Computation and Deduction

Frank Pfenning
Carnegie Mellon University

$\boxed{\text{Draft of March 6, 2001}}$

Notes for a course given at Carnegie Mellon University during the Spring semester of 2001. Please send comments to `fp@cs.cmu.edu`. These notes are to be published by Cambridge University Press.

Contents

Chapter 1

Introduction

Now, the question, *What is a judgement?* is no small question, because the notion of judgement is just about the first of all the notions of logic, the one that has to be explained before all the others, before even the notions of proposition and truth, for instance.

— Per Martin-Löf

On the Meanings of the Logical Constants and the Justifications of the Logical Laws [ML96]

In everyday computing we deal with a variety of different languages. Some of them such as C, C++, Ada, ML, or Prolog are intended as general purpose languages. Others like Emacs Lisp, Tcl, TEX, HTML, csh, Perl, SQL, Visual Basic, VHDL, or Java were designed for specific domains or applications. We use these examples to illustrate that many more computer science researchers and system developers are engaged in language design and implementation than one might at first suspect. We all know examples where ignorance or disregard of sound language design principles has led to languages in which programs are much harder to write, debug, compose, or maintain than they should be. In order to understand the principles which guide the design of programming languages, we should be familiar with their theory. Only if we understand the properties of complete languages as opposed to the properties of individual programs, do we understand how the pieces of a language fit together to form a coherent (or not so coherent) whole.

As these notes demonstrate, the theory of programming languages does not require a deep and complicated mathematical apparatus, but can be carried out in a concrete, intuitive, and computational way. With only a few exceptions, the material in these notes has been fully implemented in a meta-language, a so-called logical framework. This implementation encompasses the languages we study, the algorithms pertaining to these languages (for example, compilation), and the proofs of their properties (for example, compiler correctness). This allows hands-on experimentation with the given languages and algorithms and the exploration of variants and extensions. We now briefly sketch our approach and the organization of these notes.

1.1 The Theory of Programming Languages

The theory of programming languages covers diverse aspects of languages and their implementations. Logically first are issues of *concrete syntax* and parsing. These have been relatively well understood for some time and are covered in numerous books. We therefore ignore them in these notes in order to concentrate on deeper aspects of languages.

The next question concerns the *type structure* of a language. The importance of the type structure for the design and use of a language can hardly be overemphasized. Types help to sort out meaningless programs and type checking catches many errors before a program is ever executed. Types serve as formal, machine-checked documentation for an implementation. Types also specify interfaces to modules and are therefore important to obtain and maintain consistency in large software systems.

Next we have to ask about the meanings of programs in a language. The most immediate answer is given by the *operational semantics* which specifies the behavior of programs, usually at a relatively high level of abstraction.

Thus the fundamental parts of a language specification are the *syntax*, the *type system*, and the *operational semantics*. These lead to many meta-theoretic questions regarding a particular language. Is it effectively decidable if an input expression is well-typed? Do the type system and the operational semantics fit together? Are types needed during the execution of a program? In these notes we investigate such questions in the context of small *functional* and *logic programming* languages. Many of the same issues arise for realistic languages, and many of the same solutions still apply.

The specification of an operational semantics rarely corresponds to an efficient language implementation, since it is designed primarily to be easy to reason about. Thus we also study *compilation*, the translation from a source language to a target language which can be executed more efficiently by an *abstract machine*. Of course we want to show that compilation preserves the observable behavior of programs. Another important set of questions is whether programs satisfy some abstract specification, for example, if a particular function really computes the integer logarithm. Similarly, we may ask if two programs compute the same function, even though they may implement different algorithms and thus may differ operationally. These questions lead to general *type theory* and *denotational semantics*, which we consider only superficially in these notes. We concentrate on type systems and the operational behavior of programs, since they determine programming style and are closest to the programmer's intuition about a language. They are also amenable to immediate implementation, which is not so direct, for example, for denotational semantics.

The principal novel aspect of these notes is that the operational perspective is not limited to the programming languages we study (the *object language*), but encompasses the *meta-language*, that is, the framework in which we carry out our investigation. Informally, the meta-language is centered on the notions of *judgment* and *deductive system* explained below. They have been formalized in a *logical framework* (LF) [HHP93] in which judgments can be specified at a high level of abstraction, consistent with informal practice in computer science. LF has been given an operational interpretation in the Elf meta-programming language [Pfe91a, Pfe94], thus providing means for a computational meta-theory. Implementations of the languages, algorithms, and proofs of meta-theorems in these notes are available electronically and constitute an important supplement to these notes. They provide the basis for hands-on experimentation with language variants, extensions, proofs of exercises, and projects related to the formalization and implementation of other topics in the theory of programming languages. The most recent implementation of both LF and Elf is called Twelf [PS99], available from the Twelf home page at `http://www.cs.cmu.edu/~twelf/`.

1.2 Deductive Systems

In logic, deductive systems are often introduced as a syntactic device for establishing semantic properties. We are given a *language* and a *semantics* assigning meaning to expressions in the language, in particular to a category of expressions called *formulas*. Furthermore, we have a distinguished semantic property, such as *truth* in a particular model. A *deductive system* is then defined through a set of *axioms* (all of which are true formulas) and *rules of inference* which yield true formulas when given true formulas. A *deduction* can be viewed as a tree labelled with formulas, where the axioms are leaves and inference rules are interior nodes, and the label of the root is the formula whose truth is established by the deduction. This naturally leads to a number of meta-theoretic questions concerning a deductive system. Perhaps the most immediate are *soundness*: "*Are the axioms true, and is truth preserved by the inference rules?*" and *completeness*: "*Can every true formula be deduced?*". A difficulty with this general approach is that it requires the mathematical notion of a model, which is complex and not immediately computational.

An alternative is provided by Martin-Löf [ML96, ML85] who introduces the notion of a *judgment* (such as "*A is true*") as something we may know by virtue of a *proof*. For him the notions of judgment and proof are thus more basic than the notions of proposition and truth. The meaning of propositions is explained via the rules we may use to establish their truth. In Martin-Löf's work these notions are mostly informal, intended as a philosophical foundation for constructive mathematics and computer science. Here we are concerned with actual implementation and also the meta-theory of deductive systems. Thus, when we refer to judgments we mean *formal* judgments and we substitute the synonyms *deduction* and *derivation* for formal proof. The term *proof* is reserved for proofs in the meta-theory. We call a judgment *derivable* if (and only if) it can be established by a deduction, using the given axioms and inference rules. Thus the derivable judgments are defined inductively. Alternatively we might say that the set of derivable judgments is the least set of judgments containing the axioms and closed under the rules of inference. The underlying view that axioms and inference rules provide a semantic definition for a language was also advanced by Gentzen [Gen35] and is sometimes referred to as *proof-theoretic semantics*. A study of deductive systems is then a semantic investigation with syntactic means. The investigation of a theory of deductions often gives rise to *constructive* proofs of properties such as consistency (not every formula is provable), which was one of Gentzen's primary motivations. This is also an important reason for the relevance of deductive systems in computer science.

The study of deductive systems since the pioneering work of Gentzen has arrived at various styles of calculi, each with its own concepts and methods independent of any particular logical interpretation of the formalism. Systems in the style of Hilbert [HB34] have a close connection to combinatory calculi [CF58]. They are characterized by many axioms and a small number of inference rules. Systems of *natural deduction* [Gen35, Pra65] are most relevant to these notes, since they directly define the meaning of logical symbols via inference rules. They are also closely related to typed λ-calculi and thus programming languages via the so-called Curry-Howard isomorphism [How80]. Gentzen's *sequent calculus* can be considered a calculus of proof search and is thus relevant to logic programming, where computation is realized as proof search according to a fixed strategy.

In these notes we concentrate on calculi of natural deduction, investigating methods for

1. the definition of judgments,
2. the implementation of algorithms for deriving judgments and manipulating deductions, and
3. proving properties of deductive systems.

As an example of these three tasks, we show what they might mean in the context of the description of a programming language.

Let e range over expressions of a statically typed programming language, τ range over types, and v over those expressions which are values. The relevant judgments are

$\triangleright\, e : \tau$	e has type τ
$e \hookrightarrow v$	e evaluates to v

1. The deductive systems that define these judgments fix the type system and the operational semantics of our programming language.

2. An implementation of these judgments provides a program for type inference and an interpreter for expressions in the language.

3. A typical meta-theorem is *type preservation*, which expresses that the type system and the operational semantics are compatible:

   If $\triangleright\, e : \tau$ is derivable and $e \hookrightarrow v$ is derivable, then $\triangleright\, v : \tau$ is derivable.

In this context the deductive systems *define* the judgments under considerations: there simply exists no external, semantical notion against which our inference rules should be measured. Different inference systems lead to different notions of evaluation and thus to different programming languages.

We use standard notation for judgments and deductions. Given a judgment J with derivation $\mathcal{D}$ we write

$$\begin{array}{c}\mathcal{D}\\ J\end{array}$$

or, because of its typographic simplicity, $\mathcal{D} :: J$. An application of a rule of inference with *conclusion* J and *premises* $J_1, \ldots, J_n$ has the general form

$$\frac{J_1 \quad \ldots \quad J_n}{J}\,\textit{rule name.}$$

An axiom is simply an inference rule with no premises ($n = 0$) and we still show the horizontal line. We use script letters $\mathcal{D}, \mathcal{E}, \mathcal{P}, \mathcal{Q}, \ldots$ to range over deductions. Inference rules are almost always *schematic*, that is, they contain meta-variables. A schematic inference rule stands for all its *instances* which can be obtained by replacing the meta-variables by expressions in the appropriate syntactic category. We usually drop the byword "schematic" for the sake of simplicity.

Deductive systems are intended to provide an explicit calculus of evidence for judgments. Sometimes complex side conditions restrict the set of possible instances of an inference rule. This can easily destroy the character of the inference rules in that much of the evidence for a judgment may be implicit in the side conditions. We therefore limit ourselves to side conditions regarding legal occurrences of variables in the premises. It is no accident that our formalization techniques directly account for such side conditions. Other side conditions as they may be found in the literature can often be converted into explicit premises involving auxiliary judgments. There are a few standard means to combine judgments to form new ones. In particular, we employ *parametric* and *hypothetical* judgments. Briefly, a *hypothetical judgment* expresses that a judgment J may be derived under the assumption or hypothesis J'. If we succeed in constructing a deduction $\mathcal{D}'$ of J' we can substitute $\mathcal{D}'$ in every place where J' was used in the original, hypothetical deduction of J to obtain unconditional evidence for J. A *parametric judgment* J is a judgment containing a meta-variable x ranging over some syntactic category. It is judged evident if we can provide a deduction $\mathcal{D}$ of J such that we can replace x in $\mathcal{D}$ by any expression in the appropriate syntactic category and obtain a deduction for the resulting instance of J.

In the statements of meta-theorems we generally refer to a judgment J as derivable or not derivable. This is because judgments and deductions have now become objects of study and are themselves subjects of judgments. However, using the phrase "*is derivable*" pervasively tends to be verbose, and we will take the liberty of using "J" to stand for "J is derivable" when no confusion can arise.

1.3 Goals and Approach

We pursue several goals with these notes. First of all, we would like to convey a certain style of language definition using deductive systems. This style is standard practice in modern computer science and students of the theory of programming languages should understand it thoroughly.

Secondly, we would like to impart the main techniques for proving properties of programming languages defined in this style. Meta-theory based on deductive systems requires surprisingly few principles: induction over the structure of derivations is by far the most common technique.

Thirdly, we would like the reader to understand how to employ the LF logical framework [HHP93] and the Twelf system [PS99] in order to *implement* these definitions and related algorithms. This serves several purposes. Perhaps the most important is that it allows hands-on experimentation with otherwise dry definitions and theorems. Students can get immediate feedback on their understanding of the course material and their ideas about exercises. Furthermore, using a logical framework deepens one's understanding of the methodology of deductive systems, since the framework provides an immediate, formal account of informal explanations and practice in computer science.

Finally, we would like students to develop an understanding of the subject matter, that is, functional and logic programming. This includes an understanding of various type systems, operational semantics for functional languages, high-level compilation techniques, abstract machines, constructive logic, the connection between constructive proofs and functional programs, and the view of goal-directed proof search as the foundation for logic programming. Much of this understanding, as well as the analysis and implementation techniques employed here, apply to other paradigms and more realistic, practical languages.

The notes begin with the theory of Mini-ML, a small functional language including recursion and polymorphism (Chapter 2). We informally discuss the language specification and its meta-theory culminating in a proof of type preservation, always employing deductive systems. This exercise allows us to identify common concepts of deductive systems which drive the design of a *logical framework*. In Chapter 3 we then incrementally introduce features of the logical framework LF, which is our formal meta-language. Next we show how LF is implemented in the Elf programming language (Chapter 4). Elf endows LF with an operational interpretation in the style of logic programming, thus providing a programming language for meta-programs such as interpreters or type inference procedures. Our meta-theory will always be constructive and we observe that meta-theoretic proofs can also be implemented and executed in Elf, although at present they cannot be verified completely. Next we introduce the important concepts of parametric and hypothetical judgments (Chapter 5) and develop the implementation of the proof of type preservation. At this point the basic techniques have been established, and we devote the remaining chapters to case studies: compilation and compiler correctness (Chapter 6), natural deduction and the connection between constructive proofs and functional programs (Chapter **??**), the theory of logic programming (Chapter **??**), and advanced type systems (Chapter **??**).

Chapter 2

The Mini-ML Language

Unfortunately one often pays a price for [languages which impose no discipline of types] in the time taken to find rather inscrutable bugs—anyone who mistakenly applies CDR to an atom in LISP and finds himself absurdly adding a property list to an integer, will know the symptoms.

— Robin Milner
A Theory of Type Polymorphism in Programming [Mil78]

In preparation for the formalization of Mini-ML in a logical framework, we begin with a description of the language in a common mathematical style. The version of Mini-ML we present here lies in between the language introduced in [CDDK86, Kah87] and call-by-value PCF [Plo75, Plo77]. The description consists of three parts: (1) the abstract syntax, (2) the operational semantics, and (3) the type system. Logically, the type system would come before the operational semantics, but we postpone the more difficult typing rules until Section 2.5.

2.1 Abstract Syntax

The language of types centrally affects the kinds of expression constructs that should be available in the language. The types we include in our formulation of Mini-ML are natural numbers, products, and function types. Many phenomena in the theory of Mini-ML can be explored with these types; some others are the subject of Exercises 2.7, 2.8, and 2.10. For our purposes it is convenient to ignore certain questions of concrete syntax and parsing and present the abstract syntax of the language in Backus Naur Form (BNF). The vertical bar "|" separates alternatives on the right-hand side of the definition symbol "::=". Definitions in this style can be understood as inductive definitions of syntactic categories such as *types* or *expressions*.

$$\text{Types} \quad \tau \quad ::= \quad \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \alpha$$

Here, **nat** stands for the type of natural numbers, $\tau_1 \times \tau_2$ is the type of pairs with elements from τ_1 and τ_2, $\tau_1 \to \tau_2$ is the type of functions mapping elements of type τ_1 elements of type τ_2. Type variables are denoted by α. Even though our language supports a form of polymorphism, we do not explicitly include a polymorphic type constructor in the language; see Section 2.5 for further discussion of this issue. We follow the convention that $\times$ and $\to$ associate to the right, and that $\times$ has higher precendence than $\to$. Parentheses may be used to explicitly group type expressions. For example,

$$\mathbf{nat} \times \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$$

denotes the same type as

$$(\mathbf{nat} \times \mathbf{nat}) \to (\mathbf{nat} \to \mathbf{nat}).$$

For each concrete type (excluding type variables) we have expressions that allow us to construct elements of that type and expressions that allow us to destruct elements of that type. We choose to separate the languages of types and expressions so we can define the operational semantics without recourse to typing. We have in mind, however, that only well-typed programs will ever be executed.

$$\begin{array}{lcll}
\text{Expressions} \quad e & ::= & \mathbf{z} \mid \mathbf{s}\ e \mid (\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \ \mid \ \mathbf{s}\ x \Rightarrow e_3) & \textit{Natural numbers} \\
 & & \mid \langle e_1, e_2 \rangle \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e & \textit{Pairs} \\
 & & \mid \mathbf{lam}\ x.\ e \mid e_1\ e_2 & \textit{Functions} \\
 & & \mid \mathbf{let\,val}\ x = e_1\ \mathbf{in}\ e_2 & \textit{Definitions} \\
 & & \mid \mathbf{let\,name}\ x = e_1\ \mathbf{in}\ e_2 & \\
 & & \mid \mathbf{fix}\ x.\ e & \textit{Recursion} \\
 & & \mid x & \textit{Variables}
\end{array}$$

Most of these constructs should be familiar from functional programming languages such as ML: **z** stands for zero, **s** e stands for the successor of e. A **case**-expression chooses a branch based on whether the value of the first argument is zero or non-zero. Abstraction, **lam** $x.\ e$, forms functional expressions. It is often written $\lambda x.\ e$, but we will reserve "λ" for the formal meta-language. Application of a function to an argument is denoted simply by juxtaposition.

Definitions introduced by **let val** provide for explicit sequencing of computation, while **let name** introduces a local name abbreviating an expression. The latter incorporates a form of polymorphism. Recursion is introduced via the fixed point construct **fix** $x.\ e$ explained below using the example of addition.

We use e, e', ..., possibly subscripted, to range over expressions. The letters x, y, and occasionally u, v, f and g, range over variables. We use a boldface font for language keywords. Parentheses are used for explicit grouping as for types. Juxtaposition associates to the left. The period (in **lam** x. and **fix** x.) and the keywords **in** and **of** act as a prefix whose scope extends as far to the right as possible while remaining consistent with the present parentheses. For example, **lam** $x.\ x\ \mathbf{z}$ stands for **lam** $x.\ (x\ \mathbf{z})$ and

$$\mathbf{let\,val}\ x = \mathbf{z}\ \mathbf{in}\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow y \mid \mathbf{s}\ x' \Rightarrow f\ x'\ x$$

denotes the same expression as

$$\mathbf{let\,val}\ x = \mathbf{z}\ \mathbf{in}\ (\mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow y \mid \mathbf{s}\ x' \Rightarrow ((f\ x')\ x)).$$

As a first example, consider the following implementation of the predecessor function, where the predecessor of 0 is defined to be 0.

$$\mathit{pred} = \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow x'$$

Here "=" introduces a definition in our mathematical meta-language.

As a second example, we develop the definition of addition that illustrates the fixed point operator in the language. We begin with an informal recursive specification of the behavior or plus_1.

$$\begin{array}{lcl} \mathit{plus}_1\ \mathbf{z}\ m & = & m \\ \mathit{plus}_1\ (\mathbf{s}\ n')\ m & = & \mathbf{s}\ (\mathit{plus}_1\ n'\ m) \end{array}$$

In order to express this within our language, we need to perform several transformations. The first is to replace the two clauses of the specification by one, expressing the case distinction in Mini-ML.

$$\mathit{plus}_1\ n\ m = \mathbf{case}\ n\ \mathbf{of}\ \mathbf{z} \Rightarrow m \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathit{plus}_1\ x'\ m)$$

In the second step we explicitly abstract over the arguments of plus_1.

$$\mathit{plus}_1 = \mathbf{lam}\ x.\ \mathbf{lam}\ y.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow y \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathit{plus}_1\ x'\ y)$$

At this point we have an equation of the form

$$f = e(\ldots f \ldots)$$

where f is a variable (plus_1) and $e(\ldots f \ldots)$ is an expression with some occurrences of f. If we think of e as a function that depends on f, then f is a *fixed point* of e since $e(\ldots f \ldots) = f$. The Mini-ML language allows us to construct such a fixed point directly.

$$f = \mathbf{fix}\ x.\ e(\ldots x \ldots)$$

In our example, this leads to the definition

$$plus_1 = \textbf{fix}\ add.\ \textbf{lam}\ x.\ \textbf{lam}\ y.\ \textbf{case}\ x\ \textbf{of}\ \textbf{z} \Rightarrow y \mid \textbf{s}\ x' \Rightarrow \textbf{s}\ (add\ x'\ y).$$

Our operational semantics will have to account for the recursive nature of computation in the presence of fixed point expressions, including possible non-termination.

