$$

An *incorrect* attempt at a direct representation of the inference rule above would be

$$\textsf{ev\_lam} \quad : \quad \Pi E{:}\textsf{exp}.\ \textsf{eval}\ (\textsf{lam}\ (\lambda x{:}\textsf{exp}.\ E))\ (\textsf{lam}\ (\lambda x{:}\textsf{exp}.\ E)).$$

The problem with this formulation is that, because of the variable naming hygiene of the framework, we cannot instantiate E with an object that contains x free. That is, for example,

$$\dfrac{}{\mathbf{lam}\ x.\ x \hookrightarrow \mathbf{lam}\ x.\ x}\,\textsf{ev\_lam}$$

could not be represented by ($\textsf{ev\_lam}\ x$) since its type would be

$$\begin{array}{rl} & [x/E]\textsf{eval}\ (\textsf{lam}\ (\lambda x{:}\textsf{exp}.\ E))\ (\textsf{lam}\ (\lambda x{:}\textsf{exp}.\ E)) \\ = & \textsf{eval}\ (\textsf{lam}\ (\lambda x'{:}\textsf{exp}.\ x))\ (\textsf{lam}\ (\lambda x'{:}\textsf{exp}.\ x)) \\ \neq & \textsf{eval}\ (\textsf{lam}\ (\lambda x{:}\textsf{exp}.\ x))\ (\textsf{lam}\ (\lambda x{:}\textsf{exp}.\ x)) \\ = & \textsf{eval}\ \ulcorner \mathbf{lam}\ x.\ x \urcorner\ \ulcorner \mathbf{lam}\ x.\ x \urcorner \end{array}$$

for some new variable x'. Instead, we have to bundle the scope of the bound variable with its binder into a function from exp to exp, the type of the argument to lam.

$$\mathsf{ev\_lam} \quad : \quad \Pi E{:}\mathsf{exp} \to \mathsf{exp}.\ \mathsf{eval}\ (\mathsf{lam}\ E)\ (\mathsf{lam}\ E).$$

Now the evaluation of the identity function above would be correctly represented by $(\mathsf{ev\_lam}\ (\lambda x{:}\mathsf{exp}.\ x))$ which has type

$$\begin{array}{rl} & [(\lambda x{:}\mathsf{exp}.\ x)/E]\mathsf{eval}\ (\mathsf{lam}\ E)\ (\mathsf{lam}\ E) \\ = & \mathsf{eval}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ x))\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ x)). \end{array}$$

To summarize this case, we have

$$\ulcorner \frac{}{\mathbf{lam}\ x.\ e \hookrightarrow \mathbf{lam}\ x.\ e}\,\mathsf{ev\_lam} \urcorner = \mathsf{ev\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e \urcorner).$$

Yet another new technique is introduced in the representation of the rule which deals with applying a function formed by **lam** to an argument.

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

As in the previous example, e_1' must be represented with its binder as a function from exp to exp. But how do we represent $[v_2/x]e_1'$? Compositionality (Lemma 3.5) tell us that

$$\ulcorner [v_2/x]e_1' \urcorner = [\ulcorner v_2 \urcorner/x]\ulcorner e_1' \urcorner.$$

The right-hand side is β-convertible to $(\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)\ \ulcorner v_2 \urcorner$. Note that the function part of this application, $(\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)$ will be an argument to the constant representing the rule, and we can thus directly apply it to the argument representing v_2. These considerations lead to the declaration

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

where

$$\ulcorner \frac{\begin{array}{ccc} \mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1' & e_2 \hookrightarrow v_2 & [v_2/x]e_1' \hookrightarrow v \end{array}}{e_1\ e_2 \hookrightarrow v}\,\mathsf{ev\_app} \urcorner$$

$$= \quad \mathsf{ev\_app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)\ \ulcorner v_2 \urcorner\ \ulcorner v \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner\ \ulcorner \mathcal{D}_3 \urcorner.$$

Consider the evaluation of the Mini-ML expression $(\mathbf{lam}\ x.\ x)\ \mathbf{z}$:

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

Note that the third premise is a deduction of $[\mathbf{z}/x]x \hookrightarrow \mathbf{z}$ which is $\mathbf{z} \hookrightarrow \mathbf{z}$. The whole deduction is represented by the LF object

$$\begin{array}{l} \textsf{ev\_app}\ (\textsf{lam}\ (\lambda x{:}\textsf{exp}.\ x))\ \textsf{z}\ (\lambda x{:}\textsf{exp}.\ x)\ \textsf{z}\ \textsf{z} \\ \quad (\textsf{ev\_lam}\ (\lambda x{:}\textsf{exp}.\ x)) \\ \quad \textsf{ev\_z} \\ \quad \textsf{ev\_z}. \end{array}$$

But why is this well-typed? The crucial question arises with the last argument to ev_app. By substitution into the type of ev_app we find that the last argument is required to have type ($\textsf{eval}\ ((\lambda x{:}\textsf{exp}.\ x)\ \textsf{z})\ \textsf{z}$), while the actual argument, ev_z, has type $\textsf{eval}\ \textsf{z}\ \textsf{z}$. The rule of type conversion allows us to move from one type to the other provided they are definitionally equal. Thus our notion of definitional equality must include β-conversion in order to allow the representation technique whereby object-level substitution is represented by meta-level β-reduction.

In the seminal paper on LF [HHP93], definitional equality was based only on β-reduction, due to technical problems in proving the decidability of the system including η-conversion. The disadvantage of the system with only β-reduction is that not every object is convertible to a canonical form using only β-conversion (see the counterexample on page 44). This property holds once η-conversion is added. The decidability of the system with both $\beta\eta$-conversion has since been proven using four different techniques [Sal90, Coq91, Geu92, HP00].

The remaining rules of the operational semantics of Mini-ML follow the pattern of the previous rules.

$$\dfrac{e_1 \hookrightarrow \mathbf{s}\ v_1' \qquad [v_1'/x]e_3 \hookrightarrow v}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \hookrightarrow v}\,\textsf{ev\_case\_s}$$

$$\begin{array}{lcl} \textsf{ev\_case\_s} & : & \Pi E_1{:}\textsf{exp}.\ \Pi E_2{:}\textsf{exp}.\ \Pi E_3{:}\textsf{exp} \to \textsf{exp}.\ \Pi V_1'{:}\textsf{exp}.\ \Pi V{:}\textsf{exp}. \\ & & \quad \textsf{eval}\ E_1\ (\textsf{s}\ V_1') \to \textsf{eval}\ (E_3\ V_1')\ V \to \textsf{eval}\ (\textsf{case}\ E_1\ E_2\ E_3)\ V \end{array}$$

$$\dfrac{e_1 \hookrightarrow v_1 \qquad [v_1/x]e_2 \hookrightarrow v}{\mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 \hookrightarrow v}\,\textsf{ev\_letv}$$

$$\begin{array}{lcl} \textsf{ev\_letv} & : & \Pi E_1{:}\textsf{exp}.\ \Pi E_2{:}\textsf{exp} \to \textsf{exp}.\ \Pi V_1{:}\textsf{exp}.\ \Pi V{:}\textsf{exp}. \\ & & \quad \textsf{eval}\ E_1\ V_1 \to \textsf{eval}\ (E_2\ V_1)\ V \to \textsf{eval}\ (\textsf{letv}\ E_1\ E_2)\ V \end{array}$$

$$\frac{[e_1/x]e_2 \hookrightarrow v}{\textbf{let name } x = e_1 \textbf{ in } e_2} \textsf{ev\_letn}$$

$$\begin{array}{lcl} \textsf{ev\_letn} & : & \Pi E_1{:}\textsf{exp}.\ \Pi E_2{:}\textsf{exp} \to \textsf{exp}.\ \Pi V{:}\textsf{exp}. \\ & & \quad \textsf{eval}\ (E_2\ E_1)\ V \to \textsf{eval}\ (\textsf{letn}\ E_1\ E_2)\ V \end{array}$$

For the fixpoint construct, we have to substitute a compound expression and not just a variable.