The reader may want to convince himself now or after the detailed presentation of the operational semantics that the following are correct alternative definitions of addition.

$$\begin{array}{lcl} plus_2 & = & \textbf{lam}\ y.\ \textbf{fix}\ add.\ \textbf{lam}\ x.\ \textbf{case}\ x\ \textbf{of}\ \textbf{z} \Rightarrow y \mid \textbf{s}\ x' \Rightarrow \textbf{s}\ (add\ x') \\ plus_3 & = & \textbf{fix}\ add.\ \textbf{lam}\ x.\ \textbf{lam}\ y.\ \textbf{case}\ x\ \textbf{of}\ \textbf{z} \Rightarrow y \mid \textbf{s}\ x' \Rightarrow add\ x'\ (\textbf{s}\ y) \end{array}$$

2.2 Substitution

The concepts of *free* and *bound variable* are fundamental in this and many other languages. In Mini-ML variables are scoped as follows:

$$\begin{array}{ll} \textbf{case}\ e_1\ \textbf{of}\ \textbf{z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3 & \text{binds } x \text{ in } e_3, \\ \textbf{lam}\ x.\ e & \text{binds } x \text{ in } e, \\ \textbf{let val}\ x = e_1\ \textbf{in}\ e_2 & \text{binds } x \text{ in } e_2, \\ \textbf{let name}\ x = e_1\ \textbf{in}\ e_2 & \text{binds } x \text{ in } e_2, \\ \textbf{fix}\ x.\ e & \text{binds } x \text{ in } e. \end{array}$$

An occurrence of variable x in an expression e is a *bound occurrence* if it lies within the scope of a binder for x in e, in which case it refers to the innermost enclosing binder. Otherwise the variable is said to be *free* in e. For example, the two non-binding occurrences of x and y below are bound, while the occurrence of u is free.

$$\textbf{let name}\ x = \textbf{lam}\ y.\ y\ \textbf{in}\ x\ u$$

The names of bound variables may be important to the programmer's intuition, but they are irrelevant to the formal meaning of an expression. We therefore do not distinguish between expressions that differ only in the names of their bound variables. For example, **lam** $x.\ x$ and **lam** $y.\ y$ both denote the identity function. Of course, variables must be renamed "consistently", that is, corresponding variable occurrences must refer to the same binder. Thus

$$\textbf{lam}\ x.\ \textbf{lam}\ y.\ x = \textbf{lam}\ u.\ \textbf{lam}\ y.\ u$$

but

$$\textbf{lam}\ x.\ \textbf{lam}\ y.\ x \neq \textbf{lam}\ y.\ \textbf{lam}\ y.\ y.$$

When we wish to be explicit, we refer to expressions that differ only in the names of their bound variables as *α-convertible* and the renaming operation as *α-conversion*.

Languages in which meaning is invariant under variable renaming are said to be *lexically scoped* or *statically scoped*, since it is clear from program text, without considering the operational semantics, where a variable occurrence is bound. Languages such as Lisp that permit dynamic scoping for some variables are semantically less transparent and more difficult to describe formally and reason about.

A fundamental operation on expressions is *substitution*, the replacement of a free variable by an expression. We write $[e'/x]e$ for the result of substituting e' for all free occurrences of x in e. During this substitution operation we must make sure that no variable that is free in e' is *captured* by a binder in e. But since we may tacitly rename bound variables, the result of substitution is always uniquely defined. For example,

$$[x/y]\mathbf{lam}\ x.\ y = [x/y]\mathbf{lam}\ x'.\ y = \mathbf{lam}\ x'.\ x \neq \mathbf{lam}\ x.\ x.$$

This form of substitution is often called *capture-avoiding substitution*. It is the only meaningful form of substitution under the variable renaming convention: with pure textual replacement we could conclude that

$$\mathbf{lam}\ x.\ x = [x/y](\mathbf{lam}\ x.\ y) = [x/y](\mathbf{lam}\ x'.\ y) = \mathbf{lam}\ x'.\ x,$$

which is clearly nonsensical.

Substitution has a number of obvious and perhaps not so obvious properties. The first class of properties may be considered part of a rigorous definition of substitution. These are equalities of the form

$$\begin{array}{rcll}
[e'/x]x & = & e' & \\
[e'/x]y & = & y & \text{for } x \neq y \\
[e'/x](e_1\,e_2) & = & ([e'/x]e_1)\,([e'/x]e_2) & \\
[e'/x](\mathbf{lam}\ y.\ e) & = & \mathbf{lam}\ y.\ [e'/x]e & \text{for } x \neq y \text{ and } y \text{ not free in } e'.
\end{array}$$

Of course, there exists one of these equations for every construct in the language. A second important property states that consecutive substitutions can be permuted with each other under certain circumstances:

$$[e_2/x_2]([e_1/x_1]e) = [([e_2/x_2]e_1)/x_1]([e_2/x_2]e)$$

provided x_1 does not occur free in e_2. The reader is invited to explore the formal definition and properties of substitution in Exercise 2.9. We will take such simple properties largely for granted.

2.3 Operational Semantics

The first judgment to be defined is the evaluation judgment, $e \hookrightarrow v$ (read: e evaluates to v). Here v ranges over expressions; in Section 2.4 we define the notion of a *value* and show that the result of evaluation is in fact a value. For now we only informally think of v as representing the value of e. The definition of the evaluation judgment is given by inference rules. Here, and in the remainder of these notes, we think of axioms as inference rules with no premises, so that no explicit distinction between axioms and inference rules is necessary. A definition of a judgment via inference rules is inductive in nature, that is, e evaluates to v if and only if $e \hookrightarrow v$ can be established with the given set of inference rules. We will make use of this inductive structure of deductions throughout these notes in order to prove properties of deductive systems.

This approach to the description of the operational semantics of programming languages goes back to Plotkin [Plo75, Plo81] under the name of *structured operational semantics* and Kahn [Kah87], who calls his approach *natural semantics.* Our presentation follows the style of natural semantics.

We begin with the rules concerning the natural numbers.

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

The first rule expresses that $\mathbf{z}$ is a constant and thus evaluates to itself. The second expresses that $\mathbf{s}$ is a constructor, and that its argument must be evaluated, that is, the constructor is *eager* and not *lazy.* For more on this distinction, see Exercise 2.13. Note that the second rule is schematic in e and v: any instance of this rule is valid.

The next two inference rules concern the evaluation of the **case** construct. The second of these rules requires substitution as introduced in the previous section.

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

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

The substitution of v_1' for x in case e_1 evaluates to $\mathbf{s}\,v_1'$ eliminates the need for *environments* which are present in many other semantic definitions. These rules are *declarative* in nature, that is, we define the operational semantics by declaring rules of inference for the evaluation judgment without actually implementing an interpreter. This is exhibited clearly in the two rules for the conditional: in an interpreter, we would evaluate e_1 and then branch to the evaluation of e_2 or e_3, depending on the value of e_1. This interpreter structure is not contained in these rules; in fact, naive search for a deduction under these rules will behave differently (see Section 4.3).

As a simple example that can be expressed using only the four rules given so far, consider the derivation of $(\mathbf{case}\ \mathbf{s}\ (\mathbf{s}\ \mathbf{z})\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow x') \hookrightarrow \mathbf{s}\ \mathbf{z}$. This would arise as a subdeduction in the derivation of *pred* (**s** (**s z**)) with the earlier definition of *pred*.

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

The conclusion of the second premise arises as $[(\mathbf{s}\ \mathbf{z})/x']x' = \mathbf{s}\ \mathbf{z}$. We refer to a deduction of a judgment $e \hookrightarrow v$ as an *evaluation deduction* or simply *evaluation* of e. Thus deductions play the role of traces of computation.

Pairs do not introduce any new ideas.

$$\dfrac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{\langle e_1, e_2 \rangle \hookrightarrow \langle v_1, v_2 \rangle}\ \textsf{ev\_pair}$$

$$\dfrac{e \hookrightarrow \langle v_1, v_2 \rangle}{\mathbf{fst}\ e \hookrightarrow v_1}\ \textsf{ev\_fst} \qquad \dfrac{e \hookrightarrow \langle v_1, v_2 \rangle}{\mathbf{snd}\ e \hookrightarrow v_2}\ \textsf{ev\_snd}$$

This form of operational semantics avoids explicit error values: for some expressions e there simply does not exist any value v such that $e \hookrightarrow v$ would be derivable. For example, when trying to construct a v and a deduction of the expression $(\mathbf{case}\ \langle \mathbf{z}, \mathbf{z} \rangle\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow x') \hookrightarrow v$, one arrives at the following impasse:

$$\dfrac{\dfrac{\dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\ \textsf{ev\_z} \qquad \dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\ \textsf{ev\_z}}{\langle \mathbf{z}, \mathbf{z} \rangle \hookrightarrow \langle \mathbf{z}, \mathbf{z} \rangle}\ \textsf{ev\_pair} \qquad ?}{\mathbf{case}\ \langle \mathbf{z}, \mathbf{z} \rangle\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow x' \hookrightarrow v}\ ?$$

There is no inference rule "?" which would allow us to fill v with an expression and obtain a valid deduction. This particular kind of example will be excluded by the typing system, since the argument which determines the cases here is not a natural number. On the other hand, natural semantics does not preclude a formulation with explicit error elements (see Exercise 2.10).

In programming languages such as Mini-ML functional abstractions evaluate to themselves. This is true for languages with call-by-value and call-by-name semantics, and might be considered a distinguishing characteristic of *evaluation* compared to *normalization.*

$$\frac{}{\textbf{lam}\ x.\ e \hookrightarrow \textbf{lam}\ x.\ e}\ \textsf{ev\_lam}$$

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

This specifies a *call-by-value* discipline for our language, since we evaluate e_2 and then substitute the resulting value v_2 for x in the function body e_1'. In a call-by-name discipline, we would omit the second premise and the third premise would be $[e_2/x]e_1' \hookrightarrow v$ (see Exercise 2.13).

The inference rules above have an inherent inefficiency: the deduction of a judgment of the form $[v_2/x]e_1' \hookrightarrow v$ may have many copies of a deduction of $v_2 \hookrightarrow v_2$. In an actual interpreter, we would like to evaluate e_1' in an *environment* where x is bound to v_2 and simply look up the value of x when needed. Such a modification in the specification, however, is not straightforward, since it requires the introduction of *closures.* We make such an extension to the language as part of the compilation process in Section 6.1.

The rules for **let** are straightforward, given our understanding of function application. There are two variants, depending on whether the subject is evaluated (**let val**) or not (**let name**).

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

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

The **let val** construct is intended for the computation of intermediate results that may be needed more than once, while the **let name** construct is primarily intended to give names to functions so they can be used polymorphically. For more on this distinction, see Section 2.5.

Finally, we come to the fixed point construct. Following the considerations in the example on page 11, we arrive at the rule

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

Thus evaluation of a fixed point construct unrolls the recursion one level and evaluates the result. Typically this uncovers a **lam**-abstraction which evaluates to itself. This rule clearly exhibits another situation in which an expression does not have a value: consider $\mathbf{fix}\ x.\ x$. There is only one rule with a conclusion of the form $\mathbf{fix}\ x.\ e \hookrightarrow v$, namely ev_fix. So if $\mathbf{fix}\ x.\ x \hookrightarrow v$ were derivable for some v, then the premise, namely $[\mathbf{fix}\ x.\ x/x]x \hookrightarrow v$ would also have to be derivable. But $[\mathbf{fix}\ x.\ x/x]x = \mathbf{fix}\ x.\ x$, and the instance of ev_fix would have to have the form

$$\frac{\mathbf{fix}\ x.\ x \hookrightarrow v}{\mathbf{fix}\ x.\ x \hookrightarrow v}\,\mathsf{ev\_fix}.$$

Clearly we have made no progress, and hence there is no evaluation of $\mathbf{fix}\ x.\ x$. As an example of a successful evaluation, consider the function which doubles its argument.

$$\mathit{double} = \mathbf{fix}\ f.\ \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (f\ x'))$$

The representation of the evaluation tree for *double* (**s z**) uses a linear notation which is more amenable to typesetting. The lines are shown in the order in which they would arise during a left-to-right, depth-first construction of the evaluation deduction. Thus it might be easiest to read this from the bottom up. We use *double* as a short-hand for the expression shown above and *not* as a definition within the language in order to keep the size of the expressions below manageable. Furthermore, we use *double*$'$ for the result of unrolling the fixed point expression *double* once.

1	$\mathit{double}' \hookrightarrow \mathit{double}'$	ev_lam
2	$\mathit{double} \hookrightarrow \mathit{double}'$	ev_fix 1
3	$\mathbf{z} \hookrightarrow \mathbf{z}$	ev_z
4	$\mathbf{s}\ \mathbf{z} \hookrightarrow \mathbf{s}\ \mathbf{z}$	ev_s 3
5	$\mathbf{z} \hookrightarrow \mathbf{z}$	ev_z
6	$\mathbf{s}\ \mathbf{z} \hookrightarrow \mathbf{s}\ \mathbf{z}$	ev_s 5
7	$\mathit{double}' \hookrightarrow \mathit{double}'$	ev_lam
8	$\mathit{double} \hookrightarrow \mathit{double}'$	ev_fix 1
9	$\mathbf{z} \hookrightarrow \mathbf{z}$	ev_z
10	$\mathbf{z} \hookrightarrow \mathbf{z}$	ev_z
11	$\mathbf{z} \hookrightarrow \mathbf{z}$	ev_z
12	$(\mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (\mathit{double}\ x'))) \hookrightarrow \mathbf{z}$	ev_case_z 10, 11
13	$\mathit{double}\ \mathbf{z} \hookrightarrow \mathbf{z}$	ev_app 8, 9, 12
14	$\mathbf{s}\ (\mathit{double}\ \mathbf{z}) \hookrightarrow \mathbf{s}\ \mathbf{z}$	ev_s 13
15	$\mathbf{s}\ (\mathbf{s}\ (\mathit{double}\ \mathbf{z})) \hookrightarrow \mathbf{s}\ (\mathbf{s}\ \mathbf{z})$	ev_s 14
16	$(\mathbf{case}\ \mathbf{s}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (\mathit{double}\ x'))) \hookrightarrow \mathbf{s}\ (\mathbf{s}\ \mathbf{z})$	ev_case_s 6, 15
17	$\mathit{double}\ (\mathbf{s}\ \mathbf{z}) \hookrightarrow \mathbf{s}\ (\mathbf{s}\ \mathbf{z})$	ev_app 2, 4, 16

where

$$\begin{array}{rcl} \mathit{double} & = & \mathbf{fix}\ f.\ \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (f\ x')) \\ \mathit{double}' & = & \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (\mathit{double}\ x')) \end{array}$$

The inefficiencies of the rules we alluded to above can be seen clearly in this example: we need two copies of the evaluation of **s z**, one of which should in principle be unnecessary, since we are in a call-by-value language (see Exercise 2.12).

2.4 Evaluation Returns a Value

Before we discuss the type system, we will formulate and prove a simple meta-theorem. The set of *values* in Mini-ML can be described by the BNF grammar

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

This kind of grammar can be understood as a form of inductive definition of a subcategory of the syntactic category of expressions: a value is either **z**, the successor of a value, a pair of values, or any **lam**-expression. There are alternative equivalent definition of values, for example as those expressions which evaluate to themselves (see Exercise 2.14). Syntactic subcategories (such as values as a subcategory of expressions) can also be defined using deductive systems. The judgment in this case is unary: e *Value*. It is defined by the following inference rules:

$$\frac{}{\mathbf{z}\ \mathit{Value}}\,\textsf{val\_z} \qquad\qquad \frac{e\ \mathit{Value}}{\mathbf{s}\ e\ \mathit{Value}}\,\textsf{val\_s}$$

$$\frac{e_1\ \mathit{Value} \qquad e_2\ \mathit{Value}}{\langle e_1, e_2\rangle\ \mathit{Value}}\,\textsf{val\_pair} \qquad\qquad \frac{}{\mathbf{lam}\ x.\ e\ \mathit{Value}}\,\textsf{val\_lam}$$

Again, this definition is inductive: an expression e is a value if and only if e *Value* can be derived using these inference rules. It is common mathematical practice to use different variable names for elements of the smaller set in order to distinguish them in the presentation. But is it justified to write $e \hookrightarrow v$ with the understanding that v is a value? This is the subject of the next theorem. The proof is instructive as it uses an induction over the structure of a deduction. This is a central technique for proving properties of deductive systems and the judgments they define. The basic idea is simple: if we would like to establish a property for all deductions of a judgment we show that the property is preserved by all inference rules, that is, we assume the property holds of the deduction of the premises and we must show that the property holds of the deduction of the conclusion. For an axiom (an inference rule with no premises) this just means that we have to prove the property outright, with no assumptions. An important special case of this induction principle is an *inversion principle*: in many cases the form of a judgment uniquely determines the last rule of inference which must have been applied, and we may conclude the existence of a deduction of the premise.

Theorem 2.1 (Value Soundness) *For any two expressions e and v, if $e \hookrightarrow v$ is derivable, then v Value is derivable.*

Proof: The proof is by induction over the structure of the deduction $\mathcal{D} :: e \hookrightarrow v$. We show a number of typical cases.

Case: $\mathcal{D} = \dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z}$. Then $v = \mathbf{z}$ is a value by the rule val_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 val_s we conclude that $\mathbf{s}\, v_1$ *Value*.

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.

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 *Value*, which is what we needed to show in this case.

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

Case:

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

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

must end in an application of the val_pair rule, that is,

$$\mathcal{P}' = \cfrac{\begin{array}{cc} \mathcal{P}_1 & \mathcal{P}_2 \\ v_1 \ \mathit{Value} & v_2 \ \mathit{Value} \end{array}}{\langle v_1, v_2 \rangle \ \mathit{Value}} \ \mathsf{val\_pair}$$

for some $\mathcal{P}_1$ and $\mathcal{P}_2$. Hence v_1 *Value* must be derivable, which is what we needed to show. We call this form of argument *inversion*.

Case: If $\mathcal{D}$ ends in ev_snd we reason similar to the previous case.

Case: $\mathcal{D} = \cfrac{}{\mathbf{lam}\ x.\ e \hookrightarrow \mathbf{lam}\ x.\ e} \ \mathsf{ev\_lam}.$

Again, this case is immediate, since $v = \mathbf{lam}\ x.\ e$ is a value by rule val_lam.

Case:

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

Then the induction hypothesis on $\mathcal{D}_3$ yields that v *Value*.

Case: $\mathcal{D}$ ends in ev_letv. Similar to the previous case.

Case: $\mathcal{D}$ ends in ev_letn. Similar to the previous case.

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c} \mathcal{D}_1 \\ [\mathbf{fix}\ x.\ e/x]e \hookrightarrow v \end{array}}{\mathbf{fix}\ x.\ e \hookrightarrow v} \ \mathsf{ev\_fix}.$$

Again, the induction hypothesis on $\mathcal{D}_1$ directly yields that v is a value.

□

Since it is so pervasive, we briefly summarize the principle of *structural induction* used in the proof above. We assume we have an arbitrary derivation $\mathcal{D}$ of $e \hookrightarrow v$ and we would like to prove a property P of $\mathcal{D}$. We show this by induction on the structure of $\mathcal{D}$: For each inference rule in the system defining the judgment $e \hookrightarrow v$ we show that the property P holds for the conclusion under the assumption that it holds for every premise. In the special case of an inference rule with no premises we have no inductive assumptions; this therefore corresponds to a base case of the induction. This suffices to establish the property P for every derivation $\mathcal{D}$ since it must be constructed from the given inference rules. In our particular theorem the property P states that there exists a derivation $\mathcal{P}$ of the judgment that v is a value.

2.5 The Type System

In the presentation of the language so far we have not used types. Thus types are external to the language of expressions and a judgment such as $\triangleright\, e : \tau$ may be considered as establishing a property of the (untyped) expression e. This view of types has been associated with Curry [Cur34, CF58], and systems in this style are often called *type assignment systems*. An alternative is a system in the style of Church [Chu32, Chu33, Chu41], in which types are included within expressions, and every well-typed expression has a unique type. We will discuss such a system in Section **??**.

Mini-ML as presented by Clément *et al.* [CDDK86] is a language with some limited polymorphism in that it explicitly distinguishes between *simple types* and *type schemes* with some restrictions on the use of type schemes. This notion of polymorphism was introduced by Milner [Mil78, DM82]. We will refer to it as *schematic polymorphism*. In our formulation, we will be able to avoid using type schemes completely by distinguishing two forms of definitions via **let**, one of which is polymorphic. A formulation in this style orginates with Hannan and Miller [HM89, Han91, Han93].

$$\text{Types} \quad \tau \quad ::= \quad \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \alpha$$

Here, α stands for type variables. We also need a notion of *context* which assigns types to free variables in an expression.

$$\text{Contexts} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x{:}\tau$$

We generally omit the empty context, "$\cdot$", and, for example, write $x{:}\tau$ for $\cdot, x{:}\tau$. We also have to deal again with the problem of variable names. In order to avoid ambiguities and simplify the presentation, we stipulate that each variable may be declared at most once in a context Γ. When we wish to emphasize this assumption, we refer to contexts without repeated variables as *valid contexts*. We write $\Gamma(x)$ for the type assigned to x in Γ.

The typing judgment

$$\Gamma \triangleright e : \tau$$

states that expression e has type τ in context Γ. It is important for the meta-theory that there is exactly one inference rule for each expression constructor. We say that the definition of the typing judgment is *syntax-directed*. Of course, many deductive systems defining typing judgments are not syntax-directed (see, for example, Section **??**).

We begin with typing rules for natural numbers. We require that the two branches of a **case**-expression have the same type τ. This means that no matter which of the two branches of the **case**-expression applies during evaluation, the value of the whole expression will always have type τ.

$$\frac{}{\Gamma \triangleright \mathbf{z} : \mathbf{nat}}\,\mathsf{tp\_z} \qquad \frac{\Gamma \triangleright e : \mathbf{nat}}{\Gamma \triangleright \mathbf{s}\ e : \mathbf{nat}}\,\mathsf{tp\_s}$$

$$\frac{\Gamma \triangleright e_1 : \mathbf{nat} \qquad \Gamma \triangleright e_2 : \tau \qquad \Gamma, x{:}\mathbf{nat} \triangleright e_3 : \tau}{\Gamma \triangleright (\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) : \tau}\,\mathsf{tp\_case}$$

Implicit in the third premise of the tp_case rule is the information that x is a bound variable whose scope is e_3. Moreover, x stands for a natural number (the predecessor of the value of e_1). Note that we may have to rename the variable x in case another variable with the same name already occurs in the context Γ.

Pairing is straightforward.

$$\frac{\Gamma \triangleright e_1 : \tau_1 \qquad \Gamma \triangleright e_2 : \tau_2}{\Gamma \triangleright \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\,\mathsf{tp\_pair}$$

$$\frac{\Gamma \triangleright e : \tau_1 \times \tau_2}{\Gamma \triangleright \mathbf{fst}\ e : \tau_1}\,\mathsf{tp\_fst} \qquad \frac{\Gamma \triangleright e : \tau_1 \times \tau_2}{\Gamma \triangleright \mathbf{snd}\ e : \tau_2}\,\mathsf{tp\_snd}$$

Because of the following rule for **lam**-abstraction, the type of an expression is not unique. This is a characteristic property of a type system in the style of Curry.

$$\frac{\Gamma, x{:}\tau_1 \triangleright e : \tau_2}{\Gamma \triangleright \mathbf{lam}\ x.\ e : \tau_1 \to \tau_2}\,\mathsf{tp\_lam}$$

$$\frac{\Gamma \triangleright e_1 : \tau_2 \to \tau_1 \qquad \Gamma \triangleright e_2 : \tau_2}{\Gamma \triangleright e_1\ e_2 : \tau_1}\,\mathsf{tp\_app}$$

The rule tp_lam is (implicitly) restricted to the case where x does not already occur in Γ, since we made the general assumption that no variable occurs more than once in a context. This restriction can be satisfied by renaming the bound variable x, thus allowing the construction of a typing derivation for $\triangleright\ \mathbf{lam}\ x.\ \mathbf{lam}\ x.\ x : \alpha \to (\beta \to \beta)$, but not for $\triangleright\ \mathbf{lam}\ x.\ \mathbf{lam}\ x.\ x : \alpha \to (\beta \to \alpha)$. Note that together with this rule, we need a rule for looking up variables in the context.

$$\frac{\Gamma(x) = \tau}{\Gamma \triangleright x : \tau}\,\mathsf{tp\_var}$$

As variables occur at most once in a context, this rule does not lead to any inherent ambiguity.

Our language incorporates a **let val** expression to compute intermediate values. This is not strictly necessary, since it may be defined using **lam**-abstraction and application (see Exercise 2.20).

$$\frac{\Gamma \triangleright e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \triangleright e_2 : \tau_2}{\Gamma \triangleright \textbf{let val}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ \mathsf{tp\_letv}$$

Even though e_1 may have more than one type, only one of these types (τ_1) can be used for occurrences of x in e_2. In other words, x can *not* be used *polymorphically*, that is, at various types.

Schematic polymorphism (or *ML-style polymorphism*) only plays a role in the typing rule for **let name**. What we would like to achieve is that, for example, the following judgment holds:

$$\triangleright\ \textbf{let name}\ f = \textbf{lam}\ x.\ x\ \textbf{in}\ \langle f\ \mathbf{z}, f\ (\textbf{lam}\ y.\ \mathbf{s}\ y)\rangle : \textbf{nat} \times (\textbf{nat} \to \textbf{nat})$$

Clearly, the expression can be evaluated to $\langle \mathbf{z}, (\textbf{lam}\ y.\ \mathbf{s}\ y)\rangle$, since $\textbf{lam}\ x.\ x$ can act as the identity function on any type, that is, both

$$\begin{array}{rl} & \triangleright\ \textbf{lam}\ x.\ x : \textbf{nat} \to \textbf{nat}, \\ \text{and} & \triangleright\ \textbf{lam}\ x.\ x : (\textbf{nat} \to \textbf{nat}) \to (\textbf{nat} \to \textbf{nat}) \end{array}$$

are derivable. In a type system with explicit polymorphism a more general judgment might be expressed as $\triangleright\ \textbf{lam}\ x.\ x : \forall\alpha.\ \alpha \to \alpha$ (see Section **??**). Here, we use a different device by allowing different types to be assigned to e_1 at different occurrences of x in e_2 when type-checking $\textbf{let name}\ x = e_1\ \textbf{in}\ e_2$. We achieve this by substituting e_1 for x in e_2 and checking only that the result is well-typed.

$$\frac{\Gamma \triangleright e_1 : \tau_1 \qquad \Gamma \triangleright [e_1/x]e_2 : \tau_2}{\Gamma \triangleright \textbf{let name}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ \mathsf{tp\_letn}$$

Note that τ_1, the type assigned to e_1 in the first premise, is not used anywhere. We require such a derivation nonetheless so that all subexpressions of a well-typed term are guaranteed to be well-typed (see Exercise 2.21). The reader may want to check that with this rule the example above is indeed well-typed.

Finally we come to the typing rule for fixed point expressions. In the evaluation rule, we substitute $[\textbf{fix}\ x.\ e/x]e$ in order to evaluate $\textbf{fix}\ x.\ e$. For this to be well-typed, the body e must be well-typed under the assumption that the variable x has the type of whole fixed point expression. Thus we are lead to the rule

$$\frac{\Gamma, x{:}\tau \triangleright e : \tau}{\Gamma \triangleright \textbf{fix}\ x.\ e : \tau}\ \mathsf{tp\_fix}.$$

More general typing rules for fixed point constructs have been considered in the literature, most notably the rule of the Milner-Mycroft calculus which is discussed in Section **??**.

An important property of the system is that an expression uniquely determines the last inference rule of its typing derivation. This leads to a principle of *inversion*: from the type of an expression we can draw conclusions about the types of its constituents expressions. The inversion principle is used pervasively in the proof of Theorem 2.5, for example. In many deductive systems similar inversion principles hold, though often they turn out to be more difficult to prove.

Lemma 2.2 (Inversion) *Given a context* Γ *and an expression* e *such that* $\Gamma \triangleright e : \tau$ *is derivable for some* τ*. Then the last inference rule of any derivation of* $\Gamma \triangleright e : \tau'$ *for some* τ' *is uniquely determined.*

Proof: By inspection of the inference rules. □

Note that this does not imply that types are unique. In fact, they are not, as illustrated above in the rule for **lam**-abstraction.

2.6 Type Preservation

Before we come to the statement and proof of type preservation in Mini-ML, we need a few preparatory lemmas. The reader may wish to skip ahead and reexamine these lemmas wherever they are needed. We first note the property of weakening and then state and prove a substitution lemma for typing derivations. Substitution lemmas are basic to the investigation of many deductive systems, and we will pay special attention to them when considering the representation of proofs of meta-theorems in a logical framework. We use the notation Γ, Γ' for the result of appending the declarations in Γ and Γ' assuming implicitly that the result is valid. Recall that a context is *valid* if no variable in it is declared more than once.

Lemma 2.3 (Weakening) *If* $\Gamma_1, \Gamma_2 \triangleright e : \tau$ *then* $\Gamma_1, \Gamma', \Gamma_2 \triangleright e : \tau$ *provided* $\Gamma_1, \Gamma', \Gamma_2$ *is a valid context.*

Proof: By straightforward induction over the structure of the derivation of $\Gamma \triangleright e : \tau$. The only inference rule where the context is examined is tp_var which will be applicable if a declaration $x{:}\tau$ is present in the context Γ. It is clear that the presence of additional non-conflicting declarations does not alter this property. □

Type derivations which differ only by weakening in the type declarations Γ have identical structure. Thus we permit the weakening of type declarations in Γ during a structural induction over a typing derivation. The substitution lemma below is also central. It is closely related to the notions of parametric and hypothetical judgments introduced in Chapter 5.

Lemma 2.4 (Substitution) *If* $\Gamma \triangleright e' : \tau'$ *and* $\Gamma, x{:}\tau', \Gamma' \triangleright e : \tau$ *then* $\Gamma, \Gamma' \triangleright [e'/x]e : \tau$.

Proof: By induction over the structure of the derivation $\mathcal{D} :: (\Gamma, x{:}\tau', \Gamma' \triangleright e : \tau)$. The result should be intuitive: wherever x occurs in e we are at a leaf in the typing derivation of e. After substitution of e' for x, we have to supply a derivation showing that e' has type τ' at this leaf position, which exists by assumption. We only show a few cases in the proof in detail; the remaining ones follow the same pattern.

Case: $\mathcal{D} = \dfrac{(\Gamma, x{:}\tau', \Gamma')(x) = \tau'}{\Gamma, x{:}\tau', \Gamma' \triangleright x : \tau'}\,\mathsf{tp\_var}.$

Then $[e'/x]e = [e'/x]x = e'$, so the lemma reduces to showing $\Gamma, \Gamma' \triangleright e' : \tau'$ from $\Gamma \triangleright e' : \tau'$ which follows by weakening.

Case: $\mathcal{D} = \dfrac{(\Gamma, x{:}\tau', \Gamma')(y) = \tau}{\Gamma, x{:}\tau', \Gamma' \triangleright y : \tau}\,\mathsf{tp\_var}$, where $x \neq y$.

In this case, $[e'/x]e = [e'/x]y = y$ and hence the lemma follows from

$$\frac{(\Gamma, \Gamma')(y) = \tau}{\Gamma, \Gamma' \triangleright y : \tau}\,\mathsf{tp\_var}.$$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, x{:}\tau', \Gamma' \triangleright e_1 : \tau_2 \rightarrow \tau_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Gamma, x{:}\tau', \Gamma' \triangleright e_2 : \tau_2\end{array}}{\Gamma, x{:}\tau', \Gamma' \triangleright e_1\ e_2 : \tau_1}\,\mathsf{tp\_app}.$

Then we construct a deduction

$$\frac{\begin{array}{c}\mathcal{E}_1\\ \Gamma, \Gamma' \triangleright [e'/x]e_1 : \tau_2 \rightarrow \tau_1\end{array} \qquad \begin{array}{c}\mathcal{E}_2\\ \Gamma, \Gamma' \triangleright [e'/x]e_2 : \tau_2\end{array}}{\Gamma, \Gamma' \triangleright ([e'/x]e_1)\ ([e'/x]e_2) : \tau_1}\,\mathsf{tp\_app}$$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are known to exist from the induction hypothesis applied to $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. By definition of substitution, $[e'/x](e_1\ e_2) = ([e'/x]e_1)\ ([e'/x]e_2)$, and the lemma is established in this case.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, x{:}\tau', \Gamma', y{:}\tau_1 \triangleright e_2 : \tau_2\end{array}}{\Gamma, x{:}\tau', \Gamma' \triangleright \mathbf{lam}\ y.\ e_2 : \tau_1 \rightarrow \tau_2}\,\mathsf{tp\_lam}.$

In this case we need to apply the induction hypothesis by using $\Gamma', y{:}\tau_1$ for Γ'. This is why the lemma is formulated using the additional context Γ'. From the induction hypothesis and one inference step we obtain

$$\dfrac{\begin{array}{c}\mathcal{E}_1\\ \Gamma, \Gamma', y{:}\tau_1 \triangleright [e'/x]e_2 : \tau_2\end{array}}{\Gamma, \Gamma' \triangleright \mathbf{lam}\ y.\ [e'/x]e_2 : \tau_1 \to \tau_2}\ \mathsf{tp\_lam}$$

which yields the lemma by the equation $[e'/x](\mathbf{lam}\ y.\ e_2) = \mathbf{lam}\ y.\ [e'/x]e_2$ if y is not free in e' and distinct from x. We can assume that these conditions are satisfied, since they can always be achieved by renaming bound variables.

□

The statement of the type preservation theorem below is written in such a way that the induction argument will work directly.

Theorem 2.5 (Type Preservation) *For any e and v, if $e \hookrightarrow v$ is derivable, then for any τ such that $\triangleright\, e : \tau$ is derivable, $\triangleright\, v : \tau$ is also derivable.*

Proof: By induction on the structure of the deduction $\mathcal{D}$ of $e \hookrightarrow v$. The justification "*by inversion*" refers to Lemma 2.2. More directly, from the form of the judgment established by a derivation we draw conclusions about the possible forms of the premise, which, of course, must also derivable.

Case: $\mathcal{D} = \dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\ \mathsf{ev\_z}.$

Then we have to show that for any type τ such that $\triangleright\, \mathbf{z} : \tau$ is derivable, $\triangleright\, \mathbf{z} : \tau$ is derivable. This is obvious.

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

$\triangleright\, \mathbf{s}\ e_1 : \tau$	By assumption
$\triangleright\, e_1 : \mathbf{nat}$ and $\tau = \mathbf{nat}$	By inversion
$\triangleright\, v_1 : \mathbf{nat}$	By ind. hyp. on $\mathcal{D}_1$
$\triangleright\, \mathbf{s}\ v_1 : \mathbf{nat}$	By rule $\mathsf{tp\_s}$

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

$\triangleright\, (\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2\ \mid \mathbf{s}\ x \Rightarrow e_3) : \tau$	By assumption
$\triangleright\, e_2 : \tau$	By inversion
$\triangleright\, v : \tau$	By ind. hyp. on $\mathcal{D}_2$

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

$\triangleright (\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) : \tau$	By assumption
$x{:}\mathbf{nat} \triangleright e_3 : \tau$	By inversion
$\triangleright e_1 : \mathbf{nat}$	By inversion
$\triangleright \mathbf{s}\ v_1' : \mathbf{nat}$	By ind. hyp. on $\mathcal{D}_1$
$\triangleright v_1' : \mathbf{nat}$	By inversion
$\triangleright [v_1'/x]e_3 : \tau$	By the Substitution Lemma 2.4
$\triangleright v : \tau$	By ind. hyp. on $\mathcal{D}_3$

Cases: If $\mathcal{D}$ ends in ev_pair, ev_fst, or ev_snd we reason similar to cases above (see Exercise 2.16).

Case: $$\mathcal{D} = \dfrac{}{\mathbf{lam}\ x.\ e \hookrightarrow \mathbf{lam}\ x.\ e}\ \mathsf{ev\_lam}.$$

This case is immediate as for ev_z.

Case: $$\mathcal{D} = \dfrac{\begin{array}{ccc} \mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1' & e_2 \hookrightarrow v_2 & [v_2/x]e_1' \hookrightarrow v \end{array}}{e_1\ e_2 \hookrightarrow v}\ \mathsf{ev\_app}.$$

$\triangleright e_1\ e_2 : \tau$	By assumption
$\triangleright e_1 : \tau_2 \rightarrow \tau$ and $\triangleright e_2 : \tau_2$ for some τ_2	By inversion
$\triangleright \mathbf{lam}\ x.\ e_1' : \tau_2 \rightarrow \tau$	By ind. hyp. on $\mathcal{D}_1$
$x{:}\tau_2 \triangleright e_1' : \tau$	By inversion
$\triangleright v_2 : \tau_2$	By ind. hyp. on $\mathcal{D}_2$
$\triangleright [v_2/x]e_1' : \tau$	By the Substitution Lemma 2.4
$\triangleright v : \tau$	By ind. hyp. on $\mathcal{D}_3$

Case: $$\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ e_1 \hookrightarrow v_1 & [v_1/x]e_2 \hookrightarrow v \end{array}}{\mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 \hookrightarrow v}\ \mathsf{ev\_letv}.$$

$\triangleright \mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 : \tau$	By assumption
$\triangleright e_1 : \tau_1$ and $x{:}\tau_1 \triangleright e_2 : \tau$ for some τ_1	By inversion
$\triangleright v_1 : \tau_1$	By ind. hyp. on $\mathcal{D}_1$
$\triangleright [v_1/x]e_2 : \tau$	By the Substitution Lemma 2.4
$\triangleright v : \tau$	By ind. hyp. on $\mathcal{D}_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_2\\ [e_1/x]e_2 \hookrightarrow v\end{array}}{\textbf{let name}\ x = e_1\ \textbf{in}\ e_2 \hookrightarrow v}\ \mathsf{ev\_letn}.$

$\triangleright\ \textbf{let name}\ x = e_1\ \textbf{in}\ e_2 : \tau$	By assumption
$\triangleright\ [e_1/x]e_2 : \tau$	By inversion
$\triangleright\ v : \tau$	By ind. hyp. on $\mathcal{D}_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ [\textbf{fix}\ x.\ e_1/x]e_1 \hookrightarrow v\end{array}}{\textbf{fix}\ x.\ e_1 \hookrightarrow v}\ \mathsf{ev\_fix}.$

$\triangleright\ \textbf{fix}\ x.\ e_1 : \tau$	By assumption
$x{:}\tau \triangleright e_1 : \tau$	By inversion
$\triangleright\ [\textbf{fix}\ x.\ e_1/x]e_1 : \tau$	By the Substitution Lemma 2.4
$\triangleright\ v : \tau$	By ind. hyp. on $\mathcal{D}_1$

□

It is important to recognize that this theorem cannot be proved by induction on the structure of the expression e. The difficulty is most pronounced in the cases for **let** and **fix**: The expressions in the premises of these rules are in general much larger than the expressions in the conclusion. Similarly, we cannot prove type preservation by an induction on the structure of the typing derivation of e.

2.7 Further Discussion

Ignoring details of concrete syntax, the Mini-ML language is completely specified by its typing and evaluation rules. Consider a simple simple model of an interaction with an implementation of Mini-ML consisting of two phases: type-checking and evaluation. During the first phase the implementation only accepts expressions e that are well-typed in the empty context, that is, $\triangleright\ e : \tau$ for some τ. In the second phase the implementation constructs and prints a value v such that $e \hookrightarrow v$ is derivable. This model is simplistic in some ways, for example, we ignore the question which values can actually be printed or *observed* by the user. We will return to this point in Section **??**.

Our self-contained language definition by means of deductive systems does not establish a connection between types, values, expressions, and mathematical objects such as partial functions. This can be seen as the subject of *denotational semantics*. For example, we understand intuitively that the expression

$$ss = \textbf{lam}\ x.\ s\ (s\ x)$$

denotes the function from natural numbers to natural numbers that adds 2 to its argument. Similarly,

$$\mathit{pred}_0 = \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{fix}\ y.\ y \mid \mathbf{s}\ x' \Rightarrow x'$$

denotes the partial function from natural numbers to natural numbers that returns the predecessor of any argument greater or equal to 1 and is undefined on 0. But is this intuitive interpretation of expressions justified? As a first step, we establish that the result of evaluation (if one exists) is unique. Recall that expressions that differ only in the names of their bound variables are considered equal.

Theorem 2.6 (Uniqueness of Values) *If $e \hookrightarrow v_1$ and $e \hookrightarrow v_2$ are derivable then $v_1 = v_2$.*

Proof: Straightforward (see Exercise 2.17). □

Intuitively the type **nat** can be interpreted by the set of natural numbers. We write v_{nat} for values v such that $\triangleright\ v : \mathbf{nat}$. It can easily be seen by induction on the structure of the derivation of v_{nat} *Value* that v_{nat} could be defined inductively by

$$v_{\text{nat}} \quad ::= \quad \mathbf{z} \mid \mathbf{s}\ v_{\text{nat}}.$$

The meaning or *denotation* of a value v_{nat}, $[\![v_{\text{nat}}]\!]$, can be defined almost trivially as

$$\begin{array}{rcl} [\![\mathbf{z}]\!] & = & 0 \\ [\![\mathbf{s}\ v_{\text{nat}}]\!] & = & [\![v_{\text{nat}}]\!] + 1. \end{array}$$

It is immediate that this is a bijection between closed values of type **nat** and the natural numbers. The meaning of an arbitrary closed expression e_{nat} of type **nat** can then be defined by

$$[\![e_{\text{nat}}]\!] = \begin{cases} [\![v]\!] & \text{if } e_{\text{nat}} \hookrightarrow v \text{ is derivable} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Determinism of evaluation (Theorem 2.6) tells us that v, if it exists, is uniquely defined. Value soundness 2.1 tells us that v is indeed a value. Type preservation (Theorem 2.5) tells us that v will be a closed expression of type **nat** and thus that the meaning of an arbitrary expression of type **nat**, if it is defined, is a unique natural number. Furthermore, we are justified in overloading the $[\![\cdot]\!]$ notation for values and arbitrary expressions, since values evaluate to themselves (Exercise 2.14).