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

$$\begin{array}{lcl} \textsf{ev\_fix} & : & \Pi E{:}\textsf{exp} \to \textsf{exp}.\ \Pi V{:}\textsf{exp}. \\ & & \quad \textsf{eval}\ (E\ (\textsf{fix}\ E))\ V \to \textsf{eval}\ (\textsf{fix}\ E)\ V \end{array}$$

Again we are taking advantage of compositionality in the form

$$\ulcorner [\textbf{fix } x.\ e/x]e \urcorner = [\ulcorner \textbf{fix } x.\ e \urcorner / x] \ulcorner e \urcorner \equiv (\lambda x{:}\textsf{exp}.\ \ulcorner e \urcorner)\ \ulcorner \textbf{fix } x.\ e \urcorner.$$

The succession of representation theorems follows the pattern of Section 3.3. Note that we postulate that e and v be closed, that is, do not contain any free variables. We state this explicitly, because according to the earlier inference rules, there is no requirement that **lam** $x.\ e$ be closed in the ev_lam rule. However, we would like to restrict attention to closed expressions e, since they are the only ones which will be well-typed in the empty context within the Mini-ML typing discipline. The generalization of the canonical form judgment to LF in the presence of dependent types is given in Section 3.9.

Lemma 3.7 *Let e and v be closed Mini-ML expressions, and $\mathcal{D}$ a derivation of $e \hookrightarrow v$. Then*

$$\vdash_{EV} \ulcorner \mathcal{D} \urcorner \Uparrow \textsf{eval}\ \ulcorner e \urcorner\ \ulcorner v \urcorner.$$

Proof: The proof proceeds by induction on the structure of $\mathcal{D}$. We show only one case—the others are similar and simpler.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ e_1 \hookrightarrow \textbf{lam } x.\ e_1'\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ e_2 \hookrightarrow v_2\end{array} \qquad \begin{array}{c}\mathcal{D}_3 \\ [v_2/x]e_1' \hookrightarrow v\end{array}}{e_1\ e_2 \hookrightarrow v} \textsf{ev\_app}$. Then

$$\ulcorner \mathcal{D} \urcorner = \textsf{ev\_app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner\ (\lambda x{:}\textsf{exp}.\ \ulcorner e_1' \urcorner)\ \ulcorner v_2 \urcorner\ \ulcorner v \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner\ \ulcorner \mathcal{D}_3 \urcorner$$

By the adequacy of the representation of expressions (Theorem 3.6), $\ulcorner e_1 \urcorner$, $\ulcorner e_2 \urcorner$, $\ulcorner v_2 \urcorner$, and $\ulcorner v \urcorner$ are canonical of type exp. Furthermore, $\ulcorner e_1' \urcorner$ is canonical of type exp and one application of the carrow rule yields

$$\frac{x{:}\mathsf{exp} \vdash_{EV} \ulcorner e_1' \urcorner \Uparrow \mathsf{exp}}{\vdash_{EV} \lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner \Uparrow \mathsf{exp} \to \mathsf{exp}}\ \mathsf{carrow},$$

that is, $\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner$ is canonical of type $\mathsf{exp} \to \mathsf{exp}$.

By the induction hypothesis on $\mathcal{D}_1$, we have

$$\vdash_{EV} \mathcal{D}_1 \Uparrow \mathsf{eval}\ \ulcorner e_1 \urcorner\ \ulcorner \mathbf{lam}\ x.\ e_1' \urcorner$$

and hence by the definition of the representation function

$$\vdash_{EV} \mathcal{D}_1 \Uparrow \mathsf{eval}\ \ulcorner e_1 \urcorner\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner))$$

Furthermore, by induction hypothesis on $\mathcal{D}_2$,

$$\vdash_{EV} \mathcal{D}_2 \Uparrow \mathsf{eval}\ \ulcorner e_2 \urcorner\ \ulcorner v_2 \urcorner.$$

Recalling the declaration of $\mathsf{ev\_app}$,

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

we conclude that

$$\begin{array}{l} \mathsf{ev\_app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)\ \ulcorner v_2 \urcorner\ \ulcorner v \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner \\ \ : \mathsf{eval}\ ((\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)\ \ulcorner v_2 \urcorner)\ \ulcorner v \urcorner \to \mathsf{eval}\ (\mathsf{app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner)\ \ulcorner v \urcorner. \end{array}$$

The type here is not in canonical form, since $(\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)$ is applied to $\ulcorner v_2 \urcorner$. With the rule of type conversion we now obtain

$$\begin{array}{l} \mathsf{ev\_app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)\ \ulcorner v_2 \urcorner\ \ulcorner v \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner \\ \ : \mathsf{eval}\ ([\ulcorner v_2 \urcorner/x]\ulcorner e_1' \urcorner)\ \ulcorner v \urcorner \to \mathsf{eval}\ (\mathsf{app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner)\ \ulcorner v \urcorner. \end{array}$$

where $[\ulcorner v_2 \urcorner/x]\ulcorner e_1' \urcorner$ is a valid object of type exp. The application of the object above to $\ulcorner \mathcal{D}_3 \urcorner$ (which yields $\ulcorner \mathcal{D} \urcorner$) can be seen as type-correct, since the induction hypothesis on $\mathcal{D}_3$ yields

$$\vdash_{EV} \mathcal{D}_3 \Uparrow \mathsf{eval}\ \ulcorner [v_2/x]e_1' \urcorner\ \ulcorner v \urcorner,$$

and from compositionality (Lemma 3.5) we know that

$$\ulcorner [v_2/x]e_1' \urcorner = [\ulcorner v_2 \urcorner / x]\ulcorner e_1' \urcorner.$$

Furthermore, $\mathcal{D}$ is canonical, since it is atomic and all the arguments to $\mathsf{ev\_app}$ are in canonical form.

□

Lemma 3.8 *For any LF objects E, V, and M such that $\vdash_{EV} E \Uparrow \mathsf{exp}$, $\vdash_{EV} V \Uparrow \mathsf{exp}$ and $\vdash_{EV} M \Uparrow \mathsf{eval}\ E\ V$, there exist unique Mini-ML expressions e and v and a deduction $\mathcal{D} :: e \hookrightarrow v$ such that $\ulcorner e \urcorner = E$, $\ulcorner v \urcorner = V$ and $\ulcorner \mathcal{D} \urcorner = M$.*

Proof: The proof is by structural induction on the derivation of $\vdash_{EV} M \Uparrow \mathsf{eval}\ E\ V$ (see Exercise 3.12). □

A compositionality property does not arise here in the same way as it arose for expressions since evaluations are closed. However, as we know from the use of Lemma 2.4 in the proof of type preservation (Theorem 2.5), a substitution lemma for Mini-ML typing derivations plays an important role. We will return to this in Section **??**. As before, we summarize the correctness of the representation into an adequacy theorem. It follows directly from Lemmas 3.7 and 3.8.

Theorem 3.9 (Adequacy) *There is a bijection between deductions of $e \hookrightarrow v$ for closed Mini-ML expressions e and v and canonical LF objects M such that*

$$\vdash_{EV} M \Uparrow \mathsf{eval}\ \ulcorner e \urcorner\ \ulcorner v \urcorner$$

As a second example for the representation of deductions we consider the judgment e *Value*, defined in Section 2.4. Again, the judgment is represented as a type family, value, indexed by the representation of the expression e. That is,

$$\mathsf{value} \quad : \quad \mathsf{exp} \to \mathsf{type}$$

Objects of type $\mathsf{value}\ \ulcorner e \urcorner$ then represent deductions, and inference rules are encoded as constructors for objects of such types.

$$\begin{array}{lcl} \mathsf{val\_z} & : & \mathsf{value\ z} \\ \mathsf{val\_s} & : & \Pi E{:}\mathsf{exp}.\ \mathsf{value}\ E \to \mathsf{value}\ (\mathsf{s}\ E) \\ \mathsf{val\_pair} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}.\ \mathsf{value}\ E_1 \to \mathsf{value}\ E_2 \to \mathsf{value}\ (\mathsf{pair}\ E_1\ E_2) \\ \mathsf{val\_lam} & : & \Pi E{:}\mathsf{exp} \to \mathsf{exp}.\ \mathsf{value}\ (\mathsf{lam}\ E) \end{array}$$

In the last rule, the scope of the binder **lam** is represented as a function from expressions to expressions. We refer to the signature above (including the signature E representing Mini-ML expressions) as V. We omit the obvious definition of the representation function on value deductions. The adequacy theorem only refers to its existence implicitly.

Theorem 3.10 (Adequacy) *For closed expressions e there is a bijection between deductions* $\mathcal{P} :: e$ *Value and canonical LF objects* M *such that* $\vdash_V M \Uparrow \mathsf{value}\ \ulcorner e \urcorner$ *is derivable.*

Proof: See Exercise 3.13. □

3.7 Meta-Theory via Higher-Level Judgments

So far we have completed two of the tasks we set out to accomplish in this chapter: the representation of abstract syntax and the representation of deductive systems in a logical framework. This corresponds to the specification of a language and its semantics. The third task now before us is the representation of the meta-theory of the language, that is, proofs of properties of the language and its semantics.

This representation of meta-theory should naturally fit within the framework we have laid out so far. It should furthermore reflect the structure of the informal proof as directly as possible. We are thus looking for a formal language and methodology for expressing a given proof, and not for a system or environment for finding such a proof. Once such a methodology has been developed it can also be helpful in proof search, but we would like to emphasize that this is a secondary consideration. In order to design a proof representation we must take stock of the proof techniques we have seen so far. By far the most pervasive is *structural induction*. Structural induction is applied in various forms: we have used induction over the structure of expressions, and induction over the structure of deductions. Within proofs of the latter kind we have also frequent cause to appeal to *inversion*, that is, from the form of a derivable judgment we make statements about which inference rule must have been applied to infer it. Of course, as is typical in mathematics, we break down a proof into a succession of lemmas leading up to a main theorem. A kind of lemma which arises frequently when dealing with deductive systems is a *substitution lemma*.

We first consider the issue of structural induction and its representation in the framework. At first glance, this seems to require support for logical reasoning, that is, we need quantifiers and logical connectives to express a meta-theorem, and logical axioms and inference rules to prove it. Our framework does not support this directly—we would either have to extend it very significantly or encode the logic we are attempting to model just like any other deductive system. Both of these approaches have some problems. The first does not mesh well with the idea of higher-order abstract syntax, basically because the types (such as the type exp of Mini-ML expressions) are not inductively defined in the usual sense. The problem arises from the negative occurrences of exp in the type of case, lam, let, and fix. Similar problems arise when encoding deductive systems employing parametric and hypothetical judgments such as the Mini-ML typing judgment. The second approach, that is, to first define a logical system and then reason within it, incurs a tremendous overhead in additional machinery to be developed. Furthermore, the connection between the direct representations given in the previous sections of this chapter and this indirect method is problematic.

Thus we are looking for a more direct way to exploit the expressive power of the framework we have developed so far. We will use Theorem 2.1 (value soundness for Mini-ML) and its proof as a motivating example. Recall that the theorem states that whenever $e \hookrightarrow v$ is derivable, then v *Value* is also derivable. The proof proceeds by an induction on the structure of the derivation of $e \hookrightarrow v$.

A first useful observation is that the proof is *constructive* in the sense that it implicitly contains a method for constructing a deduction $\mathcal{P}$ of the judgment v *Value*, given a deduction $\mathcal{D}$ of $e \hookrightarrow v$. This is an example of the relationship between constructive proofs and programs considered further in Sections **??** through **??**. Could we exploit the converse, that is, in what sense might the function f for constructing $\mathcal{P}$ from $\mathcal{D}$ represent a proof of the theorem? Such a function f, if it were expressible in the framework, would presumably have type $\Pi E{:}\mathsf{exp}.\ \Pi V{:}\mathsf{exp}.\ \mathsf{eval}\ E\ V \to \mathsf{value}\ V$. If it were guaranteed that a total function of this type existed, our meta-theorem would be verified. Unfortunately, such a function is not realizable within the logical framework, since it would have to be defined by a form of recursion on an object of type $\mathsf{eval}\ E\ V$. Attempting to extend the framework in a straightforward way to encompass such function definitions invalidates our approach to abstract syntax and hypothetical judgments.

But we have one further possibility: why not represent the connection between $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{P} :: v$ *Value* as a *judgment* (defined by inference rules) rather than a function? This technique is well-known from logic programming, where predicates (defined via Horn clauses) rather than functions give rise to computation. A related operational interpretation for LF signatures (which properly generalize sets of Horn clauses) forms the basis for the Elf programming language discussed in Chapter **??**. To restate the idea: we represent the essence of the proof of value soundness as a judgment relating deductions $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{P} :: v$ *Value*. Judgments relating deductions are not uncommon in the meta-theory of logic. An important example is the judgment that a natural deduction reduces to another natural deduction, which we will discuss in Section **??**.

In order to illustrate this approach, we quote various cases in the proof of value soundness and try to extract the inference rules for the judgment we motivated above. We write the judgment as

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

and read it as "$\mathcal{D}$ *reduces to* $\mathcal{P}$." Following this analysis, we give its representation in LF. Recall that the proof is by induction over the structure of the deduction

$\mathcal{D} :: e \hookrightarrow v$.