Next we consider the meaning of expressions of functional type. Intuitively, if $\triangleright\ e : \mathbf{nat} \to \mathbf{nat}$, then the meaning of e should be a partial function from natural numbers to natural numbers. We define this as follows:

$$[\![e]\!](n) = \begin{cases} [\![v_2]\!] & \text{if } e\ v_1 \hookrightarrow v_2 \text{ and } [\![v_1]\!] = n \\ \text{undefined} & \text{otherwise} \end{cases}$$

This definition is well-formed by reasoning similar to the above, using the observation that $[\![\cdot]\!]$ is a bijection between closed values of type **nat** and natural numbers.

Thus we were justified in thinking of the type $\mathbf{nat} \to \mathbf{nat}$ as consisting of partial functions from natural numbers to natural numbers. Partial functions in mathematics are understood in terms of their input/output behavior rather than in terms of their concrete definition; they are viewed *extensionally.* For example, the expressions

$$\begin{array}{rcl} ss & = & \mathbf{lam}\, x.\, s\ (s\ x) \quad \text{and} \\ ss' & = & \mathbf{fix}\, f.\, \mathbf{lam}\, x.\, \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{s}\ (\mathbf{s}\ \mathbf{z}) \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (f\ x') \end{array}$$

denote the same function from natural numbers to natural numbers: $[\![ss]\!] = [\![ss']\!]$. Operationally, of course, they have very different behavior. Thus denotational semantics induces a non-trivial notion of equality between expressions in our language. On the other hand, it is not immediately clear how to take advantage of this equality due to its non-constructive nature. The notion of extensional equality between partial recursive function is not recursively axiomatizable and therefore we cannot write a complete deductive system to prove functional equalities. The denotational approach can be extended to higher types (for example, functions that map functions from natural numbers to natural numbers to functions from natural numbers to natural numbers) in a natural way.

It may seem from the above development that the denotational semantics of a language is uniquely determined. This is not the case: there are many choices. Especially the mathematical domains we use to interpret expressions and the structure we impose on them leave open many possibilites. For more on the subject of denotational semantics see, for example, [Gun92].

In the approach above, the meaning of an expression depends on its type. For example, for the expression $id = \mathbf{lam}\ x.\ x$ we have $\triangleright\ id : \mathbf{nat} \to \mathbf{nat}$ and by the reasoning above we can interpret it as a function from natural numbers to natural numbers. We also have $\triangleright\ id : (\mathbf{nat} \to \mathbf{nat}) \to (\mathbf{nat} \to \mathbf{nat})$, so it also maps every function between natural numbers to itself. This inherent ambiguity is due to our use of Curry's approach where types are assigned to untyped expressions. It can be remedied in two natural ways: we can construct denotations independently of the language of types, or we can give meaning to typing derivations. In the first approach, types can be interpreted as subsets of a universe from which the meanings of untyped expressions are drawn. The disadvantage of this approach is that we have to give meanings to all expressions, even those that are intuitively meaningless, that is, ill-typed. In the second approach, we only give meaning to expressions that have typing derivations. Any possible ambiguity in the assignment of types is resolved, since the typing derivation will choose are particular type for the expression. On the other hand we may have to consider *coherence*: different typing derivations for the same expression and type should lead to the same meaning. At the very least the meanings should be compatible in some way so that arbitrary decisions made during type inference do not lead to observable differences in the behavior of a program. In the Mini-ML language we discussed so far, this property is easily seen to hold, since an expression uniquely determines its typing derivation. For more complex languages this may require non-trivial proof. Note that the ambiguity problem does not usually arise when we choose a language presentation in the style of Church where each expression contains enough type information to uniquely determine its type.

2.8 Exercises

Exercise 2.1 Write Mini-ML programs for multiplication, exponentiation, subtraction, and a function that returns a pair of (integer) quotient and remainder of two natural numbers.

Exercise 2.2 The *principal type* of an expression e is a type τ such that *any* type τ' of e can be obtained by instantiating the type variables in τ. Even though types in our formulation of Mini-ML are not unique, every well-typed expression has a principal type [Mil78]. Write Mini-ML programs satisfying the following informal specifications and determine their principal types.

1. *compose* f g to compute the composition of two functions f and g.
2. *iterate* n f x to iterate the function f n times over x.

Exercise 2.3 Write down the evaluation of $plus_2$ $(\mathbf{s}\ \mathbf{z})$ $(\mathbf{s}\ \mathbf{z})$, given the definition of $plus_2$ in the example on page 11.

Exercise 2.4 Write out the typing derivation that shows that the function *double* on page 17 is well-typed.

Exercise 2.5 Explore a few alternatives to the definition of expressions given in Section 2.1. In each case, give the relevant inference rules for evaluation and typing.

1. Add a type of Booleans and replace the constructs concerning natural numbers by
$$e ::= \ldots \mid \mathbf{z} \mid \mathbf{s}\ e \mid \mathbf{pred}\ e \mid \mathbf{zerop}\ e$$
2. Replace the constructs concerning pairs by
$$e ::= \ldots \mid \mathbf{pair} \mid \mathbf{fst} \mid \mathbf{snd}$$
3. Replace the constructs concerning pairs by
$$e ::= \ldots \mid \langle e_1, e_2 \rangle \mid \mathbf{split}\ e_1\ \mathbf{as}\ \langle x_1, x_2 \rangle \Rightarrow e_2$$

Exercise 2.6 One might consider replacing the rule ev_fst by

$$\frac{e_1 \hookrightarrow v_1}{\mathbf{fst}\ \langle e_1, e_2\rangle \hookrightarrow v_1}\ \mathsf{ev\_fst}'.$$

Show why this is incorrect.

Exercise 2.7 Consider an extension of the language by the unit type 1 (often written as unit) and disjoint sums $\tau_1 + \tau_2$:

$$\begin{array}{rcl} \tau & ::= & \ldots \mid 1 \mid (\tau_1 + \tau_2) \\ e & ::= & \ldots \mid \langle\,\rangle \mid \mathbf{inl}\, e \mid \mathbf{inr}\, e \mid (\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}\ x_2 \Rightarrow e_2 \mid \mathbf{inr}\ x_3 \Rightarrow e_3) \end{array}$$

For example, an alternative to the predecessor function might return $\langle\,\rangle$ if the argument is zero, and the predecessor otherwise. Because of the typing discipline, the expression

$$\mathit{pred}' = \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \langle\,\rangle \mid \mathbf{s}\ x' \Rightarrow x'$$

is not typable. Instead, we have to inject the values into a disjoint sum type:

$$\mathit{pred}' = \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{inl}\ \langle\,\rangle \mid \mathbf{s}\ x' \Rightarrow \mathbf{inr}\ x'$$

so that

$$\triangleright\ \mathit{pred}' : \mathbf{nat} \to (1 + \mathbf{nat})$$

Optional values of type τ can be modelled in general by using the type $(1 + \tau)$.

1. Give appropriate rules for evaluation and typing.

2. Extend the notion of *value*.

3. Extend the proof of value soundness (Theorem 2.1).

4. Extend the proof type preservation (Theorem 2.5).

Exercise 2.8 Consider a language extension

$$\tau \quad ::= \quad \ldots \mid \tau^*.$$

where τ^* is the type of lists whose members have type τ. Introduce appropriate value constructor and destructor expressions and proceed as in Exercise 2.7.

Exercise 2.9 In this exercise we explore the operation of substitution in some more detail than in Section 2.2. We limit ourselves to the fragment containing **lam**-abstraction and application.

1. Define x *free in* e which should hold when the variable x occurs free in e.

2. Define $e =_\alpha e'$ which should hold when e and e' are alphabetic variants, that is, they differ only in the names assigned to their bound variables as explained in Section 2.2.

3. Define $[e'/x]e$, the result of substituting e' for x in e. This operation should avoid capture of variables free in e' and the result should be unique up to renaming of bound variables.

4. Prove $[e'/x]e =_\alpha e$ if x does not occur free in e'.

5. Prove $[e_2/x_2]([e_1/x_1]e) =_\alpha [([e_2/x_2]e_1)/x_1]([e_2/x_2]e)$, provided x_1 does not occur free in e_2.

Exercise 2.10 In this exercise we will explore different ways to treat errors in the semantics.

1. Assume there is a new value **error** of arbitary type and modify the operational semantics appropriately. You may assume that only well-typed expressions are evaluated. For example, evaluation of **s** (**lam** $x.\ x$) does not need to result in **error**.

2. Add an empty type 0 (often called void) containing no values. Are there any closed expressions of type 0? Add a new expression form **abort** e which has arbitrary type τ whenever e has type 0, but add no evaluation rules for **abort**. Do the value soundness and type preservation properties extend to this language? How does this language compare to the one in item 1.

3. An important semantic property of type systems is often summarized as "*well-typed programs cannot go wrong.*" The meaning of ill-typed expressions such as **fst z** would be defined as a distinguished semantic value *wrong* (in contrast to intuitively non-terminating expressions such as **fix** $x.\ x$) and it is then shown that no well-typed expression has meaning *wrong*. A related phrase is that in statically typed languages "*no type-errors can occur at runtime.*" Discuss how these properties might be expressed in the framework presented here and to what extent they are already reflected in the type preservation theorem.

Exercise 2.11 In the language Standard ML [MTH90], occurrences of fixed point expressions are syntactially restricted to the form **fix** x. **lam** $y.\ e$. This means that evaluation of a fixed point expression always terminates in one step with the value **lam** y. [**fix** x. **lam** $y.\ e/x]e$.

It has occasionally been proposed to extend ML so that one can construct recursive values. For example, $\omega =$ **fix** x. **s** x would represent a "circular value" **s** (**s** $\ldots$) which could not be printed finitely. The same value could also be defined, for example, as $\omega' =$ **fix** x. **s** (**s** x).

In our language, the expressions ω and ω' are not values and, in fact, they do not even have a value. Intuitively, their evaluation does not terminate.

Define an alternative semantics for the Mini-ML language that permits recursive values. Modify the definition of values and the typing rules as necessary. Sketch the required changes to the statements and proofs of value soundness, type preservation, and uniqueness of values. Discuss the relative merits of the two languages.

Exercise 2.12 Explore an alternative operational semantics in which expressions that are known to be values (since they have been evaluated) are not evaluated again. State and prove in which way the new semantics is equivalent to the one given in Section 2.3.
Hint: It may be necessary to extend the language of expressions or explicitly separate the language of values from the language of expressions.

Exercise 2.13 Specify a call-by-name operational semantics for our language, where function application is given by

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

We would like constructors (successor and pairing) to be *lazy*, that is, they should not evaluate their arguments. Consider if it still makes sense to have **let val** and **let name** and what their respective rules should be. Modify the affected inference rules, define the notion of a lazy value, and prove that call-by-name evaluation always returns a lazy value. Furthermore, write a function *observe* : **nat** $\rightarrow$ **nat** that, given a lazy value of type **nat**, returns the corresponding eager value if it exists.

Exercise 2.14 Prove that v *Value* is derivable if and only if $v \hookrightarrow v$ is derivable. That is, values are exactly those expressions that evaluate to themselves.

Exercise 2.15 A *replacement lemma* is necessary in some formulations of the type preservation theorem. It states:

> If, for any type τ', $\triangleright\, e_1' : \tau'$ implies $\triangleright\, e_2' : \tau'$, then $\triangleright\, [e_1'/x]e : \tau$ implies $\triangleright\, [e_2'/x]e : \tau$.

Prove this lemma. Be careful to generalize as necessary and clearly exhibit the structure of the induction used in your proof.

Exercise 2.16 Complete the proof of Theorem 2.5 by giving the cases for ev_pair, ev_fst, and ev_snd.

Exercise 2.17 Prove Theorem 2.6.

Exercise 2.18 (*Non-Determinism*) Consider a non-deterministic extension of Mini-ML with two new expression constructors $\circ$ and $e_1 \oplus e_2$ with the evaluation rules

$$\frac{e_1 \hookrightarrow v}{e_1 \oplus e_2 \hookrightarrow v}\,\mathsf{ev\_choice}_1 \qquad\qquad \frac{e_2 \hookrightarrow v}{e_1 \oplus e_2 \hookrightarrow v}\,\mathsf{ev\_choice}_2$$

Thus, $\oplus$ signifies non-deterministic choice, while $\circ$ means failure (choice between zero alternatives).

1. Modify the type system and extend the proofs of value soundness and type preservation.
2. Write an expression that may evaluate to an arbitrary natural number.
3. Write an expression that may evaluate precisely to the numbers that are not prime.
4. Write an expression that may evaluate precisely to the prime numbers.

Exercise 2.19 (*General Pattern Matching*) Patterns for Mini-ML can be defined by

$$\text{Patterns} \quad p \quad ::= \quad x \mid \mathbf{z} \mid \mathbf{s}\, p \mid \langle p_1, p_2 \rangle.$$

Devise a version of Mini-ML where **case** (for natural numbers), **fst**, and **snd** are replaced by a single form of **case**-expression with arbitrarily many branches. Each branch has the form $p \Rightarrow e$, where the variables in p are bound in e.

1. Define an operational semantics.
2. Define typing rules.
3. Prove type preservation and any lemmas you may need. Show only the critical cases in proofs that are very similar to the ones given in the notes.
4. Is your language deterministic? If not, devise a restriction that makes your language deterministic.
5. Does your operational semantics require equality on expressions of functional type? If yes, devise a restriction that requires equality only on *observable types*—in this case (inductively) natural numbers and products of observable type.

Exercise 2.20 Prove that the expressions **let val** $x = e_1$ **in** e_2 and (**lam** $x.\ e_2$) e_1 are equivalent in sense that

1. for any context Γ, $\Gamma \triangleright$ **let val** $x = e_1$ **in** $e_2 : \tau$ iff $\Gamma \triangleright$ (**lam** $x.\ e_2$) $e_1 : \tau$, and
2. **let val** $x = e_1$ **in** $e_2 \hookrightarrow v$ iff (**lam** $x.\ e_2$) $e_1 \hookrightarrow v$.

Is this sufficient to guarantee that if we replace one expression by the other somewhere in a larger program, the value of the whole program does not change?

Exercise 2.21 Carefully define a notion of *subexpression* for Mini-ML and prove that if $\Gamma \triangleright e : \tau$ then every subexpression e' of e is also well-typed in an appropriate context.

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 4. 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.

$$
\begin{array}{llcl}
\text{Types} & A & ::= & a \mid A_1 \rightarrow A_2 \\
\text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \\[4pt]
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}\mathsf{type} \mid \Sigma, c{:}A \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}
$$

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.

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

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

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

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 exp:type through fix:(exp $\to$ exp) $\to$ 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,

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

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

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

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

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

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

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

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

$$\begin{array}{lll}
= & \ulcorner [e_1/x]e_2 \urcorner & \\
= & \ulcorner [e_1/x](\textbf{let val } y = e'_2 \textbf{ in } e''_2) \urcorner & \\
= & \ulcorner \textbf{let val } y = [e_1/x]e'_2 \textbf{ in } [e_1/x]e''_2 \urcorner & \\
= & \mathsf{letv}\ \ulcorner [e_1/x]e'_2 \urcorner\ (\lambda y{:}\mathsf{exp}.\ \ulcorner [e_1/x]e''_2 \urcorner) & \\
= & \mathsf{letv}\ ([\ulcorner e_1 \urcorner/x]\ulcorner e'_2 \urcorner)\ (\lambda y{:}\mathsf{exp}.\ [\ulcorner e_1 \urcorner/x]\ulcorner e''_2 \urcorner) & \text{by induction hypothesis} \\
= & [\ulcorner e_1 \urcorner/x](\mathsf{letv}\ \ulcorner e'_2 \urcorner\ (\lambda y{:}\mathsf{exp}.\ \ulcorner e''_2 \urcorner)) & \\
= & [\ulcorner e_1 \urcorner/x]\ulcorner \textbf{let val } y = e'_2 \textbf{ in } e''_2 \urcorner & \\
= & [\ulcorner e_1 \urcorner/x]\ulcorner e_2 \urcorner. &
\end{array}$$

□

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.

```
eval  :  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 ev_case_z must be an evaluation yielding $\mathbf{z}$, while the corresponding argument to ev_case_z, namely (ev_s 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 ($\mathsf{eval\ z\ z}$) 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

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

The declarations of $\mathsf{ev\_s}$ and $\mathsf{ev\_case\_z}$ 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 ($\mathsf{ev\_case\_z}$ ($\mathsf{ev\_s\ ev\_z}$) $\mathsf{ev\_z}$) is no longer well-typed, since ($\mathsf{ev\_s\ ev\_z}$) has type eval ($\mathsf{s\ z}$) ($\mathsf{s\ z}$), while the first argument to $\mathsf{ev\_case\_z}$ should have type $\mathsf{eval}\ E_1\ \mathsf{z}$ 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 $\mathsf{ev\_case\_z}$. 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,

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

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

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

Note that the right-hand side is now a closed type since Π binds the variable it quantifies. The function $\mathsf{ev\_case\_z}$ 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.

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

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

$\Gamma \vdash_\Sigma A : \mathsf{type}$	A is a valid type
$\Gamma \vdash_\Sigma M : A$	M is a valid object of type A

and auxiliary judgments

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

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.

int	:	type
plus	:	$\mathsf{int} \to \mathsf{int} \to \mathsf{int}$
vector	:	$\mathsf{int} \to \mathsf{type}$

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

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

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.

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

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

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

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

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

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

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 exp $\rightarrow$ 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 (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

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

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

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

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

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

where

$$\begin{array}{rl} & \ulcorner \dfrac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1'\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ e_2 \hookrightarrow v_2\end{array} \qquad \begin{array}{c}\mathcal{D}_3\\ [v_2/x]e_1' \hookrightarrow v\end{array}}{e_1\ e_2 \hookrightarrow v}\,\mathsf{ev\_app} \urcorner \\ = & \mathsf{ev\_app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e_1' \urcorner)\ \ulcorner v_2 \urcorner\ \ulcorner v \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner\ \ulcorner \mathcal{D}_3 \urcorner. \end{array}$$

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

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

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 (eval $((\lambda x{:}\textsf{exp}.\ x)$ z) z), while the actual argument, ev_z, has type eval z 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.

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

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

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

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

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

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

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

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

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

Again we are taking advantage of compositionality in the form

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

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*

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

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

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

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

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 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 5.4. 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,

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

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

$\mathsf{val\_z}$	:	$\mathsf{value\ z}$
$\mathsf{val\_s}$	:	$\Pi E{:}\mathsf{exp}.\ \mathsf{value}\ E \rightarrow \mathsf{value}\ (\mathsf{s}\ E)$
$\mathsf{val\_pair}$	:	$\Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}.\ \mathsf{value}\ E_1 \rightarrow \mathsf{value}\ E_2 \rightarrow \mathsf{value}\ (\mathsf{pair}\ E_1\ E_2)$
$\mathsf{val\_lam}$	:	$\Pi E{:}\mathsf{exp} \rightarrow \mathsf{exp}.\ \mathsf{value}\ (\mathsf{lam}\ E)$

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

This gives rise to the axiom

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

Case:

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

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

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

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

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:

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

Then the induction hypothesis applied to $\mathcal{D}_2$ yields a deduction of $v\ \mathit{Value}$, 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.

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

Case:

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

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.

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

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.

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

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,

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

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

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

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

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

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.

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

The declaration corresponding to vs_s:

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

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

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

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

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

The rule above can then be represented as

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

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 $\mathcal{P} :: v$ 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

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

which is defined via a single axiom

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

By value soundness and the uniqueness of the deduction of v *Value* for a given v, $\mathcal{D} \Longrightarrow \mathcal{P}$ is derivable iff $\mathcal{D} \overset{triv}{\Longrightarrow} \mathcal{P}$ is derivable, but one would hardly claim that $\mathcal{D} \overset{triv}{\Longrightarrow} \mathcal{P}$ 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

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

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 $A \rightarrow K$ 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."

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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
$\Gamma \vdash_\Sigma A \Uparrow \mathsf{type}$	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 **let val** $x = e_1$ **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 $$\frac{e \hookrightarrow v}{\mathbf{s}\ e \hookrightarrow \mathbf{s}\ v}\ \mathsf{ev\_s}$$	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 4 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 $\mathit{observe} : \mathbf{nat} \to \mathbf{nat}$ 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).

Chapter 4

The Elf Programming Language

Elf, thou lovest best, I think,
The time to sit in a cave and drink.
— William Allingham
In Fairy Land [All75]

In Chapter 2 we have seen how deductive systems can be used systematically to specify aspects of the semantics of programming languages. In later chapters, we will see many more examples of this kind, including some examples from logic. In Chapter 3 we explored the logical framework LF as a formal meta-language for the representation of programming languages, their semantics, and their meta-theory. An important motivation behind the development of LF has been to provide a formal basis for the implementation of proof-checking and theorem proving tools, independently of any particular logic or deductive system. Note that search in the context of LF is the dual of type-checking: given a type A, find a closed object M of type A. If such an object M exists we refer to A as *inhabited*. Since types represent judgments and objects represent deductions, this is a natural formulation of the search for a deduction of a judgment via its representation in LF. Unlike type-checking, of course, the question whether a closed object of a given type exists is in general undecidable. The question of general search procedures for LF has been studied by Elliott [Ell89, Ell90] and Pym and Wallen [PW90, Pym90, PW91, Pym92], including the question of unification of LF objects modulo $\beta\eta$-conversion.

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

Our approach to the implementation of algorithms is inspired by logic programming: specifications and programs are written in the same language. In traditional logic programming, the common basis for specifications and implementations has been the logic of Horn clauses; here, the common basis will be the logical framework LF. We would like to emphasize that specifications and programs are generally *not* the same: many specifications are not useful if interpreted as programs, and many programs would not normally be considered specifications. In the logic programming paradigm, execution is the search for a derivation of some instance of a query. The operational semantics of the logic programming language specifies precisely how this search will be performed, given a list of inference rules that constitute the program. Thus, if one understands this operational reading of inference rules, the programmer can obtain the desired execution behavior by defining judgments appropriately. We explain this in more detail in Section **??** and investigate it more formally in Chapter **??**.

Elf is a strongly typed language, since it is directly based on LF. The Elf interpreter must thus perform type reconstruction on programs and queries before executing them. Because of the complex type system of LF, this is a non-trivial task. In fact, it has been shown by Dowek [Dow93] that the general type inference problem for LF is undecidable, and thus not all types may be omitted from Elf programs. The algorithm for type reconstruction which is used in the implementation [Pfe91a, Pfe94] is based on the same constraint solving algorithm employed during execution.

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

4.1 Concrete Syntax

The concrete syntax of Elf is very simple, since we only have to model the relatively few constructs of LF. While LF is stratified into the levels of kinds, families, and objects, the syntax is overloaded in that, for example, the symbol Π constructs dependent function types and dependent kinds. Similarly, juxtaposition is concrete syntax for instantiation of a type family and application of objects. We maintain this overloading in the concrete syntax for Elf and refer to expressions from any of the three levels collectively as *terms*. A signature is given as a sequence of *declarations*. We describe here only the core language which corresponds very closely to LF. The main addition is a form of declaration *id* `:` $term_1$ `=` $term_2$ that introduces an abbreviation *id* for $term_2$.

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

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

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

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

4.2 Type and Term Reconstruction

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

```
exp  : type.  %name exp E x.