Case: $\mathcal{D} = \dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z}$. Then $v = \mathbf{z}$ is a value by the rule $\mathsf{val\_z}$.

This gives rise to the axiom

$$\dfrac{}{\dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z} \quad\Longrightarrow\quad \dfrac{}{\mathbf{z}\ \mathit{Value}}\,\mathsf{val\_z}}\,\mathsf{vs\_z}$$

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow v_1\end{array}}{\mathbf{s}\ e_1 \hookrightarrow \mathbf{s}\ v_1}\,\mathsf{ev\_s}.$$

The induction hypothesis on $\mathcal{D}_1$ yields a deduction of v_1 *Value*. Using the inference rule $\mathsf{val\_s}$ we conclude that $\mathbf{s}\ v_1$ *Value*.

This case in the proof is represented by the following inference rule.

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow v_1\end{array} \quad\Longrightarrow\quad \begin{array}{c}\mathcal{P}_1\\ v_1\ \mathit{Value}\end{array}}{\dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow v_1\end{array}}{\mathbf{s}\ e_1 \hookrightarrow \mathbf{s}\ v_1}\,\mathsf{ev\_s} \quad\Longrightarrow\quad \dfrac{\begin{array}{c}\mathcal{P}_1\\ v_1\ \mathit{Value}\end{array}}{\mathbf{s}\ v_1\ \mathit{Value}}\,\mathsf{val\_s}}\,\mathsf{vs\_s}$$

Here, the appeal to the induction hypothesis on $\mathcal{D}_1$ has been represented in the premise, where we have to establish that $\mathcal{D}_1$ reduces to $\mathcal{P}_1$.

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow \mathbf{z}\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ e_2 \hookrightarrow v\end{array}}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \hookrightarrow v}\,\mathsf{ev\_case\_z}.$$

Then the induction hypothesis applied to $\mathcal{D}_2$ yields a deduction of v *Value*, which is what we needed to show in this case.

In this case, the appeal to the induction hypothesis immediately yields the correct deduction; no further inference is necessary.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_2\\ e_2 \hookrightarrow v\end{array} \quad \begin{array}{c}\mathcal{D}_2\\ e_2 \hookrightarrow v\end{array} \Longrightarrow \begin{array}{c}\mathcal{P}_2\\ v\ \mathit{Value}\end{array}}{}}{\dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow \mathbf{z}\end{array} \quad \begin{array}{c}\mathcal{D}_2\\ e_2 \hookrightarrow v\end{array}}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \hookrightarrow v}\,\mathsf{ev\_case\_z} \quad \Longrightarrow \quad \begin{array}{c}\mathcal{P}_2\\ v\ \mathit{Value}\end{array}}\,\mathsf{vs\_case\_z}$$

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow \mathbf{s}\ v_1'\end{array} \qquad \begin{array}{c}\mathcal{D}_3\\ [v_1'/x]e_3 \hookrightarrow v\end{array}}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \hookrightarrow v}\,\mathsf{ev\_case\_s}.$$

Then the induction hypothesis applied to $\mathcal{D}_3$ yields a deduction of $v\ \mathit{Value}$, which is what we needed to show in this case.

This is like the previous case.

$$\dfrac{\begin{array}{c}\mathcal{D}_3\\ [v_1'/x]e_3 \hookrightarrow v\end{array} \Longrightarrow \begin{array}{c}\mathcal{P}_3\\ v\ \mathit{Value}\end{array}}{\dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow \mathbf{s}\ v_1'\end{array} \quad \begin{array}{c}\mathcal{D}_3\\ [v_1'/x]e_3 \hookrightarrow v\end{array}}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \hookrightarrow v}\,\mathsf{ev\_case\_s} \quad \Longrightarrow \quad \begin{array}{c}\mathcal{P}_3\\ v\ \mathit{Value}\end{array}}\,\mathsf{vs\_case\_s}$$

If $\mathcal{D}$ ends in ev_pair we reason similar to cases above.

Case:

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

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.

In this case we also have to deal with an application of *inversion* in the informal proof, analyzing the possible inference rules in the last step of the derivation $\mathcal{P}' :: \langle v_1, v_2 \rangle$ *Value*. The only possibility is val_pair. In the representation of this case as an inference rule for the reduction judgment, we require that the right-hand side of the premise end in this inference rule.

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

The remaining cases are similar to the ones shown above and left as an exercise (see Exercise 3.8). While our representation technique should be clear from the example, it also appears to be extremely unwieldy. The explicit definition of the reduction judgment given above is fortunately only a crutch in order to explain the LF signature which follows below. In practice we do not make this intermediate form explicit, but directly express the proof of a meta-theorem as an LF signature. Such signatures may seem very cumbersome, but the type reconstruction phase of the Elf implementation allows very concise signature specifications that are internally expanded into the form shown below.

The representation techniques given so far suggest that we represent the judgment

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

as a type family indexed by the representation of the deductions $\mathcal{D}$ and $\mathcal{P}$, that is,

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

Once again we need to resolve the status of the free variables E and V in order to achieve (in general) a decidable type reconstruction problem. Before, we used the dependent function type constructor Π to turn them into explicit arguments to object level constants. Here, we need to index the type family vs explicitly by E and V, both of type exp. Thus we need to extend the language for kinds (which classify type families) to admit dependencies and allow the declaration

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

The necessary generalization of the system from Section 3.5 is given in Section 3.8. The main change is a refinement of the language for kinds by admitting dependencies, quite analogous to the previous refinement of the language of types when we generalized the simply-typed fragment of Section 3.1.

We now consider the representation of some of the rules of the judgment $\mathcal{D} \Longrightarrow \mathcal{P}$ as LF objects. The axiom

$$\dfrac{}{\dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z} \quad \Longrightarrow \quad \dfrac{}{\mathbf{z}\ \mathit{Value}}\,\mathsf{val\_z}}\,\mathsf{vs\_z}$$

is represented as

$$\mathsf{vs\_z} \quad : \quad \mathsf{vs\ z\ z\ ev\_z\ val\_z}.$$

The instantiation of the type family vs is valid, since ev_z : eval z z and val_z : value z.

The second rule we considered arose from the case where the evaluation ended in the rule for successor.

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow v_1\end{array} \Longrightarrow \begin{array}{c}\mathcal{P}_1\\ v_1\ \mathit{Value}\end{array}}{\dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow v_1\end{array}}{\mathbf{s}\ e_1 \hookrightarrow \mathbf{s}\ v_1}\,\mathsf{ev\_s} \quad \Longrightarrow \quad \dfrac{\begin{array}{c}\mathcal{P}_1\\ v_1\ \mathit{Value}\end{array}}{\mathbf{s}\ v_1\ \mathit{Value}}\,\mathsf{val\_s}}\,\mathsf{vs\_s}$$

Recall the declarations for ev_s and val_s.

$$\begin{array}{lcl} \mathsf{ev\_s} & : & \Pi E{:}\mathsf{exp}.\ \Pi V{:}\mathsf{exp}.\ \mathsf{eval}\ E\ V \to \mathsf{eval}\ (\mathsf{s}\ E)\ (\mathsf{s}\ V) \\ \mathsf{val\_s} & : & \Pi E{:}\mathsf{exp}.\ \mathsf{value}\ E \to \mathsf{value}\ (\mathsf{s}\ E) \end{array}$$

The declaration corresponding to vs_s:

$$\begin{array}{lcl} \mathsf{vs\_s} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi V_1{:}\mathsf{exp}. \\ & & \Pi D_1{:}\mathsf{eval}\ E_1\ V_1.\ \Pi P_1{:}\mathsf{value}\ V_1. \\ & & \quad \mathsf{vs}\ E_1\ V_1\ D_1\ P_1 \to \mathsf{vs}\ (\mathsf{s}\ E_1)\ (\mathsf{s}\ V_1)\ (\mathsf{ev\_s}\ E_1\ V_1\ D_1)\ (\mathsf{val\_s}\ V_1\ P_1). \end{array}$$

We consider one final example, where inversion was employed in the informal proof.