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

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

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

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

In Elf's concrete syntax these would be written as

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

A simple deduction, such as

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

is represented in Elf as

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

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

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

The interpreter re-prints the declaration, which indicates that the given judgment holds, that is, the object to the left of the colon has type type to the right of the colon in the *current signature*. The current signature is embodied in the state of the Twelf system and comprises all loaded files. Please see the Twelf User's Guide for details.

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

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

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

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

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

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

The typing judgment

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

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

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

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

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

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

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

the derivation above is concisely represented by

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

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

For debugging purposes it is sometimes useful to know the values of reconstructed types and objects. The front-end of the Elf implementation can thus print the internal and fully explicit form of all the declarations if desired. Type reconstruction is discussed in further detail in the documentation of the implementation. For the remainder of this chapter, the main feature to keep in mind is the duality between implicit quantifiers and implicit arguments.

4.3 A Mini-ML Interpreter in Elf

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

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

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

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

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

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

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

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

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

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

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

where $\mathcal{D}_1$, $\mathcal{D}_2$, and v are still to be filled in. Thus computation in Elf corresponds to bottom-up search for a derivation of a judgment. We solve the currently unsolved subgoals going through the partial deduction in a depth-first, left-to-right manner. So the next step would be to solve

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

We see that only one inference rule can apply, namely ev_z, instantiating D1 to ev_z. Now the subgoal D2 can be matched against the type of ev_s, leading to the further subgoal

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

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

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

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

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

Eliminating the intermediate variables we obtain the same answer that Elf would return.

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

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

One can see that the matching process which is required for this search procedure must allow instantiation of the query as well as the declarations. The problem of finding a common instance of two terms is called *unification*. A unification algorithm for terms in first-order logic was first sketched by Herbrand [Her30]. The first full description of an efficient algorithm for unification was given by Robinson [Rob65], which has henceforth been a central building block for automated theorem proving procedures and logic programming languages. In Elf, Robinson's algorithm is not directly applicable because of the presence of types and λ-abstraction. Huet showed that unification in the simply-typed λ-calculus is undecidable [Hue73], a result later sharpened by Goldfarb [Gol81]. The main difficulty stems from the notion of definitional equality, which can be taken as β or $\beta\eta$-convertibility. Of course, the simply-typed λ-calculus is a subsystem of the LF type theory, and thus unifiability is undecidable for LF as well. A practical semi-decision procedure for unifiability in the simply-typed λ-calculus has been proposed by Huet [Hue75] and used in a number of implementations of theorem provers and logic programming languages [AINP88, Pfe91a, Pau94]. However, the procedure has the drawback that it may not only diverge but also branch, which is difficult to control in logic programming. Thus, in Elf, we have adopted the approach of constraint logic programming languages first proposed by Jaffar and Lassez [JL87], whereby difficult unification problems are postponed and carried along as constraints during execution. We will say more about the exact nature of the constraint solving algorithm employed in Elf in Section **??**. In this chapter, all unification problems encountered will be essentially first-order.

We have not payed close attention to the order of various operations during computation. In the first approximation, the operational semantics of Elf can be described as follows. Assume we are given a list of goals $A_1, \ldots, A_n$ with some free variables. Each type of an object-level constant c in a signature has the form $\Pi y_1{:}B_1 \ldots \Pi y_m{:}B_m.\ C_1 \to \cdots \to C_k \to C$, where C is an atomic type. We call C the *target type* of c. Also, in analogy to logic programming, we call c a *clause*, C the *head* of the clause c. Recall, that some of these quantifiers may remain implicit in Elf. We instantiate $y_1, \ldots, y_m$ with fresh variables $Y_1, \ldots, Y_m$ and unify the resulting instance of C' with A_1, trying each constant in the signature in turn until unification succeeds. Unification may instantiate $C_1, \ldots, C_k$ to $C'_1, \ldots, C'_k$. We now set these up as subgoals, that is, we obtain the new list of goals $C'_k, \ldots, C'_1, A_2, \ldots, A_n$. The object we were looking for will be $c\, Y_1 \ldots Y_n\ M_1 \ldots M_k$, where $M_1, \ldots, M_k$ are the objects of type $C'_1, \ldots, C'_k$, respectively, yet to be determined. We say that the goal A_1 has been *resolved* with the clause c and refer to the process as *back-chaining.* Note that the subgoals will be solved "from the inside out," that is, C'_k is the first one to be considered. If unification should fail and no further constants are available in the signature, we *backtrack*, that is, we return to the most recent point where a goal unified with a clause head (that is, a target type of a constant declaration in a signature) and further choices were available. If there are no such choice points, the overall goal fails.

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

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

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

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

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

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

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

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

```
ev_pair : eval (pair E1 E2) (pair V1 V2)
            <- eval E1 V1
```

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

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

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

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

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

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

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

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

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

The first goal is

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This declaration introduces non-terminating computations into the interpreter. Reconsider the example from page 17, **fix** $x.\ x$. Its representation in Elf is given by `fix ([x:exp] x)`. Attempting to evaluate this expression leads to the following sequence of goals.

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

The step from the original goal to the first subgoal is simply the back-chaining step, instantiating `E` to `[x:exp] x`. The second is a β-reduction required to transform the goal into canonical form, relying on the rule of type conversion. The third goal is then a renaming of the first one, and computation will diverge. This corresponds to the earlier observation (see page 17) that there is no v such that the judgment $\mathbf{fix}\, x.\, x \hookrightarrow v$ is derivable.

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

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

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

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

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

This generates the following answer:

```
V = s (s z),
D =
   ev_app
      (ev_case_s
          (ev_s (ev_s (ev_app (ev_case_z ev_z ev_z)
                         ev_z (ev_fix ev_lam))))
          (ev_s ev_z))
      (ev_s ev_z) (ev_fix ev_lam).
```

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

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

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

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

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

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

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

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

```
?- value (pair z (s z)).
Empty Substitution.
More? n
?- value (fst (pair z (s z))).
no
?- value (lam [x] (fst x)).
Empty Substitution.
More? y
No more solutions
```

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

4.4 An Implementation of Value Soundness

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

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

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

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

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

Case:

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

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.

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

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

Its representation in LF is given by

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

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

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

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

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

Here is the declaration of the vs_fst constant:

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

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

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

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

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

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

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

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

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

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

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

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

of type

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

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

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

P = val_s val_z.
```

The sequence of subgoals considered is

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

This approach to testing the meta-theory is feasible for this simple example. As evaluations become more complicated, however, we would like to use the program for evaluation to generate a appropriate derivations and then transform them. This form of sequencing of computation can be achieved in Elf by using the declaration `%solve c : A`. This will solve the query `A` obtain the first solution `A'` with proof term `M` and then making the definition `c : A' = M`. Later queries can then refer to `c`. For example,

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

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

4.5 Input and Output Modes

Via the judgments-as-types and deductions-as-object principles of representation, Elf unifies concepts which are ordinarily distinct in logic programming languages. For example, a goal is represented as a type in Elf. If we look a little deeper, Elf associates a variable `M` with each goal type `A` such that solving the goal requires finding an object `M` of some instance of `A`. Therefore in some way Elf unifies the concepts of goal and logic variable. On the other hand, the intuition underlying the operational interpretation of judgments makes a clear distinction between construction of a derivation and unification: unification is employed only to see if an inference rule can be applied to reduce a goal to subgoals.

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

Therefore, a typical constant declaration which has the form

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

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

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

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

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

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

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

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

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

```
append  : list -> list -> list -> type.
%mode append +L +K -M.

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

The mode declaration

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

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

This mode verification proceeds as follows. We first consider

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

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

Next we consider the second clause.

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

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

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

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

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

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

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

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

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

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

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

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

```
?- even N.
```

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

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

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

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

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

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

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

enumerates the first 10 answers and then stops.

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

4.6 Exercises

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

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

Exercise 4.3 Implement the optimized version of evaluation from Exercise 2.12 in which values that are substituted for variables during evaluation are not evaluated again. Based on the modified interpreter, implement *bounded evaluation* $e \overset{n}{\hookrightarrow} v$ with the intended meaning that e evaluates to v in at most n steps (for a natural number n). You may make the simplifying but unrealistic assumption that every inference rule represents one step in the evaluation.

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

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

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

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

Chapter 5

Parametric and Hypothetical Judgments

Many deductive systems employ reasoning from hypotheses. We have seen an example in Section 2.5: a typing derivation of a Mini-ML expression requires assumptions about the types of its free variables. Another example occurs in the system of natural deduction in Chapter **??**, where a deduction of the judgment that $A \supset B$ *is true* can be given as a deduction of B *is true* from the hypothesis A *is true*. We refer to a judgment that J is derivable under a hypothesis J' as a *hypothetical judgment*. Its critical property is that we can substitute a derivation $\mathcal{D}$ of J' for every use of the hypothesis J' to obtain a derivation which no longer depends on the assumption J'.

Related is reasoning with parameters, which also occurs frequently. The system of natural deduction provides once again a typical example: we can infer that $\forall x.\ A$ *is true* if we can show that $[a/x]A$ *is true*, where a is a new parameter which does not occur in any undischarged hypothesis. Similarly, in the typing rules for Mini-ML we postulate that every variable is declared at most once in a context Γ, that is, in the rule

$$\frac{\Gamma, x{:}\tau_1 \triangleright e : \tau_2}{\Gamma \triangleright \textbf{lam}\ x.\ e : \tau_1 \rightarrow \tau_2}\ \mathsf{tp\_lam}$$

the variable x is new with respect to Γ (which represents the hypotheses of the derivation). This side condition can always be fulfilled by tacitly renaming the bound variable. We refer to a judgment that J is derivable with parameter x as a *parametric judgment*. Its critical property is that we can substitute an expression t for x throughout a derivation of a parametric judgment to obtain a derivation which no longer depends on the parameter x.

Since parametric and hypothetical judgments are common, it is natural to ask if we can directly support them within the logical framework. The answer is affirmative—the key is the notion of function provided in LF. Briefly, the derivation of a hypothetical judgment is represented by a function which maps a derivation of the hypothesis to a derivation of the conclusion. Applying this function corresponds to substituting a derivation for appeals to the hypothesis. Similarly, the derivation of a parametric judgment is represented by a function which maps an expression to a derivation of the instantiated conclusion. Applying this function corresponds to substituting an expressions for the parameter throughout the parametric derivation.

In the remainder of this chapter we elaborate the notions of parametric and hypothetical judgment and their representation in LF. We also show how to exploit them to arrive at a natural and elegant representation of the proof of type preservation for Mini-ML.

5.1 Closed Expressions

When employing parametric and hypothetical judgments, we must formulate the representation theorems carefully in order to avoid paradoxes. As a simple example, we consider the judgment e *Closed* which expresses that e has no free variables. Expression constructors which do not introduce any bound variables are treated in a straightforward manner.

$$\frac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z} \qquad \frac{e\ \mathit{Closed}}{\mathbf{s}\ e\ \mathit{Closed}}\,\mathsf{clo\_s}$$

$$\frac{e_1\ \mathit{Closed} \qquad e_2\ \mathit{Closed}}{\langle e_1, e_2\rangle\ \mathit{Closed}}\,\mathsf{clo\_pair}$$

$$\frac{e\ \mathit{Closed}}{\mathbf{fst}\ e\ \mathit{Closed}}\,\mathsf{clo\_fst} \qquad \frac{e\ \mathit{Closed}}{\mathbf{snd}\ e\ \mathit{Closed}}\,\mathsf{clo\_snd}$$

$$\frac{e_1\ \mathit{Closed} \qquad e_2\ \mathit{Closed}}{e_1\ e_2\ \mathit{Closed}}\,\mathsf{clo\_app}$$

In order to give a concise formulation of the judgment whenever variables are bound we use hypothetical judgments. For example, in order to conclude that **lam** $x.\ e$ is closed, we must show that e is closed under the assumption that x is closed. The hypothesis about x may only be used in the deduction of e, but not elsewhere. Furthermore, in order to avoid confusion between different bound variables with the same name, we would like to make sure that the name x is not already used, that is, the judgment should be parametric in x. The hypothetical judgment that J is derivable from hypotheses $J_1, \ldots, J_n$ is written as

$$\begin{array}{c} J_1 \ \ldots J_n \\ \vdots \\ J \end{array}$$

The construction of a deduction of a hypothetical judgment should be intuitively clear: in addition to the usual inference rules, we may also use a hypothesis as evidence for a judgment. But we must also indicate where an assumption is *discharged*, that is, after which point in a derivation it is no longer available. We indicate this by providing a name for the hypothesis J and labelling the inference at which the hypothesis is discharged correspondingly. Similarly, we label the inference at which a parameter is discharged. The remaining inference rules for the judgment e *Closed* using this notation are given below.

$$\frac{e_1\ \mathit{Closed} \qquad e_2\ \mathit{Closed} \qquad \begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u}\\ \vdots \\ e_3\ \mathit{Closed}\end{array}}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3)\ \mathit{Closed}}\ \mathsf{clo\_case}^{x,u}$$

$$\frac{\begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u}\\ \vdots \\ e\ \mathit{Closed}\end{array}}{\mathbf{lam}\ x.\ e\ \mathit{Closed}}\ \mathsf{clo\_lam}^{x,u} \qquad \frac{e_1\ \mathit{Closed} \qquad \begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u}\\ \vdots \\ e_2\ \mathit{Closed}\end{array}}{\mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2\ \mathit{Closed}}\ \mathsf{clo\_letv}^{x,u}$$

$$\frac{e_1\ \mathit{Closed} \qquad \begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u}\\ \vdots \\ e_2\ \mathit{Closed}\end{array}}{\mathbf{let\ name}\ x = e_1\ \mathbf{in}\ e_2\ \mathit{Closed}}\ \mathsf{clo\_letn}^{x,u} \qquad \frac{\begin{array}{c}\overline{x\ \mathit{Closed}}^{\,u}\\ \vdots \\ e\ \mathit{Closed}\end{array}}{\mathbf{fix}\ x.\ e\ \mathit{Closed}}\ \mathsf{clo\_fix}^{x,u}$$

In order to avoid ambiguity we assume that in a given deduction, all labels for the inference rules clo_case, clo_lam, clo_letv, clo_letn and clo_fix are distinct. An alternative to this rather stringent, but convenient requirement is suggestive of the representation of hypothetical judgments in LF: we can think of a label u as a variable ranging over deductions. The variable is bound by the inference which discharges the hypothesis.

The following derivation shows that the expression **let name** $f = \mathbf{lam}\ x.\ x\ \mathbf{in}\ f\ (f\ \mathbf{z})$ is closed.

$$\dfrac{\dfrac{\dfrac{}{x\ \mathit{Closed}}\,u}{\mathbf{lam}\ x.\ x\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u} \qquad \dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \qquad \dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \qquad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{f\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{f\ (f\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}}{\mathbf{let\ name}\ f = \mathbf{lam}\ x.\ x\ \mathbf{in}\ f\ (f\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_letn}^{f,w}$$

This deduction has no undischarged assumptions, but it contains subderivations with hypotheses. The right subderivation, for example, would traditionally be written as

$$\dfrac{f\ \mathit{Closed} \qquad \dfrac{f\ \mathit{Closed} \qquad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{f\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{f\ (f\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}.$$

In this notation we can not determine if there are two hypotheses (which happen to coincide) or two uses of the same hypothesis. This distinction may be irrelevant under some circumstances, but in many situations it is critical. Therefore we retain the labels even for hypothetical derivations, with the restriction that the free labels must be used consistently, that is, all occurrences of a label must justify the same hypothesis. The subderivation above then reads

$$\dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \qquad \dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \qquad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{f\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{f\ (f\ \mathbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}.$$

There are certain reasoning principles for hypothetical derivations which are usually not stated explicitly. One of them is that hypotheses need not be used. For example, **lam** $x.\ \mathbf{z}$ is closed as witnessed by the derivation

$$\dfrac{\dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{\mathbf{lam}\ x.\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u}$$

which contains a subdeduction of $\mathbf{z}$ *Closed* from hypothesis $u :: x$ *Closed*. Another principle is that hypotheses may be used more than once and thus, in fact, arbitrarily often. Finally, the order of the hypotheses is irrelevant (although their labelling is not). This means that a hypothetical deduction in this notation could be evidence for a variety of hypothetical judgments which differ in the order of the hypotheses or may contain further, unused hypotheses. One can make these principles explicit as inference rules, in which case we refer to them as *weakening* (hypotheses need not be used), *contraction* (hypotheses may be used more than once), and *exchange* (the order of the hypotheses is irrelevant). We should keep in mind that if these principles do not apply then the judgment should not be considered to be hypothetical in the usual sense, and the techniques below may not apply. These properties have been studied abstractly as *consequence relations* [Gar92].

The example derivation above is not only hypothetical in w, but also parametric in f, and we can therefore substitute an expression such as **lam** $x.\ x$ for f and obtain another valid deduction.

$$\dfrac{\dfrac{}{\textbf{lam}\ x.\,x\ \mathit{Closed}}\,w \qquad \dfrac{\dfrac{}{\textbf{lam}\ x.\,x\ \mathit{Closed}}\,w \qquad \dfrac{}{\textbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{(\textbf{lam}\ x.\,x)\ \textbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{(\textbf{lam}\ x.\,x)\ ((\textbf{lam}\ x.\,x)\ \textbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}$$

If $\mathcal{C} :: e\ \mathit{Closed}$ is parametric in x, then we write $[e'/x]\mathcal{C} :: [e'/x]e\ \mathit{Closed}$ for the result of substituting e' for x in the deduction $\mathcal{C}$. In the example, the deduction still depends on the hypothesis w, which suggests another approach to understanding hypothetical judgments. If a deduction depends on a hypothesis $u :: J$ we can substitute any valid deduction of J for this hypothesis to obtain another deduction which no longer depends on $u :: J$. Let $\mathcal{C}$ be the deduction above and let $\mathcal{C}' ::$ **lam** $x.\ x\ \mathit{Closed}$ be

$$\dfrac{\dfrac{}{x\ \mathit{Closed}}\,u}{\textbf{lam}\ x.\,x\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u}.$$

Note that this deduction is *not* parametric in x, that is, x must be considered a bound variable within $\mathcal{C}'$. The result of substituting $\mathcal{C}'$ for w in $\mathcal{C}$ is

$$\dfrac{\dfrac{\dfrac{}{x\ \mathit{Closed}}\,u}{\textbf{lam}\ x.\,x\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u} \qquad \dfrac{\dfrac{\dfrac{}{x'\ \mathit{Closed}}\,u'}{\textbf{lam}\ x'.\,x'\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x',u'} \quad \dfrac{}{\textbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{(\textbf{lam}\ x'.\,x')\ \textbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}}{(\textbf{lam}\ x.\,x)\ ((\textbf{lam}\ x'.\,x')\ \textbf{z})\ \mathit{Closed}}\,\mathsf{clo\_app}$$

where we have renamed some occurrences of x and u in order to satisfy our global side conditions on parameter names. In general we write $[\mathcal{D}'/u]\mathcal{D}$ for the result of substituting $\mathcal{D}'$ for the hypothesis $u :: J'$ in $\mathcal{D}$, where $\mathcal{D}' :: J'$. During substitution we may need to rename parameters or labels of assumptions to avoid violating side conditions on inference rules. This is analogous to the renaming of bound variables during substitution in terms in order to avoid variable capture.

The representation of the judgment e *Closed* in LF follows the judgment-as-types principle: we introduce a type family 'closed' indexed by an expression.

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

The inference rules that do not employ hypothetical judgments are represented straightforwardly.

$$\begin{array}{lcl}
\mathsf{clo\_z} & : & \mathsf{closed\ z} \\
\mathsf{clo\_s} & : & \Pi E{:}\mathsf{exp}\mathsf{closed}\ E \to \mathsf{closed}\ (\mathsf{s}\ E) \\
\mathsf{clo\_pair} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}. \\
 & & \quad \mathsf{closed}\ E_1 \to \mathsf{closed}\ E_2 \to \mathsf{closed}\ (\mathsf{pair}\ E_1\ E_2) \\
\mathsf{clo\_fst} & : & \Pi E{:}\mathsf{exp}.\ \mathsf{closed}\ E \to \mathsf{closed}\ (\mathsf{fst}\ E) \\
\mathsf{clo\_snd} & : & \Pi E{:}\mathsf{exp}.\ \mathsf{closed}\ E \to \mathsf{closed}\ (\mathsf{snd}\ E) \\
\mathsf{clo\_app} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}. \\
 & & \quad \mathsf{closed}\ E_1 \to \mathsf{closed}\ E_2 \to \mathsf{closed}\ (\mathsf{app}\ E_1\ E_2)
\end{array}$$

Now we reconsider the rule clo_lam.

$$\dfrac{\begin{array}{c}\overline{x\ \mathit{Closed}}\ u \\ \vdots \\ e\ \mathit{Closed}\end{array}}{\mathbf{lam}\ x.\ e\ \mathit{Closed}}\ \mathsf{clo\_lam}^{x,u}$$

The judgment in the premiss is parametric in x and hypothetical in x *Closed*. We thus consider it as a function which, when applied to an e' and a deduction $\mathcal{C} :: e'$ *Closed* yields a deduction of $[e'/x]e$ *Closed*.

$$\begin{array}{lcl}
\mathsf{clo\_lam} & : & \Pi E{:}\mathsf{exp} \to \mathsf{exp}. \\
 & & \quad (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E\ x)) \to \mathsf{closed}\ (\mathsf{lam}\ E)
\end{array}$$

Recall that it is necessary to represent the scope of a binding operator in the language of expressions as a function from expressions to expressions. Similar declarations are necessary for the hypothetical judgments in the premisses of the clo_letv, clo_letn, and clo_fix rules.

$$\begin{array}{lll}
\mathsf{clo\_case} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp}.\ \Pi E_3{:}\mathsf{exp} \to \mathsf{exp}. \\
& & \quad \mathsf{closed}\ E_1 \to \mathsf{closed}\ E_2 \\
& & \quad \to (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E_3\ x)) \\
& & \quad \to \mathsf{closed}\ (\mathsf{case}\ E_1\ E_2\ E_3) \\
\mathsf{clo\_letv} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp} \to \mathsf{exp}. \\
& & \quad \mathsf{closed}\ E_1 \to (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E_2\ x)) \\
& & \quad \to \mathsf{closed}\ (\mathsf{letv}\ E_1\ E_2) \\
\mathsf{clo\_letn} & : & \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp} \to \mathsf{exp}. \\
& & \quad \mathsf{closed}\ E_1 \to (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E_2\ x)) \\
& & \quad \to \mathsf{closed}\ (\mathsf{letn}\ E_1\ E_2) \\
\mathsf{clo\_fix} & : & \Pi E{:}\mathsf{exp} \to \mathsf{exp}. \\
& & \quad (\Pi x{:}\mathsf{exp}.\ \mathsf{closed}\ x \to \mathsf{closed}\ (E\ x)) \to \mathsf{closed}\ (\mathsf{fix}\ E)
\end{array}$$

We refer to the signature which includes expression constructors, the declarations of the family closed and the encodings of the inference rules above as EC.

In order to appreciate how this representation works, it is necessary to understand the representation function $\ulcorner\cdot\urcorner$ on deductions. As usual, the definition of the representation function follows the structure of $\mathcal{C} :: e$ *Closed*. We only show a few typical cases.

Case: $\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1 \\ e_1\ \mathit{Closed}\end{array} \qquad \begin{array}{c}\mathcal{C}_2 \\ e_2\ \mathit{Closed}\end{array}}{e_1\ e_2\ \mathit{Closed}}\ \mathsf{clo\_app}$. Then

$$\ulcorner\mathcal{C}\urcorner = \mathsf{clo\_app}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner\ \ulcorner\mathcal{C}_1\urcorner\ \ulcorner\mathcal{C}_2\urcorner.$$

Case: $\mathcal{C} = \dfrac{\begin{array}{c}\overline{x\ \mathit{Closed}}\ u \\ \mathcal{C}_1 \\ e\ \mathit{Closed}\end{array}}{\mathbf{lam}\ x.\ e\ \mathit{Closed}}\ \mathsf{clo\_lam}^{x.u}$. Then

$$\ulcorner\mathcal{C}\urcorner = \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}_1\urcorner).$$

Case: $\mathcal{C} = \dfrac{}{x\ \mathit{Closed}}\ u$. Then

$$\ulcorner\mathcal{C}\urcorner = u.$$

The example deduction above which is evidence for the judgment **let name** $f =$ **lam** $x.\ x$ **in** f **z** *Closed* is represented as

$$\begin{array}{ll}
\vdash_{EC} & \mathsf{clo\_letn}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ x))\ (\lambda f{:}\mathsf{exp}.\ \mathsf{app}\ f\ \mathsf{z}) \\
& \quad (\mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ x)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ u.\ u)) \\
& \quad (\lambda f{:}\mathsf{exp}.\ \lambda w{:}\mathsf{closed}\ f.\ \mathsf{clo\_app}\ f\ \mathsf{z}\ w\ \mathsf{clo\_z}) \\
& : \mathsf{closed}\ (\mathsf{letn}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ x))\ (\lambda f{:}\mathsf{exp}.\ \mathsf{app}\ f\ \mathsf{z}))
\end{array}$$

To show that the above is derivable we need to employ the rule of type conversion as in the example on page 59. The naive formulation of the soundness of the representation does not take the hypothetical or parametric judgments into account.

Property 5.1 (Soundness of Representation, Version 1)
Given any deduction $\mathcal{C}$ :: e Closed. Then $\vdash_{EC} \ulcorner\mathcal{C}\urcorner \Uparrow \mathsf{closed}\ \ulcorner e\urcorner$.

While this indeed a theorem, it cannot be proven directly by induction—the induction hypothesis will not be strong enough to deal with the inference rules whose premisses require deductions of hypothetical judgments. In order to formulate and prove a more general property we have to consider the representation of hypothetical judgments. As one can see from the example above, the deduction fragment

$$\dfrac{\dfrac{}{f\ \mathit{Closed}}\,w \qquad \dfrac{}{\mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_z}}{f\ \mathbf{z}\ \mathit{Closed}}\,\mathsf{clo\_app}$$

is represented by the LF object

$$\mathsf{clo\_app}\ f\ \mathsf{z}\ w\ \mathsf{clo\_z}$$

which is valid in the context with the declarations $f{:}\mathsf{exp}$ and $w{:}\mathsf{closed}\ f$. In order to make the connection between the hypotheses and the LF context explicit, we retain the labels of the assumptions and explicitly define the representation of a list of hypotheses. Let $\Delta = u_1 :: x_1\ \mathit{Closed}, \ldots, u_n :: x_n\ \mathit{Closed}$ be a list of hypotheses where all labels are distinct. Then

$$\ulcorner\Delta\urcorner = x_1{:}\mathsf{exp}, u_1{:}\mathsf{closed}\ x_1, \ldots, x_n{:}\mathsf{exp}, u_n{:}\mathsf{closed}\ x_n.$$

The reformulated soundness property now references the available hypotheses.

Property 5.2 (Soudness of Representation, Version 2)
Given any deduction $\mathcal{C}$ :: e Closed from hypotheses $\Delta = u_1 :: x_1$ Closed, $\ldots, u_n ::$ x_n Closed. Then $\vdash_{EC} \ulcorner\Delta\urcorner$ *Ctx and*

$$\ulcorner\Delta\urcorner \vdash_{EC} \ulcorner\mathcal{C}\urcorner \Uparrow \mathsf{closed}\ \ulcorner e\urcorner.$$

Proof: The proof is by induction on the structure of $\mathcal{C}$. We show three typical cases.
Case:

$$\mathcal{C} = \dfrac{\begin{matrix}\mathcal{C}_1\\ e_1\ \mathit{Closed}\end{matrix} \qquad \begin{matrix}\mathcal{C}_2\\ e_2\ \mathit{Closed}\end{matrix}}{e_1\ e_2\ \mathit{Closed}}\,\mathsf{clo\_app}.$$

Since $\mathcal{C}$ is a deduction from hypotheses Δ, both $\mathcal{C}_1$ and $\mathcal{C}_2$ are also deductions from hypotheses Δ. From the induction hypothesis we conclude then that

1. $\ulcorner\Delta\urcorner \vdash_{EC} \ulcorner\mathcal{C}_1\urcorner \Uparrow \mathsf{closed}\ \ulcorner e_1\urcorner$, and
2. $\ulcorner\Delta\urcorner \vdash_{EC} \ulcorner\mathcal{C}_2\urcorner \Uparrow \mathsf{closed}\ \ulcorner e_2\urcorner$

are both derivable. Thus, from the type of clo_app,

$$\ulcorner\Delta\urcorner \vdash_{EC} \mathsf{clo\_app}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner\ \ulcorner\mathcal{C}_1\urcorner\ \ulcorner\mathcal{C}_2\urcorner \Uparrow \mathsf{closed}\ (\mathsf{app}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner).$$

It remains to notice that $\ulcorner e_1\ e_2\urcorner = \mathsf{app}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner$.

Case:

$$\mathcal{C} = \dfrac{\begin{array}{c}\overline{x\ \mathit{Closed}}\ u\\ \mathcal{C}_1\\ e\ \mathit{Closed}\end{array}}{\mathbf{lam}\ x.\ e\ \mathit{Closed}}\ \mathsf{clo\_lam}^{x,u}.$$

Then $\mathcal{C}_1$ is a deduction from hypotheses $\Delta, u :: x\ \mathit{Closed}$, and

$$\ulcorner\Delta, u :: x\ \mathit{Closed}\urcorner = \ulcorner\Delta\urcorner, x{:}\mathsf{exp}, u{:}\mathsf{closed}\ x.$$

By the induction hypothesis on $\mathcal{C}_1$ we thus conclude that

$$\ulcorner\Delta\urcorner, x{:}\mathsf{exp}, u{:}\mathsf{closed}\ x \vdash_{EC} \ulcorner\mathcal{C}_1\urcorner \Uparrow \mathsf{closed}\ \ulcorner e\urcorner$$

is derivable. Hence, by two applications of the canpi rule for canonical forms,

$$\ulcorner\Delta\urcorner \vdash_{EC} \lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}_1\urcorner \Uparrow \Pi x{:}\mathsf{exp}.\ \Pi u{:}\mathsf{closed}\ x.\ \mathsf{closed}\ \ulcorner e\urcorner$$

is also derivable.

By the representation theorem for expressions (Theorem 3.6) and the weakening for LF we also know that

$$\ulcorner\Delta\urcorner \vdash_{EC} \lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner \Uparrow \mathsf{exp} \to \mathsf{exp}$$

is derivable. From this and the type of clo_lam we infer

$$\begin{array}{lll}\ulcorner\Delta\urcorner & \vdash_{EC} & \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\\ & & \quad\downarrow (\Pi x{:}\mathsf{exp}.\ \Pi u{:}\mathsf{closed}\ x.\ \mathsf{closed}\ ((\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\ x))\\ & & \qquad\quad \to \mathsf{closed}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)).\end{array}$$

By the rule atmcnv, using one β-conversion in the type above, we conclude

$$\begin{array}{lll}\ulcorner\Delta\urcorner & \vdash_{EC} & \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\\ & & \quad\downarrow (\Pi x{:}\mathsf{exp}.\ \Pi u{:}\mathsf{closed}\ x.\ \mathsf{closed}\ \ulcorner e\urcorner)\\ & & \qquad\quad \to \mathsf{closed}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)).\end{array}$$

Using the rules atmapp and cancon which are now applicable we infer

$$\begin{array}{ll}\ulcorner\Delta\urcorner \quad \vdash_{EC} & \mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}_1\urcorner) \\ & \Uparrow \mathsf{closed}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)),\end{array}$$

which is the desired conclusion since $\ulcorner\mathbf{lam}\ x.\ e\urcorner = \mathsf{lam}\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)$.

Case:

$$\mathcal{C} = \dfrac{}{x\ \mathit{Closed}}\,u.$$

Then $\ulcorner\mathcal{C}\urcorner = u$, to which $\ulcorner\Delta\urcorner$ assigns type $\mathsf{closed}\ x = \mathsf{closed}\ \ulcorner x\urcorner$, which is what we needed to show.

□

The inverse of the representation function, $\llcorner\cdot\lrcorner$, is defined on canonical objects C of type closed E for some E of type exp. This is sufficient for the adequacy theorem below—one can extend it to arbitrary valid objects via conversion to canonical form. Again, we only show three critical cases.

$$\llcorner\mathsf{clo\_app}\ E_1\ E_2\ C_1\ C_2\lrcorner \;=\; \dfrac{\llcorner C_1\lrcorner \qquad \llcorner C_2\lrcorner}{\llcorner E_1\lrcorner\ \llcorner E_2\lrcorner\ \mathit{Closed}}\,\mathsf{clo\_app}$$

$$\llcorner\mathsf{clo\_lam}\ (\lambda x{:}\mathsf{exp}.\ E)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ C_1)\lrcorner \;=\; \dfrac{\begin{array}{c}\dfrac{}{x\ \mathit{Closed}}\,u \\ \llcorner C_1\lrcorner \\ \llcorner E\lrcorner\ \mathit{Closed}\end{array}}{\mathbf{lam}\ x.\ \llcorner E\lrcorner\ \mathit{Closed}}\,\mathsf{clo\_lam}^{x,u}$$

$$\llcorner u\lrcorner \;=\; \dfrac{}{x\ \mathit{Closed}}\,u$$

The last case reveals that the inverse of the representation function should be parameterized by a context so we can find the x which is assumed to be closed according to hypothesis u. Alternatively, we can assume that we always know the type of the canonical object we are translating to a deduction. Again, we are faced with the problem that the natural theorem regarding the function $\llcorner\cdot\lrcorner$ cannot be proved directly by induction.

Property 5.3 (Completeness of Representation, Version 1) *Given LF objects E and C such that $\vdash_{EC} E \Uparrow \mathsf{exp}$ and $\vdash_{EC} C \Uparrow \mathsf{closed}\ E$. Then $\llcorner C\lrcorner :: \llcorner E\lrcorner$ Closed.*

In order to prove this, we generalize it to allow appropriate contexts. These contexts need to be translated to an appropriate list of hypotheses. Let Γ be a context of the form $x_1{:}\mathsf{exp}, u_1{:}\mathsf{closed}\ x_1, \ldots, x_n{:}\mathsf{closed}\ x_n$. Then $\llcorner\Gamma\lrcorner$ is the list of hypotheses $u_1 :: x_1$ *Closed*$, \ldots, u_n :: x_n$ *Closed*.

Property 5.4 (Completeness of Representation, Version 2) *Given a context* $\Gamma = x_1{:}\mathsf{exp}, u_1{:}\mathsf{closed}\ x_1, \ldots, x_n{:}\mathsf{closed}\ x_n$ *and LF objects E and C such that* $\Gamma \vdash_{EC} E \Uparrow \mathsf{exp}$ *and* $\Gamma \vdash_{EC} C \Uparrow \mathsf{closed}\ E$. *Then* $\llcorner C\lrcorner :: \llcorner E\lrcorner$ *Closed is a valid deduction from hypotheses* $\llcorner\Gamma\lrcorner$. *Moreoever,* $\llcorner\ulcorner\mathcal{C}\urcorner\lrcorner = \mathcal{C}$ *and* $\llcorner\ulcorner\Delta\urcorner\lrcorner = \Delta$ *for deductions* $\mathcal{C} :: e$ *Closed and hypotheses* Δ.

Proof: By induction on the structure of the derivation $\Gamma \vdash_{EC} C \Uparrow \mathsf{closed}\ E$. The restriction to contexts Γ of a certain form is crucial in this proof (see Exercise 5.1). □

The usual requirement that $\ulcorner\cdot\urcorner$ be a compositional bijection can be understood in terms of substitution for deductions. Let $\mathcal{C} :: e$ *Closed* be a deduction from hypothesis $u :: x$ *closed*. Then compositionality of the representation function requires

$$\ulcorner[\mathcal{C}'/u][e'/x]\mathcal{C}\urcorner = [\ulcorner\mathcal{C}'\urcorner/u][\ulcorner e'\urcorner/x]\ulcorner\mathcal{C}\urcorner$$

whenever $\mathcal{C}' :: e'$ *Closed*. Note that the substitution on the left-hand side is substitution for undischarged hypotheses in a deduction, while substitution on the right is at the level of LF objects. Deductions of parametric and hypothetical judgments are represented as functions in LF. Applying such functions means to substitute for deductions, which can be exhibited if we rewrite the right-hand side of the equation above, preserving definitional equality.

$$[\ulcorner\mathcal{C}'\urcorner/u][\ulcorner e'\urcorner/x]\ulcorner\mathcal{C}\urcorner \equiv (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}\urcorner)\ \ulcorner e'\urcorner\ \ulcorner\mathcal{C}'\urcorner$$

The discipline of dependent function types ensures the validity of the object on the right-hand side:

$$(\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}\urcorner)\ \ulcorner e'\urcorner : [\ulcorner e'\urcorner/x](\mathsf{closed}\ x \to \mathsf{closed}\ \ulcorner e\urcorner).$$

Hence, by compositionality of the representation for expressions,

$$(\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{closed}\ x.\ \ulcorner\mathcal{C}\urcorner)\ \ulcorner e'\urcorner : \mathsf{closed}\ \ulcorner e'\urcorner \to \mathsf{closed}\ \ulcorner[e'/x]e\urcorner$$

and the application of this object to $\ulcorner\mathcal{C}'\urcorner$ is valid and of the appropriate type.