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

We recall the types for the inference rule encodings involved here:

$$\begin{array}{lcl} \mathsf{val\_pair} & : & \Pi E{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}.\ \mathsf{value}\ E_1 \to \mathsf{value}\ E_2 \to \mathsf{value}\ (\mathsf{pair}\ E_1\ E_2) \\ \mathsf{ev\_fst} & : & \Pi E{:}\mathsf{exp}.\ \Pi V_1{:}\mathsf{exp}.\ \Pi V_2{:}\mathsf{exp}. \\ & & \quad \mathsf{eval}\ E\ (\mathsf{pair}\ V_1\ V_2) \to \mathsf{eval}\ (\mathsf{fst}\ E)\ V_1 \end{array}$$

The rule above can then be represented as

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

What have we achieved with this representation of the proof of value soundness in LF? The first observation is the obvious one, namely a representation theorem relating this signature to the judgment $\mathcal{D} \Longrightarrow \mathcal{P}$. Let P be the signature containing the declaration for expressions, evaluations, value deductions, and the declarations above encoding the reduction judgment via the type family vs.

Theorem 3.11 (Adequacy) *For closed expressions e and v, there is a compositional bijection between deductions of*

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

and canonical LF objects M such that

$$\vdash_P M \Uparrow \mathsf{vs}\ \ulcorner e \urcorner\ \ulcorner v \urcorner\ \ulcorner \mathcal{D} \urcorner\ \ulcorner \mathcal{P} \urcorner$$

is derivable.

This representation theorem is somewhat unsatisfactory, since the connection between the informal proof of value soundness and the LF signature remains unstated and unproven. It is difficult to make this relationship precise, since the informal proof is not given as a mathematical object. But we can claim and prove a stronger version of the value soundness theorem in which this connection is more explicit.

Theorem 3.12 (Explicit Value Soundness) *For any two expressions e and v and deduction $\mathcal{D} :: e \hookrightarrow v$ there exists a deduction $\mathcal{P} :: v\ \mathit{Value}$ such that*

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

is derivable.

Proof: By a straightfoward induction on the structure of $\mathcal{D} :: e \hookrightarrow v$ (see Exercise 3.14). □

Coupled with the proofs of the various representation theorems for expressions and deductions this establishes a formal connection between value soundness and the vs type family. Yet the essence of the relationship between the informal proof and its representation in LF lies in the connection between to the reduction judgment, and this remains implicit. To appreciate this problem, consider the judgment

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

which is defined via a single axiom

$$\overline{\begin{array}{c} \mathcal{D} \\ e \hookrightarrow v \end{array} \stackrel{\mathit{triv}}{\Longrightarrow} \begin{array}{c} \mathcal{P} \\ v\ \mathit{Value} \end{array}}\ \mathsf{vs\_triv}.$$

By value soundness and the uniqueness of the deduction of $v\ \mathit{Value}$ for a given v, $\mathcal{D} \Longrightarrow \mathcal{P}$ is derivable iff $\mathcal{D} \stackrel{\mathit{triv}}{\Longrightarrow} \mathcal{P}$ is derivable, but one would hardly claim that $\mathcal{D} \stackrel{\mathit{triv}}{\Longrightarrow} \mathcal{P}$ represents some informal proof of value soundness.

Ideally, we would like to establish some decidable, formal notion similar to the validity of LF objects which would let us check that the type family vs indeed represents *some* proof of value soundness. Such a notion can be given in the form of *schema-checking* which guarantees that a type family such as vs inductively defines a total function from its first three arguments to its fourth argument. A discussion of schema-checking [RP96, Sch00] is beyond the scope of these notes. Some material may also be found in the documentation which accompanies the implementation of Elf in the Twelf system [PS99].[1]

[1] [*update on final revision*]

3.8 The Full LF Type Theory

The levels of kinds and types in the system from Section 3.5 were given as

$$\begin{array}{llcl} \text{Kinds} & K & ::= & \mathsf{type} \mid A_1 \to \cdots \to A_n \to K \\ \text{Types} & A & ::= & a\, M_1 \ldots M_n \mid A_1 \to A_2 \mid \Pi x{:}A_1.\, A_2 \end{array}$$

We now make two changes: the first is a generalization in that we allow dependent kinds $\Pi x{:}A.\ K$. The kind of the form $A \to K$ is then a special case of the new construct where x does not occur in K. The second change is to eliminate the multiple argument instantiation of type families. This means we generalize to a level of families, among which we distinguish the types as families of kind "type."

$$\begin{array}{llcl} \text{Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}A.\ K \\ \text{Families} & A & ::= & a \mid A\ M \mid \Pi x{:}A_1.\ A_2 \\ \text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \\ \\ \text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \end{array}$$

This system differs only minimally from the one given by Harper, Honsell, and Plotkin in [HHP93]. They also allow families to be formed by explicit abstraction, that is, $\lambda x{:}A_1.\ A_2$ is a legal family. These do not occur in normal forms and we have thus chosen to omit them from our system. As mentioned previously, it also differs in that we allow β and η-conversion between objects as the basis for our notion of definitional equality, while in [HHP93] only β-conversion is considered. The judgments take a slightly different form than in Section 3.5, in that we now need to introduce a judgment to explicitly classify families.

$\Gamma \vdash_\Sigma A : K$	A is a valid family of kind K
$\Gamma \vdash_\Sigma M : A$	M is a valid object of type A
$\Gamma \vdash_\Sigma K : \mathsf{kind}$	K is a valid kind
$\vdash \Sigma\ \mathit{Sig}$	Σ is a valid signature
$\vdash_\Sigma \Gamma\ \mathit{Ctx}$	Γ is a valid context
$\Gamma \vdash_\Sigma M \equiv N : A$	M is definitionally equal to N at type A
$\Gamma \vdash_\Sigma A \equiv B : K$	A is definitionally equal to B at kind K
$\Gamma \vdash_\Sigma K \equiv K' : \mathsf{kind}$	kind K is definitionally equal to K'

These judgments are defined via the following inference rules.