Theorem 5.5 (Adequacy) *There is a bijection between deductions* $\mathcal{C} :: e$ *Closed from hypotheses* $u_1 :: x_1$ *Closed*$, \ldots, u_n :: x_n$ *Closed and LF objects C such that*

$$x_1{:}\mathsf{exp}, u_1{:}\mathsf{closed}\ x_1, \ldots, x_n{:}\mathsf{exp}, u_n{:}\mathsf{closed}\ x_n \vdash_{EC} C \Uparrow \mathsf{closed}\ \ulcorner e\urcorner.$$

The bijection is compositional in the sense that for an expression e_i and a deduction $\mathcal{C}_i :: e_i$ Closed, we have

$$\ulcorner [\mathcal{C}_i/u_i][e_i/x_i]\mathcal{C} \urcorner = [\ulcorner \mathcal{C}_i \urcorner/u_i][\ulcorner e_i \urcorner/x_i]\ulcorner \mathcal{C} \urcorner$$

Proof: Properties 5.2 and 5.4 show the existence of the bijection. To show that it is compositional we reason by induction over the structure of $\mathcal{C}$ (see Exercise 5.2). □

5.2 Function Types as Goals in Elf

Below we give the transcription of the LF signature above in Elf.

```
closed : exp -> type.  %name closed U u.

% Natural Numbers
clo_z     : closed z.
clo_s     : closed (s E)
             <- closed E.
clo_case  : closed (case E1 E2 E3)
             <- closed E1
             <- closed E2
             <- ({x:exp} closed x -> closed (E3 x)).

% Pairs
clo_pair : closed (pair E1 E2)
              <- closed E1
              <- closed E2.
clo_fst  : closed (fst E)
              <- closed E.
clo_snd  : closed (snd E)
              <- closed E.

% Functions
clo_lam : closed (lam E)
             <- ({x:exp} closed x -> closed (E x)).
clo_app : closed (app E1 E2)
             <- closed E1
             <- closed E2.


% Definitions
```

```
clo_letv : closed (letv E1 E2)
             <- closed E1
             <- ({x:exp} closed x -> closed (E2 x)).
clo_letn : closed (letn E1 E2)
             <- closed E1
             <- ({x:exp} closed x -> closed (E2 x)).

% Recursion
clo_fix : closed (fix E)
            <- ({x:exp} closed x -> closed (E x)).
```

Note that we have changed the order of arguments as in other examples. It seems reasonable to expect that this signature could be used as a program to determine if a given object e of type `exp` is closed. Let us consider the subgoals as they arise in a query to check if **lam** $y.\ y$ is closed.

```
?- closed (lam [y:exp] y).
% Resolved with clause clo_lam
?- {x:exp} closed x -> closed (([y:exp] y) x).
```

Recall that solving a goal means to find a closed expression of the query type. Here, the query type is a (dependent) function type. From Theorem 3.13 we know that if a closed object of type $\Pi x{:}A.\ B$ exists, then there is a definitionally equal object of the form $\lambda x{:}A.\ M$ such that M has type B in the context augmented with the assumption $x{:}A$. It is thus a complete strategy in this case to make the assumption that `x` has type `exp` and solve the goal

```
?- closed x -> closed (([y:exp] y) x).
```

However, `x` now is not a free variable in same sense as `V` in the query

```
?- eval (lam [y:exp] y) V.
```

since it is not subject to instantiation during unification. In order to distinguish these different kinds of variables, we call variables which are subject to instantiation *logic variables* and variables which act as constants to unification *parameters.* Unlike logic variables, parameters are shown as lower-case constants. Thus the current goal might be presented as

```
x : exp
?- closed x -> closed (([y:exp] y) x).
```

Here we precede the query with the typings for the current parameters. Now recall that `A -> B` is just a concrete syntax for `{_:A} B` where `_` is an anonymous variable which cannot appear free in `B`. Thus, this case is handled similarly: we introduce a new parameter `u` of type `closed x` and then solve the subgoal

```
x : exp
u : closed x
?- closed (([y:exp] y) x).
```

By an application of β-conversion this is transformed into the equivalent goal

```
x : exp
u : closed x
?- closed x.
```

Now we can use the parameter `u` as the requested object of type `closed x` and the query succeeds without further subgoals.

We now briefly consider, how the appropriate closed object of the original query, namely `?- closed (lam [y:exp] y).` would be constructed. Recall that if $\Gamma, x{:}A \vdash_\Sigma M : B$ then $\Gamma \vdash_\Sigma \lambda x{:}A.\ M : \Pi x{:}A.\ B$. Using this we can now through the trace of the search in reverse, constructing inhabiting objects as we go along and inserting conversions where necessary.

```
                 u : closed x.
     [u:closed x] u : closed x -> closed x.
[x:exp][u:closed x] u : {x:exp} closed x -> closed x.
[x:exp][u:closed x] u : {x:exp} closed x -> closed (([y:exp] y) x).
clo_lam ([x:exp][u:closed x] u)
                    : closed (lam [y:exp] y).
```

Just as in Prolog, search proceeds according to a fixed operational semantics. This semantics specifies that clauses (that is, LF constant declarations) are tried in order from the first to the last. Before referring to the fixed signature, however, the temporary hypotheses are consulted, always considering the most recently introduced parameter first. After all of them have been considered, then the current signature is traversed. In this example the search order happens to be irrelevant as there will always be at most one assumption available for any expression parameter.

The representations of parametric and hypothetical judgments can also be given directly at the top-level. Here are two examples: the first to find the representation of the hypothetical deduction of $f\ (f\ \mathbf{z})$ *closed* from the hypothesis f *closed*, the second to illustrate failure when given an expression ($\langle x, x\rangle$) which is not closed.

```
?- Q : {f:exp} closed f -> closed (app f (app f z)).

Q = [f:exp] [u:closed f] clo_app (clo_app clo_z u) u.
More? y
no more solutions
?- Q : {x:exp} closed (pair x x).

no
```

Note that the quantification on the variable `x` is necessary, since the query `?- closed (pair x x).` is considered to contain an undeclared constant `x` (which is an error), and the query `?- closed (pair X X)` considers `X` as a logic variable subject to instantation:

```
?- Q : closed (pair X X).
Solving...

X = z,
Q = clo_pair clo_z clo_z.
More? y

X = s z,
Q = clo_pair (clo_s clo_z) (clo_s clo_z).

yes
```

5.3 Negation

Now that we have seen how to write a program to detect closed expressions, how do we write a program which succeeds if an expression is *not* closed? In Prolog, one has the possibility of using the unsound technique of negation-as-failure to write a predicate which succeeds if and only if another predicate fails finitely. In Elf, this technique is not available. Philosophically one might argue that the absence of evidence for e *Closed* does not necessarily mean that e is not closed. More pragmatically, note that if we possess evidence that e is closed, then this will continue to be evidence regardless of any further inference rules or hypotheses we might introduce to demonstrate that expressions are closed. However, the judgment that $\langle x, x\rangle$ is *not* closed does not persist if we add the hypothesis that x is closed. Only under a so-called *closed-world assumption*, that is, the assumption that no further hypotheses or inference rules will be considered, is it reasonable to conclude the $\langle x, x\rangle$ is not closed. The philosophy behind the logical framework is that we work with an implicit *open-world assumption*, that is, all judgments, once judged to be evident since witnessed by a deduction, should remain evident under extensions of the current rules of inference. Note that this is clearly *not* the case for the meta-theorems we prove. Their proofs rely on induction on the structure of derivations and they may no longer be valid when further rules are added.

Thus it is necessary to explicitly define a judgment e *Open* to provide means for giving evidence that e is open, that is, it contains at least one free variable. Below is the implementation of such a judgment in Elf.

```
open : exp -> type.  %name open v
```

```
% Natural Numbers
open_s     : open (s E) <- open E.
open_case1 : open (case E1 E2 E3) <- open E1.
open_case2 : open (case E1 E2 E3) <- open E2.
open_case3 : open (case E1 E2 E3) <- ({x:exp} open (E3 x)).

% Pairs
open_pair1 : open (pair E1 E2) <- open E1.
open_pair2 : open (pair E1 E2) <- open E2.
open_fst   : open (fst E) <- open E.
open_snd   : open (snd E) <- open E.

% Functions
open_lam  : open (lam E) <- ({x:exp} open (E x)).
open_app1 : open (app E1 E2) <- open E1.
open_app2 : open (app E1 E2) <- open E2.

% Definitions
open_letv1 : open (letv E1 E2) <- open E1.
open_letv2 : open (letv E1 E2) <- ({x:exp} open (E2 x)).
open_letn1 : open (letn E1 E2) <- open E1.
open_letn2 : open (letn E1 E2) <- ({x:exp} open (E2 x)).

% Recursion
open_fix : open (fix E) <- ({x:exp} open (E x)).
```

One curious fact about this judgment is that there is no base case, that is, without any hypotheses any query of the form `?- open` $\ulcorner e \urcorner$. will fail! That is, with a given query we must provide evidence that any parameters which may occur in it are open. For example,

```
?- Q : {x:exp} open (pair x x).
no
?- Q : {x:exp} open x -> open (pair x x).

Empty Substitution.
Q = [x:exp] [v:open x] open_pair1 v.
More? y
Empty Substitution.
Q = [x:exp] [v:open x] open_pair2 v.
More? y
No more solutions
```

```
?- Q : {x:exp} open x -> open (lam [x:exp] pair x x).
no
```

5.4 Representing Mini-ML Typing Derivations

In this section we will show a natural representation of Mini-ML typing derivations in LF. In order to avoid confusion between the contexts of LF and the contexts of Mini-ML, we will use Δ throughout the remainder of this chapter to designate a Mini-ML context. The typing judgment of Mini-ML then has the form

$$\Delta \triangleright e : \tau$$

and expresses that e has type τ in context Δ. We observe that this judgment can be interpreted as a hypothetical judgment with hypotheses Δ. There is thus an alternative way to describe the judgment $e : \tau$ which employs hypothetical judgments without making assumptions explicit.

$$\frac{\Delta, x{:}\tau_1 \triangleright e : \tau_2}{\Delta \triangleright \mathbf{lam}\ x.\ e : \tau_1 \to \tau_2}\ \mathsf{tp\_lam} \qquad \frac{\begin{array}{c}\overline{\triangleright x : \tau_1}\ u \\ \vdots \\ \triangleright e : \tau_2\end{array}}{\triangleright \mathbf{lam}\ x.\ e : \tau_1 \to \tau_2}\ \mathsf{tp\_lam}^{x,u}$$

The judgment in the premiss in the formulation of the rule on the right is parametric in x and hypothetical in $u :: x{:}\tau_1$. On the left, all available hypothesis are represented explicitly. The restriction that each variable may be declared at most once in a context and bound variables may be renamed tacitly encodes the parametricity with respect to x.

First, however, the representation of Mini-ML types. We declare an LF type constant, tp, for the representation of Mini-ML types. Recall, from Section 2.5,

$$\text{Types} \quad \tau \quad ::= \quad \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \alpha$$

It is important to bear in mind that $\to$ is overloaded here, since it stands for the function type constructor in Mini-ML and in LF. It should be clear from the context which constructor is meant in each instance. The representation function and the LF declarations are straightforward.

$$\begin{array}{rcl@{\qquad}rcl}
 & & & \mathsf{tp} & : & \mathsf{type} \\
\ulcorner\alpha\urcorner & = & \alpha & & & \\
\ulcorner\mathbf{nat}\urcorner & = & \mathsf{nat} & \mathsf{nat} & : & \mathsf{tp} \\
\ulcorner\tau_1 \times \tau_2\urcorner & = & \mathsf{cross}\ \ulcorner\tau_1\urcorner\ \ulcorner\tau_2\urcorner & \mathsf{cross} & : & \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp} \\
\ulcorner\tau_1 \to \tau_2\urcorner & = & \mathsf{arrow}\ \ulcorner\tau_1\urcorner\ \ulcorner\tau_2\urcorner & \mathsf{arrow} & : & \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}
\end{array}$$

Here α on the right-hand side stands for a variable named α in the LF type theory. We refer to the signature in the right-hand column as T. We briefly state (without proof) the representation theorem.

Theorem 5.6 (Adequacy) *The representation function $\ulcorner\cdot\urcorner$ is a compositional bijection between Mini-ML types and canonical LF objects of type* tp *over the signature T.*

Now we try to apply the techniques for representing hypothetical judgments developed in Section 5.1 to the representation of the typing judgment (for an alternative, see Exercise 5.6). The representation will be as a type family 'of' such that

$$\ulcorner\Delta\urcorner \vdash \ulcorner\mathcal{P}\urcorner : \mathsf{of}\ \ulcorner e\urcorner\ \ulcorner\tau\urcorner$$

whenever $\mathcal{P}$ is a deduction of $\Delta \triangleright e : \tau$. Thus,

$$\mathsf{of} \quad : \quad \mathsf{exp} \to \mathsf{tp} \to \mathsf{type}$$

with the representation for Mini-ML contexts Δ as LF contexts $\ulcorner\Delta\urcorner$.

$$\begin{array}{rcl} \ulcorner\cdot\urcorner & = & \cdot \\ \ulcorner\Delta, x{:}\tau\urcorner & = & \ulcorner\Delta\urcorner, x{:}\mathsf{exp}, u{:}\mathsf{of}\ x\ \ulcorner\tau\urcorner \end{array}$$

Here u must be chosen to be different from x and any other variable in $\ulcorner\Delta\urcorner$ in order to satisfy the general assumption about LF contexts. This assumption can be satisfied since we made a similar assumption about Δ.

For typing derivation themselves, we only show three critical cases in the definition of $\ulcorner\mathcal{P}\urcorner$ for $\mathcal{P} :: \Delta \triangleright e : \tau$. The remainder is given directly in Elf later. The type family $\mathsf{of} : \mathsf{exp} \to \mathsf{tp} \to \mathsf{type}$ represents the judgment $e : \tau$.

Case:

$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1 \\ \Delta \triangleright e_1 : \tau_2 \to \tau_1\end{array} \qquad \begin{array}{c}\mathcal{P}_2 \\ \Delta \triangleright e_2 : \tau_2\end{array}}{\Delta \triangleright e_1\ e_2 : \tau_1}\ \mathsf{tp\_app}.$$

In this simple case we let

$$\ulcorner\mathcal{P}\urcorner = \mathsf{tp\_app}\ \ulcorner\tau_1\urcorner\ \ulcorner\tau_2\urcorner\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner\ \ulcorner\mathcal{P}_1\urcorner\ \ulcorner\mathcal{P}_2\urcorner$$

where

$$\begin{array}{lcl} \mathsf{tp\_app} & : & \Pi T_1{:}\mathsf{tp}.\ \Pi T_2{:}\mathsf{tp}.\ \Pi E_1{:}\mathsf{exp}.\ \Pi E_2{:}\mathsf{exp} \\ & & \quad \mathsf{of}\ E_1\ (\mathsf{arrow}\ T_2\ T_1) \to \mathsf{of}\ E_2\ T_2 \to \mathsf{of}\ (\mathsf{app}\ E_1\ E_2)\ T_1 \end{array}$$

Case:

$$\mathcal{P} = \cfrac{\begin{array}{c}\mathcal{P}' \\ \Delta, x{:}\tau_1 \triangleright e : \tau_2\end{array}}{\Delta \triangleright \textbf{lam}\ x.\ e : \tau_1 \to \tau_2}\ \mathsf{tp\_lam}.$$

In this case we view $\mathcal{P}'$ as a deduction of a hypothetical judgment, that is, a derivation of $e : \tau_2$ from the hypothesis $x{:}\tau_1$. We furthermore note that $\mathcal{P}'$ is parametric in x and choose an appropriate functional representation.

$$\ulcorner\mathcal{P}\urcorner \quad = \quad \mathsf{tp\_lam}\ \ulcorner\tau_1\urcorner\ \ulcorner\tau_2\urcorner\ (\lambda x{:}\mathsf{exp}.\ \ulcorner e\urcorner)\ (\lambda x{:}\mathsf{exp}.\ \lambda u{:}\mathsf{of}\ x\ \ulcorner\tau_1\urcorner.\ \ulcorner\mathcal{P}'\urcorner)$$

The constant $\mathsf{tp\_lam}$ must thus have the following type:

$$\begin{array}{lll}\mathsf{tp\_lam} & : & \Pi T_1{:}\mathsf{tp}.\ \Pi T_2{:}\mathsf{tp}.\ \Pi E{:}\mathsf{exp} \to \mathsf{exp}. \\ & & \quad (\Pi x{:}\mathsf{exp}.\ \mathsf{of}\ x\ T_1 \to \mathsf{of}\ (E\ x)\ T_2) \\ & & \quad \to \mathsf{of}\ (\mathsf{lam}\ E)\ (\mathsf{arrow}\ T_1\ T_2).\end{array}$$

Representation of a deduction $\mathcal{P}$ with hypotheses Δ requires unique labels for the various hypotheses, in order to return the appropriate variable whenever an hypothesis is used. While we left this correspondence implicit, it should be clear that in the case of $\ulcorner\mathcal{P}'\urcorner$ above, the hypothesis $x{:}\tau_1$ should be considered as labelled by u.

Case:

$$\mathcal{P} = \cfrac{\Delta(x) = \tau}{\Delta \triangleright x : \tau}\ \mathsf{tp\_var}.$$

This case is not represented using a fixed inference rule, but we will have a variable u of type '$\mathsf{of}\ x\ \ulcorner\tau\urcorner$' which implicitly provides a label for the assumption x. We simply return this variable.

$$\ulcorner\mathcal{P}\urcorner \quad = \quad u$$

The adequacy of this representation is now a straightforward exercise, given in the following two properties. We refer to the full signature (which includes the signatures T for Mini-ML types and E for Mini-ML expressions) as TD.

Property 5.7 (Soundness) *If* $\mathcal{P} :: \Delta \triangleright e : \tau$ *then* $\vdash_{TD} \ulcorner\Delta\urcorner$ *Ctx and*

$$\ulcorner\Delta\urcorner \vdash_{TD} \ulcorner\mathcal{P}\urcorner \Uparrow \mathsf{of}\ \ulcorner e\urcorner\ \ulcorner\tau\urcorner.$$

Property 5.8 (Completeness) *Let Δ be a Mini-ML context and $\Gamma = \ulcorner\Delta\urcorner$. If $\vdash_{TD} E \Uparrow \mathsf{exp}$, $\vdash_{TD} T \Uparrow \mathsf{tp}$, and*

$$\Gamma \vdash_{TD} P \Uparrow \mathsf{of}\, E\, T$$

then there exist e, τ, and a derivation $\mathcal{P} :: \Delta \triangleright e : \tau$ such that $\ulcorner e\urcorner = E$, $\ulcorner T\urcorner = \tau$, and $\ulcorner \mathcal{P}\urcorner = P$.

It remains to understand the compositionality property of the bijection. We reconsider the substitution lemma (Lemma 2.4):

If $\Delta \triangleright e_1 : \tau_1$ and $\Delta, x_1{:}\tau_1 \triangleright e_2 : \tau_2$ then $\Delta \triangleright [e_1/x_1]e_2 : \tau_2$.

The proof is by induction on the structure of $\mathcal{P}_2 :: (\Delta, x_1{:}\tau_1 \triangleright e_2 : \tau_2)$. Wherever the assumption $x_1{:}\tau_1$ is used, we substitute a version of the derivation $\mathcal{P}_1 :: \Delta \triangleright e_1 : \tau_1$ where some additional (and unused) typing assumptions may have been added to Δ. A reformulation using the customary notation for hypothetical judgments exposes the similarity to the considerations for the judgment e *Closed* considered in Section 5.1.

$$\text{If} \quad \begin{array}{c} \overline{\phantom{\triangleright x_1{:}\tau_1}}\, u_1 \\ \triangleright x_1{:}\tau_1 \\ \mathcal{P}_2 \\ \triangleright e_2{:}\tau_2 \end{array} \quad \text{and} \quad \begin{array}{c} \mathcal{P}_1 \\ \triangleright e_1{:}\tau_1 \end{array} \quad \text{then} \quad \begin{array}{c} \mathcal{P}_1 \\ \overline{\phantom{\triangleright e_1{:}\tau_1}}\, u_1 \\ \triangleright e_1{:}\tau_1 \\ [e_1/x_1]\mathcal{P}_2 \\ \triangleright [e_1/x_1]e_2 : \tau_2 \end{array}$$

Here, $[e_1/x_1]\mathcal{P}_2$ is the substitution of e_1 for x_1 in the deduction $\mathcal{P}_2$, which is legal since $\mathcal{P}_2$ is a deduction of a judgment parametric in x_1. Furthermore, the deduction $\mathcal{P}_1$ has been substituted for the hypotheses labelled u in $\mathcal{P}_2$, indicated by writing $\mathcal{P}_1$ above the appropriate hypothesis. Using the conventions established for hypothetical and parametric judgments, the final deduction above can also be written as $[\mathcal{P}_1/u_1][e_1/x_1]\mathcal{P}_2$. Compositionality of the representation then requires

$$\begin{array}{rcl} \ulcorner[\mathcal{P}_1/u_1][e_1/x_1]\mathcal{P}_2\urcorner & = & [\ulcorner\mathcal{P}_1\urcorner/u_1][\ulcorner e_1\urcorner/x_1]\ulcorner\mathcal{P}_2\urcorner \\ & \equiv & (\lambda x_1{:}\mathsf{exp}.\ \lambda u_1{:}\mathsf{of}\ x_1\ \ulcorner\tau_1\urcorner.\ \ulcorner\mathcal{P}_2\urcorner)\ \ulcorner e_1\urcorner\ \ulcorner\mathcal{P}_1\urcorner \end{array}$$

After appropriate generalization, this is proved by a straightforward induction over the structure of $\mathcal{P}_2$, just as the substitution lemma for typing derivation which lies at the heart of this property.