$$\frac{\Sigma(a) = K}{\Gamma \vdash_\Sigma a : K}\,\textsf{famcon}$$

$$\frac{\Gamma \vdash_\Sigma A : \Pi x{:}B.\ K \qquad \Gamma \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma A\ M : [M/x]K}\,\textsf{famapp}$$

$$\frac{\Gamma \vdash_\Sigma A : \textsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma B : \textsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.\ B : \textsf{type}}\,\textsf{fampi}$$

$$\frac{\Sigma(c) = A}{\Gamma \vdash_\Sigma c : A}\,\textsf{objcon} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash_\Sigma x : A}\,\textsf{objvar}$$

$$\frac{\Gamma \vdash_\Sigma A : \textsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M : \Pi x{:}A.\ B}\,\textsf{objlam}$$

$$\frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.\ B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : [N/x]B}\,\textsf{objapp}$$

$$\frac{\Gamma \vdash_\Sigma M : A \qquad \Gamma \vdash_\Sigma A \equiv B : \textsf{type}}{\Gamma \vdash_\Sigma M : B}\,\textsf{typcnv}$$

$$\frac{}{\Gamma \vdash_\Sigma \textsf{type} : \textsf{kind}}\,\textsf{kndtyp}$$

$$\frac{\Gamma \vdash_\Sigma A : \textsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma K : \textsf{kind}}{\Gamma \vdash_\Sigma \Pi x{:}A.\ K : \textsf{kind}}\,\textsf{kndpi}$$

$$\frac{\Gamma \vdash_\Sigma A : K \qquad \Gamma \vdash_\Sigma K \equiv K' : \textsf{kind}}{\Gamma \vdash_\Sigma A : K'}\,\textsf{kndcnv}$$

$$\frac{}{\vdash_\Sigma \cdot\ Ctx}\,\textsf{ctxemp} \qquad \frac{\vdash_\Sigma \Gamma\ Ctx \qquad \Gamma \vdash_\Sigma A : \textsf{type}}{\vdash_\Sigma \Gamma, x{:}A\ Ctx}\,\textsf{ctxobj}$$

$$\frac{}{\vdash \cdot \ Sig}\,\mathsf{sigemp}$$

$$\frac{\vdash \Sigma \ Sig \qquad \vdash_\Sigma K : \mathsf{kind}}{\vdash \Sigma, a : K \ Sig}\,\mathsf{sigfam}$$

$$\frac{\vdash \Sigma \ Sig \qquad \vdash_\Sigma A : \mathsf{type}}{\vdash \Sigma, c : A \ Sig}\,\mathsf{sigobj}$$

For definitional equality, we have several classes of rules. The first rules introduces β-conversion.

$$\frac{\Gamma \vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M : B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma (\lambda x{:}A.\ M)\ N \equiv [N/x]M : [N/x]B}\,\mathsf{beta}$$

We verify the validity of the objects and types involved in order to guarantee that $\Gamma \vdash_\Sigma M \equiv N : A$ implies $\Gamma \vdash_\Sigma M : A$ and $\Gamma \vdash_\Sigma N : A$. The second rule is extensionality: two objects of function type are equal, if they are equal on an arbitrary argument x.

$$\frac{\Gamma \vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M\ x \equiv N\ x : B}{\Gamma \vdash_\Sigma M \equiv N : \Pi x{:}A.\ B}\,\mathsf{ext}$$

This rule is equivalent to η-conversion

$$\frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.\ B}{\Gamma \vdash_\Sigma (\lambda x{:}A.\ M\ x) \equiv M : \Pi x{:}A.\ B}\,\mathsf{eta}^*.$$

where η is restricted to the case the x is not free in M. The second class of rules specifies that $\equiv$ is an *equivalence*, satisfying reflexivity, symmetry, and transitivity at each level. We only show the rules for objects; the others are obvious analogues.

$$\frac{\Gamma \vdash_\Sigma M : A}{\Gamma \vdash_\Sigma M \equiv M : A}\,\mathsf{objrefl} \qquad \frac{\Gamma \vdash_\Sigma N \equiv M : A}{\Gamma \vdash_\Sigma M \equiv N : A}\,\mathsf{objsym}$$

$$\frac{\Gamma \vdash_\Sigma M \equiv O : A \qquad \Gamma \vdash_\Sigma O \equiv N : A}{\Gamma \vdash_\Sigma M \equiv N : A}\,\mathsf{objtrans}$$

Finally we require rules to ensure that $\equiv$ is a *congruence*, that is, conversion can be applied to subterms. Technically, we use the notion of a *simultaneous congruence* that allows simultaneous conversion in all subterms of a given term. We only show the congruence rules at the levels of objects.

$$\frac{\Sigma(c) = A}{\Gamma \vdash_\Sigma c = c : A}\,\textsf{cngobjcon} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash_\Sigma x = x : A}\,\textsf{cngobjvar}$$

$$\frac{\Gamma \vdash_\Sigma M_1 \equiv N_1 : \Pi x{:}A_2.\ A_1 \qquad \Gamma \vdash_\Sigma M_2 \equiv N_2 : A_2}{\Gamma \vdash_\Sigma M_1\ M_2 \equiv N_1\ N_2 : [M_2/x]A_1}\,\textsf{cngobjapp}$$

$$\frac{\Gamma \vdash_\Sigma A' \equiv A : \textsf{type} \qquad \Gamma \vdash_\Sigma A'' \equiv A : \textsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M \equiv N : B}{\Gamma \vdash_\Sigma \lambda x{:}A'.\ M \equiv \lambda x{:}A''.\ N : \Pi x{:}A.\ B}\,\textsf{cngobjlam}$$

In addition we also need type and kind conversion rules for the same reason they are needed in the typing judgments (see Exercise 3.15). Some important properties of the LF type theory are stated at the end of next section.

3.9 Canonical Forms in LF

The notion of a canonical form, which is central to the representation theorems for LF encodings, is somewhat more complicated in full LF than in the simply typed fragment given in Section 3.1. In particular, we need to introduce auxiliary judgments for canonical types. At the same time we replace the rules with an indeterminate number of premises by using another auxiliary judgment which establishes that an object is *atomic*, that is, of the form $x\ M_1 \ldots M_n$ or $c\ M_1 \ldots M_n$, and its arguments $M_1, \ldots, M_n$ are again canonical. An analogous judgment exists at the level of families. Thus we arrive at the judgments

$\Gamma \vdash_\Sigma M \Uparrow A$	M is canonical of type A
$\Gamma \vdash_\Sigma A \Uparrow \textsf{type}$	A is a canonical type
$\Gamma \vdash_\Sigma M \downarrow A$	M is atomic of type A
$\Gamma \vdash_\Sigma A \downarrow K$	A is atomic of kind K

These are defined by the following inference rules.

$$\frac{\Gamma \vdash_\Sigma A \Uparrow \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M \Uparrow B}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M \Uparrow \Pi x{:}A.\ B}\ \mathsf{canpi}$$

$$\frac{\Gamma \vdash_\Sigma A \downarrow \mathsf{type} \qquad \Gamma \vdash_\Sigma M \downarrow A}{\Gamma \vdash_\Sigma M \Uparrow A}\ \mathsf{canatm}$$

$$\frac{\Gamma \vdash_\Sigma M \Uparrow A \qquad \Gamma \vdash_\Sigma A \equiv B : \mathsf{type}}{\Gamma \vdash_\Sigma M \Uparrow B}\ \mathsf{cancnv}$$

$$\frac{\Sigma(c) = A}{\Gamma \vdash_\Sigma c \downarrow A}\ \mathsf{atmcon} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash_\Sigma x \downarrow A}\ \mathsf{atmvar}$$

$$\frac{\Gamma \vdash_\Sigma M \downarrow \Pi x{:}A.\ B \qquad \Gamma \vdash_\Sigma N \Uparrow A}{\Gamma \vdash_\Sigma M\ N \downarrow [N/x]B}\ \mathsf{atmapp}$$

$$\frac{\Gamma \vdash_\Sigma M \downarrow A \qquad \Gamma \vdash_\Sigma A \equiv B : \mathsf{type}}{\Gamma \vdash_\Sigma M \downarrow B}\ \mathsf{atmcnv}$$

The conversion rules are included here for the same reason they are included among the inference rules for valid types and terms.

$$\frac{\Sigma(a) = K}{\Gamma \vdash_\Sigma a \downarrow K}\ \mathsf{attcon}$$

$$\frac{\Gamma \vdash_\Sigma A \downarrow \Pi x{:}B.\ K \qquad \Gamma \vdash_\Sigma M \Uparrow B}{\Gamma \vdash_\Sigma A\ M \downarrow [M/x]K}\ \mathsf{attapp}$$

$$\frac{\Gamma \vdash_\Sigma A \downarrow K \qquad \Gamma \vdash_\Sigma K \equiv K' : \mathsf{kind}}{\Gamma \vdash_\Sigma A \downarrow K'}\ \mathsf{attcnv}$$

$$\frac{\Gamma \vdash_\Sigma A \Uparrow \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma B \Uparrow \mathsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.\ B \Uparrow \mathsf{type}}\ \mathsf{cntpi}$$

$$\frac{\Gamma \vdash_\Sigma A \downarrow \mathsf{type}}{\Gamma \vdash_\Sigma A \Uparrow \mathsf{type}}\ \mathsf{cntatm}$$

We state, but do not prove a few critical properties of the LF type theory. Basic versions of the results are due to Harper, Honsell, and Plotkin [HHP93], but their seminal paper does not treat extensionality or η-conversion. The theorem below is a consequence of results in [HP00]. The proofs are quite intricate, because of the mutually dependent nature of the levels of objects and types and are beyond the scope of these notes.

Theorem 3.13 (Properties of LF) *Assume Σ is a valid signature, and Γ a context valid in Σ. Then the following hold.*

1. *If $\Gamma \vdash_\Sigma M \Uparrow A$ then $\Gamma \vdash_\Sigma M : A$.*
2. *If $\Gamma \vdash_\Sigma A \Uparrow \mathsf{type}$ then $\Gamma \vdash_\Sigma A : \mathsf{type}$.*
3. *For each object M such that $\Gamma \vdash_\Sigma M : A$ there exists a unique object M' such that $\Gamma \vdash_\Sigma M \equiv M' : A$ and $\Gamma \vdash_\Sigma M' \Uparrow A$. Moreover, M' can be effectively computed.*
4. *For each type A such that $\Gamma \vdash_\Sigma A : \mathsf{type}$ there exists a unique type A' such that $\Gamma \vdash_\Sigma A \equiv A' : \mathsf{type}$ and $\Gamma \vdash_\Sigma A' \Uparrow \mathsf{type}$. Moreover, A' can be effectively computed.*
5. *Type checking in the LF type theory is decidable.*

3.10 Summary and Further Discussion

In this chapter we have developed a methodology for representing deductive systems and their meta-theory within the LF Logical Framework. The LF type theory is a refinement of the Church's simply-typed λ-calculus with dependent types.

The cornerstone of the methodology is a technique for representing the expressions of a language, whereby object-language variables are represented by meta-language variables. This leads to the notion of higher-order abstract syntax, since now syntactic operators that bind variables must be represented by corresponding binding operators in the meta-language. As a consequence, expressions that differ only in the names of bound variables in the object language are α-convertible in the meta-language. Furthermore, substitution can be modelled by β-reduction. These relationships are expressed in the form of an adequacy theorem for the representation which postulates the existence of a compositional bijection between object language expressions and meta-language objects of a given type. Ordinarily, the representation of abstract syntax of a language does not involve dependent, but only simple types. This means that the type of representations of expressions, which was exp in the example used throughout this chapter, is a type constant and not an indexed type family. We refer to such a constant as a family at level 0. We summarize the methodology in the following table.

Object Language	Meta-Language
Syntactic Category Expressions	Level 0 Type Family $\mathsf{exp} : \mathsf{type}$
Variable x	Variable x
Constructor $\langle e_1, e_2 \rangle$	Constant $\mathsf{pair}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner$, where $\mathsf{pair} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$
Binding Constructor $\mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2$	Second-Order Constant $\mathsf{letv}\ \ulcorner e_1 \urcorner\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_2 \urcorner)$, where $\mathsf{letv} : \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$

An alternative approach, which we do not pursue here, is to use terms in a first-order logic to represent Mini-ML expressions. For example, we may have a binary function constant *pair* and a ternary function constant *letv*. We then define a predicate *exp* which is true for expressions and false otherwise. This predicate is defined via a set of axioms. For example, $\forall e_1.\ \forall e_2.\ \mathit{exp}(e_1) \wedge \mathit{exp}(e_1) \supset \mathit{exp}(\mathit{pair}(e_1, e_2))$. Similarly, $\forall x.\ \forall e_1.\ \forall e_2.\ \mathit{var}(x) \wedge \mathit{exp}(e_1) \wedge \mathit{exp}(e_2) \supset \mathit{exp}(\mathit{letv}(x, e_1, e_2))$, where *var* is another predicate which is true on variables and false otherwise. Since first-order logic is undecidable, we must then impose some restriction on the possible definitions of predicates such as *exp* or *var* in order to guarantee decidable representations. Under appropriate restrictions such predicates can then be seen to define types. A commonly used class are *regular tree types*. Membership of a term in such a type can be decided by a finite tree automaton [GS84]. This approach to representation and types is the one usually taken in logic programming which has its roots in first-order logic. For a collection of papers describing this and related approaches see [Pfe92]. The principal disadvantage of regular tree types in a first-order term language is that it does not admit representation techniques such as higher-order abstract syntax. Its main advantage is that it naturally permits subtypes. For example, we could easily define the set of Mini-ML values as a subtype of expressions, while the representation of values in LF requires an explicit judgment. Thus, we do not capture in LF that it is decidable if an expression is a value. Some initial work towards combining regular tree types and function types is reported in [FP91] and [Pfe93].

The second representation technique translates judgments to types and deductions to objects. This is often summarized by the motto *judgments-as-types*. This can be seen as a methodology for formalizing the semantics of a language, since semantic judgments (such as evaluation or typing judgments) can be given conveniently and elegantly as deductive systems. The goal is now to reduce checking of deductions to type-checking within the framework (which is decidable). For this reduction to work correctly, the simply-typed framework which is sufficient for abstract syntax in most cases, needs to be refined by type families and dependent function types. The index objects for type families typically are representations of expressions, which means that they are typed at level 0. We refer to a family which is indexed by objects typed at level 0 as a level 1 family. We can summarize this representation technique in the following table.

Object Language	Meta-Language
Semantic Judgment $e \hookrightarrow v$	Level 1 Type Family $\mathsf{eval} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{type}$
Inference Rule $\dfrac{e \hookrightarrow v}{\mathbf{s}\, e \hookrightarrow \mathbf{s}\, v}\,\mathsf{ev\_s}$	Constant Declaration $\mathsf{ev\_s}$: $\Pi E{:}\mathsf{exp}.\ \Pi V{:}\mathsf{exp}.$ $\mathsf{eval}\ E\ V$ $\to \mathsf{eval}\ (\mathsf{s}\ E)\ (\mathsf{s}\ V)$
Deduction	Well-Typed Object
Deductive System	Signature

An alternative to dependent types (which we do not pursue here) is to define predicates in a higher-order logic which are true of valid deductions and false otherwise. The type family eval, indexed by two expressions, then becomes a simple type *eval* and we additionally require a predicate *valid*. The logics of higher-order Horn clauses [NM98] and hereditary Harrop formulas [MNPS91] support this approach and the use of higher-order abstract syntax. They have been implemented in the logic programming language λProlog [NM99] and the theorem prover Isabelle [Pau94]. The principal disadvantage of this approach is that checking the validity of a deduction is reduced to theorem proving in the meta-logic. Thus decidability is not guaranteed by the representation and we do not know of any work to isolate decidable classes of higher-order predicates which would be analogous to regular tree types. Hereditary Harrop formulas have a natural logic programming interpretation, which permits them to be used as the basis for implementing programs related to judgments specified via deductive systems. For example, programs for evaluation or type inference in Mini-ML can be easily and elegantly expressed in λProlog. In Chapter **??** we show that a similar operational interpretation is also possible for the LF type theory, leading to the language Elf.

The third question we considered was how to represent the proofs of properties of deductive systems. The central idea was to formulate the functions implicit in a constructive proof as a judgment relating deductions. For example, the proof that evaluation returns a value proceeds by induction over the structure of the deduction $\mathcal{D} :: e \hookrightarrow v$. This gives rise to a total function f, mapping each $\mathcal{D} :: e \hookrightarrow v$ into a deduction $\mathcal{P} :: v\ \mathit{Value}$. We then represent this function as a judgment $\mathcal{D} \Longrightarrow \mathcal{P}$ such that $\mathcal{D} \Longrightarrow \mathcal{P}$ is derivable if and only if $f(\mathcal{D}) = \mathcal{P}$. A strong adequacy theorem, however, is not available, since the mathematical proof is informal, and not itself introduced as a mathematical object. The judgment between deductions is then again represented in LF using the idea of judgments-as-types, although now the index objects to the representing family represent deductions. We refer to a family indexed by objects whose type is constructed from a level 1 family as a level 2 family. The technique for representing proofs of theorems about deductive systems which have been formalized in the previous step is summarized in the following table.

Object Language	Meta-Language
Informal Proof Value Soundness	Level 2 Type Family $\mathsf{vs} : \Pi E{:}\mathsf{exp}.\ \Pi V{:}\mathsf{exp}.$ $\mathsf{eval}\ E\ V \to \mathsf{value}\ V \to \mathsf{type}$
Case in Structural Induction	Constant Declaration
Base Case for Axioms	Constant of Atomic Type
Induction Step	Constant of Functional Type

A decidable criterion on when a given type family represents a proof of a theorem about a deductive system is subject of current research [RP96, Sch00].[2]

An alternative to this approach is to work in a stronger type theory with explicit induction principles in which we can directly express induction arguments. This approach is taken, for example, in the Calculus of Inductive Constructions [PM93] which has been implemented in the Coq system [DFH$^+$93]. The disadvantage of this approach is that it does not coexist well with the techniques of higher-order abstract syntax and judgments-as-types, since the resulting representation types (for example, exp) are not inductively defined in the usual sense.

3.11 Exercises

Exercise 3.1 Consider a variant of the typing rules given in Section 3.1 where the rules var, con, lam, tcon, and ectx are replaced by the following rules.

$$\frac{\vdash_\Sigma \Gamma\ \mathit{Ctx} \qquad \Sigma(c) = A}{\Gamma \vdash_\Sigma c : A}\,\mathsf{con}' \qquad \frac{\vdash_\Sigma \Gamma\ \mathit{Ctx} \qquad \Gamma(x) = A}{\Gamma \vdash_\Sigma x : A}\,\mathsf{var}'$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M : A \to B}\,\mathsf{lam}'$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \qquad \Sigma(a) = \mathsf{type}}{\vdash_\Sigma a : \mathsf{type}}\,\mathsf{tcon}' \qquad \frac{\vdash \Sigma\ \mathit{Sig}}{\vdash_\Sigma \cdot\ \mathit{Ctx}}\,\mathsf{ectx}$$

In what sense are these two systems equivalent? Formulate and carefully prove an appropriate theorem.

[2] [*update in final revision*]

Exercise 3.2 Prove Theorem 3.1.

Exercise 3.3 Prove Lemma 3.4

Exercise 3.4 Give LF representations of the natural semantics rules ev_pair, ev_fst, and ev_snd (see Section 2.3).

Exercise 3.5 Reconsider the extension of the Mini-ML language by unit and disjoint sum type (see Exercise 2.7). Give LF representation for

1. the new expression constructors,
2. the new rules in the evaluation and value judgments, and
3. the new cases in the proof of value soundness.

Exercise 3.6 Give the LF representation of the evaluations in Exercise 2.3. You may need to introduce some abbreviations in order to make it feasible to write it down.

Exercise 3.7 Complete the definition of the representation function for evaluations given in Section 3.6.

Exercise 3.8 Complete the definition of the judgment

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

given in Section 3.7 and give the LF encoding of the remaining inference rules.

Exercise 3.9 Formulate and prove a theorem which expresses that the rules lam″ and app″ in Section 3.5 are no longer necessary, if $A \to B$ stands for $\Pi x{:}A.\ B$ for some x which does not occur in B.

Exercise 3.10 State the rules for valid signatures, contexts, and kinds which were omitted in Section 3.8.

Exercise 3.11 Formulate an adequacy theorem for the representation of evaluations which is more general than Theorem 3.9 by allowing free variables in the expressions e and v.

Exercise 3.12 Show the case for ev_app in the proof of Lemma 3.8.

Exercise 3.13 Prove Theorem 3.10.

Exercise 3.14 Prove Theorem 3.12.

Exercise 3.15 Complete the rules defining the full LF type theory.

Exercise 3.16 Prove items 1 and 2 of Theorem 3.13.

Exercise 3.17 In Exercise 2.13 you were asked to write a function $observe : \mathbf{nat} \rightarrow \mathbf{nat}$ that, given a lazy value of type **nat** returns the corresponding eager value if it exists.

1. Carefully state and prove the correctness of your function *observe*.
2. Explain the meaning of your proof as a higher-level judgment (without necessarily giving all details).

Bibliography

[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.

[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.

[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.

[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.

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

[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.

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

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

[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.

[Pfe91] 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.

[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.

[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.