Theorem 5.9 (Adequacy) *There is a bijection between deductions*

$$\begin{array}{c} \mathcal{P} \\ x_1{:}\tau_1, \ldots, x_n{:}\tau_n \triangleright e : \tau \end{array}$$

and LF objects P such that

$$x_1{:}\mathsf{exp}, u_1{:}\mathsf{of}\ x_1\ \ulcorner\tau_1\urcorner, \ldots, x_n{:}\mathsf{exp}, u_n{:}\mathsf{of}\ x_n\ \ulcorner\tau_n\urcorner \vdash_{TD} P \Uparrow \mathsf{of}\ \ulcorner e\urcorner\ \ulcorner\tau\urcorner.$$

The bijection is compositional in the sense that for an expression e_i and deduction $\mathcal{P}_i :: \Delta_i \triangleright e_i : \tau_i$ we have

$$\ulcorner[\mathcal{P}_i/u_i][e_i/x_i]\mathcal{P}\urcorner = [\ulcorner\mathcal{P}_i\urcorner/u_i][\ulcorner e_i\urcorner/x_i]\ulcorner\mathcal{P}\urcorner$$

where $\Delta_i = x_1{:}\mathsf{exp}, u_1{:}\mathsf{of}\ x_1\ \ulcorner\tau_1\urcorner, \ldots, x_i{:}\mathsf{exp}, u_i{:}\mathsf{of}\ x_i\ \ulcorner\tau_i\urcorner$.

Proof: As usual, by induction on the given derivations in each direction combined with verifying that correctness of the inverse of the representation function. Compositionality follows by induction on the structure of $\mathcal{P}$. □

5.5 An Elf Program for Mini-ML Type Inference

We now complete the signature from the previous section by transcribing the rules from the previous section and Section 2.5 into Elf. The notation will be suggestive of a reading of this signature as a program for type inference. First, the declarations of Mini-ML types.

```
tp : type.  %name tp T.

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

Next, the typing rules.

```
of : exp -> tp -> type.  %name of P u.
%mode of +E *T.

% Natural Numbers
tp_z     : of z nat.
tp_s     : of (s E) nat
            <- of E nat.
tp_case  : of (case E1 E2 E3) T
            <- of E1 nat
            <- of E2 T
            <- ({x:exp} of x nat -> of (E3 x) T).

% Pairs
tp_pair : of (pair E1 E2) (cross T1 T2)
```

```
           <- of E1 T1
           <- of E2 T2.
tp_fst  : of (fst E) T1
           <- of E (cross T1 T2).
tp_snd  : of (snd E) T2
           <- of E (cross T1 T2).

% Functions
tp_lam : of (lam E) (arrow T1 T2)
          <- ({x:exp} of x T1 -> of (E x) T2).
tp_app : of (app E1 E2) T1
          <- of E1 (arrow T2 T1)
          <- of E2 T2.

% Definitions
tp_letv : of (letv E1 E2) T2
           <- of E1 T1
           <- ({x:exp} of x T1 -> of (E2 x) T2).
tp_letn : of (letn E1 E2) T2
           <- of E1 T1
           <- of (E2 E1) T2.

% Recursion
tp_fix : of (fix E) T
          <- ({x:exp} of x T -> of (E x) T).
```

As for evaluation, we take advantage of compositionality in order to represent substitution of an expression for a bound variable in representation of tp_letn,

$$\frac{\Delta \triangleright e_1 : \tau_1 \qquad \Delta \triangleright [e_1/x]e_2 : \tau_2}{\Delta \triangleright \textbf{let name}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\,\textsf{tp\_letn}.$$

Since we are using higher-order abstract syntax, e_2 is represented together with its bound variable as a function of type exp $\to$ exp. Applying this function to the representation of e_1 yields the representation of $[e_1/x]e_2$.

The Elf declarations above are suggestive of an operational interpretation as a program for type inference. The idea is to pose queries of the form `?- of` $\ulcorner e \urcorner$ `T.` where `T` is a free variable subject to instantiation and e is a concrete Mini-ML expression. We begin by considering a simple example: **lam** $x.\ \langle x, \mathbf{s}\ x\rangle$. For this purpose we assume that `of` has been declared dynamic and `exp` and `tp` are static. This means that free variables of type `exp` and `tp` may appear in an answer.

```
?- of (lam [x] pair x (s x)) T.
```

```
Resolved with clause tp_lam
?- {x:exp} of x T1 -> of (pair x (s x)) T2.
```

In order to perform this first resolution step, the interpreter performed the substitutions

```
E = [x:exp] pair x (s x),
T1 = T1,
T2 = T2,
T = arrow T1 T2.
```

where `E`, `T1`, and `T2` come from the clause `tp_lam`, and `T` appears in the original query. Now the interpreter applies the rules for solving goals of functional type and introduces a new parameter `x`.

```
Introducing new parameter x : exp

x : exp
?- of x T1 -> of (pair x (s x)) T2.
Introducing new parameter u : of x T1.

x : exp,
u : of x T1
?- of (pair x (s x)) T2.
Resolved with clause tp_pair
```

This last resolution again requires some instantiation. We have

```
E1 = x,
E2 = (s x),
T2 = cross T21 T22.
```

Here, `E1`, `E2`, `T21`, and `T22` come from the clause (the latter two renamed from `T1` and `T2`, respectively). Now we have to solve two subgoals, namely `?- of x T21.` and `?- of (s x) T22.` The first subgoal immediately succeeds by using the assumption `u`, which requires the instantiation `T21 = T1` (or vice versa).

```
x : exp,
u : of x T1
?- of x T21.
Resolved with clause u
```

Here is the remainder of the computation.

```
x : exp,
u : of x T1
?- of (s x) T22.
Resolved with clause tp_s
```

This instantiates T22 to nat and produces one subgoal.

```
x : exp,
u : of x T1
?- of x nat.
Resolved with clause u
```

This last step instantiates T1 (and thereby indirectly T21) to nat. Thus we obtain the final answer

```
T = arrow nat (cross nat nat).
```

We can also ask for the typing derivation Q:

```
?- Q : of (lam [x] pair x (s x)) T.

T = arrow nat (cross nat nat),
Q = tp_lam [x:exp] [P:of x nat] tp_pair (tp_s P) P.
```

There will always be at most one answer to a type query, since for each expression constructor there exists at most one applicable clause. Of course, type inference will fail for ill-typed queries, and it will report failure, again because the rules are syntax-directed. We have stated above that there will be at most one answer yet we also know that types of expressions such as **lam** x. x are not unique. This apparent contradiction is resolved by noting that the given answer subsumes all others in the sense that all other types will be instances of the given type. This deep property of Mini-ML type inference is called the *principal type property*.

```
?- of (lam [x] x) T.
Resolved with clause tp_lam
?- {x:exp} of x T1 -> of x T2.
Introducing new parameter x
?- of x T1 -> of x T2.
Assuming u1 : of x T1
?- of x T2.
Resolved with clause u1 [with T2 = T1]

T = arrow T1 T1.
```

Here the final answer contains a free variable T1 of type tp. This is legal, since we have declare tp to be a static type. Any instance of the final answer will yield an answer to the original problem and an object of the requested type. This can be expressed by stating that search has constructed a closed object, namely

```
([T1:tp] tp_lam ([x:exp] [P:of x T1] P)) :
{T1:tp} of (lam ([x:exp] x)) (arrow T1 T1).
```

If we interpret this result as a deduction, we see that search has constructed a deduction of a parametric judgment, namely that $\triangleright$ **lam** $x.\ x : \tau_1 \to \tau_1$ for any concrete type τ_1. In order to include such generic derivations we permitted type variables α in our language. The most general or principal derivation above would then be written (in two different notations):

$$\dfrac{\dfrac{}{\triangleright x : \alpha}\,u}{\triangleright\ \mathbf{lam}\ x.\ x : \alpha \to \alpha}\,\mathsf{tp\_lam}^{x,u} \qquad \dfrac{\dfrac{}{x{:}\alpha \triangleright x : \alpha}\,\mathsf{tp\_var}}{\triangleright\ \mathbf{lam}\ x.\ x : \alpha \to \alpha}\,\mathsf{tp\_lam}$$

From the program above one can see, that the type inference problem has been reduced to the satisfiability of some equations which arise from the question if a clause head and the goal have a common instance. For example, the goal `?- of (lam [x] x) (cross T1 T2).` will fail immediately, since the only possible rule, `tp_lam`, is not applicable because `arrow T1 T2` and `cross T1 T2` do not have a common instance. The algorithm for finding common instances which also has the additional property that it does not make any unnecessary instantiation is called a *unification algorithm.* For first-order terms (such as LF objects of type `tp` in the type inference problem) a least committed common instance can always be found and is unique (modulo renaming of variables). When variables are allowed to range over functions, this is no longer the case. For example, consider the objects `E2 z` and `pair z z`, where `E2` is a free variable of type `exp -> exp`. Then there are four canonical closed solutions for `E2`:[1]

```
E2 = [x:exp] pair x x ;
E2 = [x:exp] pair z x ;
E2 = [x:exp] pair x z ;
E2 = [x:exp] pair z z.
```

In general, the question whether two objects have a common instance in the LF type theory is undecidable. This follows from the same result (due to Goldfarb [Gol81]) for a much weaker theory, the second-order fragment of the simply-typed lambda-calculus.

The operational reading of LF we sketched so far thus faces a difficulty: one of the basic steps (finding a common instance) is an undecidable problem, and, moreover, may not have a least committed solution. We deal with this problem by approximation: Elf employs an algorithm which finds a greatest common instance or detects failure in many cases and postpones other equations which must be satisfied as *constraints.* In particular, it will solve all problems which are essentially first order, as they arise in the type inference program above. Thus Elf is in spirit a *constraint logic programming language*, even though in many aspects it goes beyond the definition of the CLP family of languages described by Jaffar and Lassez [JL87].

[1] A fifth possibility, `E2 = pair z` is not canonical and η-equivalent to the second solution.

The algorithm, originally discovered by Miller in the simply-typed λ-calculus [Mil91] has been generalized to dependent and polymorphic types in [Pfe91b]. The precise manner in which it is employed in Elf is described in [Pfe91a].[2]. Here we content ourselves with a simple example which illustrates how constraints may arise.

```
?- of (lam [x] x) ((F:tp -> tp) nat).

F = F.
(( arrow T1 T1 = F nat ))
```

Here the remaining constraint is enclosed within double parentheses. Any solution to this equation yields an answer to the original query. It is important to realize that this constitutes only a *conditional success*, that is, we can in general not be sure that the given constraint set is indeed be satisfiable. In the example above, this is obvious: there are infinitely many solutions of which we show two.

```
F = [T:tp] arrow T1 T1,
T1 = T1 ;
F = [T:tp] arrow (arrow T T) (arrow T T),
T1 = arrow nat nat.
```

The same algorithm is also employed during Elf's term reconstruction phase. In practice this means that Elf term reconstruction may also terminate with remaining constraints which, in this case, is considered an error and accompanied by a request to the programmer to supply more type information.

The operational behavior of the program above may not be satisfactory from the point of view of efficiency, since expressions bound to a variable by a **let name** are type-checked once for each occurrence of the variable in the body of the expression. The following is an example for a derivation involving **let name** in Elf.

```
?- Q : of (letn (lam [y] y) ([f] pair (app f z) (app f (pair z z)))) T.
Solving...

T = cross nat (cross nat nat),
Q =
   tp_letn
      (tp_pair
          (tp_app (tp_pair tp_z tp_z)
              (tp_lam [x:exp] [P:of x (cross nat nat)] P))
          (tp_app tp_z (tp_lam [x:exp] [P:of x nat] P)))
      (tp_lam [x:exp] [P:of x T1] P).
More? y
no more solutions
```

[2] *[further discussion of unification elsewhere?]*

Notice the two occurrences of `tp_lam` which means that `(lam [y] y)` was type-checked twice. Usually, ML's type system is defined with explicit constructors for polymorphic types so that we can express $\triangleright$ **lam** $x.\ x : \forall t.\ t \rightarrow t$. The type inference algorithm can then instantiate such a most general type in the body e_2 of a **let name**-expression **let name** $x = e_1$ **in** e_2 without type-checking e_1 again. This is the essence of Milner's algorithm W in [Mil78]. It is difficult to realize this algorithm directly in Elf. Some further discussion and avenues towards a possible solution are given in [Har90], [DP91], and [Lia95]. . Theoretically, however, algorithm W of [Mil78] is not more efficient compared to the algorithm presented above as shown by [Mai92].

5.6 Representing the Proof of Type Preservation

We now return to the proof of type preservation from Section 2.6. In order to prepare for its representation in Elf, we reformulate the theorem to explicitly mention the deductions involved.

For any e, v, τ, $\mathcal{D} :: e \hookrightarrow v$, and $\mathcal{P} :: \triangleright\ e : \tau$ there exists a $\mathcal{Q} :: \triangleright\ v : \tau$.

The proof is by induction on the structure of $\mathcal{D}$ and relies heavily on inversion to predict the shape of $\mathcal{P}$ from the structure of e. The techniques from Section 3.7 suggest casting this proof as a higher-level judgment relating $\mathcal{D}$, $\mathcal{P}$, and $\mathcal{Q}$. This higher-level judgment can be represented in LF and then be implemented in Elf as a type family. We forego the intermediate step and directly map the informal proof into Elf, calling the type family `tps`.

```
tps : eval E V -> of E T -> of V T -> type.
%mode tps +D +P -Q.
```

All of the cases in the induction proof now have a direct representation in Elf. The interesting cases involve appeals to the substitution lemma (Lemma 2.4).

Case: $\mathcal{D} = \dfrac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\mathsf{ev\_z}$.

Then we have to show that for any type τ such that $\triangleright\ \mathbf{z} : \tau$ is derivable, $\triangleright\ \mathbf{z} : \tau$ is derivable. This is obvious.

There are actually two slightly different, but equivalent realizations of this case. The first uses the deduction $\mathcal{P} :: \triangleright\ \mathbf{z} : \tau$ that exists by assumption.

```
tps_z0 : tps (ev_z) P P.
```

The second, which we prefer, uses inversion to conclude that $\mathcal{P}$ must be tp_z, since it is the only rule which assigns a type to **z**.

```
tps_z     : tps (ev_z) (tp_z) (tp_z).
```

The appeal to the inversion principle is implicit in these declarations. For each $\mathcal{D}$ and $\mathcal{P}$ there should be a $\mathcal{Q}$ such that `tps` $\ulcorner\mathcal{D}\urcorner\ \ulcorner\mathcal{P}\urcorner\ \ulcorner\mathcal{Q}\urcorner$ is inhabited. The declaration above appears to work only for the case where the second argument $\mathcal{P}$ is the axiom `tp_z`. But by inversion we know that this is the only possible case. This pattern of reasoning is applied frequently when representing proofs of meta-theorems.

The next case deals with the successor constructor for natural numbers. We have taken the liberty of giving names to the deductions whose existence is shown in the proof in Section 2.6. This will help use to connect the informal statement with its implementation in Elf.

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

$\mathcal{P} :: \triangleright\ \mathbf{s}\ e_1 : \tau$	By assumption
$\mathcal{P}_1 :: \triangleright\ e_1 : \mathbf{nat}$ and $\tau = \mathbf{nat}$	By inversion
$\mathcal{Q}_1 :: \triangleright\ v_1 : \mathbf{nat}$	By ind. hyp. on $\mathcal{D}_1$
$\mathcal{Q} :: \triangleright\ \mathbf{s}\ v_1 : \mathbf{nat}$	By rule `tp_s`

Recall that an appeal to the induction hypothesis is modelled by a recursive call in the program which implements the proof. Here, the induction hypothesis is applied to $\mathcal{D}_1 :: e_1 \hookrightarrow v_1$ and $\mathcal{P}_1 :: \triangleright\ e_1 : \mathbf{nat}$ to conclude that there is a $\mathcal{Q}_1 :: \triangleright\ v_1 : \mathbf{nat}$. This is what we needed to show and can thus be directly returned.

```
tps_s     : tps (ev_s D1) (tp_s P1) (tp_s Q1)
             <- tps D1 P1 Q1.
```

We return to the cases involving **case**-expressions later after we have discussed the case for functions. The rules for pairs are straightforward.

```
tps_pair : tps (ev_pair D2 D1) (tp_pair P2 P1) (tp_pair Q2 Q1)
              <- tps D1 P1 Q1
              <- tps D2 P2 Q2.
tps_fst  : tps (ev_fst D1) (tp_fst P1) Q1
              <- tps D1 P1 (tp_pair Q2 Q1).
tps_snd  : tps (ev_snd D1) (tp_snd P1) Q2
              <- tps D1 P1 (tp_pair Q2 Q1).
```

There is an important phenomenon one should note here. Since we used the backwards arrow notation in the declarations for `ev_pair` and `tp_pair`

```
ev_pair : eval (pair E1 E2) (pair V1 V2)
             <- eval E1 V1
             <- eval E2 V2.

tp_pair : of (pair E1 E2) (cross T1 T2)
           <- of E1 T1
           <- of E2 T2.
```

their arguments are reversed from what one might expect. This is why we called the first argument to `ev_pair` above `D2` and the second argument `D1`, and similiary for `tp_pair`. The case for **lam**-expressions is simple, since they evaluate to themselves. For stylistic reasons we apply inversion here as in all other cases.

```
tps_lam     : tps (ev_lam) (tp_lam P) (tp_lam P).
```

The case for evaluating an application $e_1\ e_2$ is more complicated than the cases above. The informal proof appeals to the substitution lemma.

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

$\mathcal{P} :: \triangleright\ e_1\ e_2 : \tau_1$	By assumption
$\mathcal{P}_1 :: \triangleright\ e_1 : \tau_2 \to \tau_1$ and $\mathcal{P}_2 :: \triangleright\ e_2 : \tau_2$ for some τ_2	By inversion
$\mathcal{Q}_1 :: \triangleright\ \textbf{lam}\ x.\ e_1' : \tau_2 \to \tau_1$	By ind. hyp. on $\mathcal{D}_1$
$\mathcal{Q}_1' :: x{:}\tau_2 \triangleright\ e_1' : \tau_1$	By inversion
$\mathcal{Q}_2 :: \triangleright\ v_2 : \tau_2$	By ind. hyp. on $\mathcal{D}_2$
$\mathcal{P}_3 :: \triangleright\ [v_2/x]e_1' : \tau_1$	By the Substitution Lemma 2.4
$\mathcal{Q}_3 :: \triangleright\ v : \tau_1$	By ind. hyp. on $\mathcal{D}_3$

We repeat the declarations of the `ev_app` and `tp_lam` clauses here with some variables renamed in order to simplify the correspondence to the names used above.

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

tp_lam : of (lam E1') (arrow T2 T1)
          <- ({x:exp} of x T2 -> of (E1' x) T1).
```

The deduction $\mathcal{Q}_1$ is a deduction of a parametric and hypothetical judgment (parametric in x, hypothetical in $\triangleright\ x{:}\tau_2$). In Elf this is represented as a function which, when applied to `V2` and an object `Q2 : of V2 T2` yields an object of type `of (E1' V2) T1`, that is

```
Q1' : {x:exp} of x T2 -> of (E1' x) T1.
```

The Elf variable `E1' : exp -> exp` represents λx:exp. $\ulcorner e_1' \urcorner$, and `V2` represents v_2. Thus the appeal to the substitution lemma has been transformed into a function application using `Q1'`, that is, `P3 = Q1' V2 Q2`.

```
tps_app : tps (ev_app D3 D2 D1) (tp_app P2 P1) Q3
           <- tps D1 P1 (tp_lam Q1')
           <- tps D2 P2 Q2
           <- tps D3 (Q1' V2 Q2) Q3.
```

This may seem like black magic—where did the appeal to the substitution lemma go? The answer is that it is hidden in the proof of the adequacy theorem for the representation of typing derivations (Theorem 5.9) combined with the substitution lemma for LF itself! We have thus factored the proof effort: in the proof of the adequacy theorem, we establish that the typing judgment employs parametric and hypothetical judgments (which permit weakening and substitution). The implementation above can then take this for granted and model an appeal to the substitution lemma simply by function application.

One very nice property is the conciseness of the representation of the proofs of the meta-theorems in this fashion. Each case in the induction proof is represented directly as a clause, avoiding explicit formulation and proof of many properties of substitution, variable occurrences, *etc.* This is due to the principles of higher-order abstract syntax, judgments-as-types, and hypothetical judgments as functions. Another important factor is the Elf type reconstruction algorithm which eliminates the need for much redundant information. In the clause above, for example, we need to refer explicitly to only one expression (the variable `V2`). All other constraints imposed on applications of inferences rules can be inferred in a most general way in all of these examples. To illustrate this point, here is the fully explicit form of the above declaration, as generated by Elf's term reconstruction.

```
tps_app :
  {E:exp -> exp} {V2:exp} {E1:exp} {T:tp} {D3:eval (E V2) E1}
     {T1:tp} {Q1':{E1:exp} of E1 T1 -> of (E E1) T} {Q2:of V2 T1}
     {Q3:of E1 T} {E2:exp} {D2:eval E2 V2} {P2:of E2 T1} {E3:exp}
     {D1:eval E3 (lam E)} {P1:of E3 (arrow T1 T)}
     tps (E V2) E1 T D3 (Q1' V2 Q2) Q3 -> tps E2 V2 T1 D2 P2 Q2
        -> tps E3 (lam E) (arrow T1 T) D1 P1 (tp_lam T1 E T Q1')
        -> tps (app E3 E2) E1 T (ev_app E V2 E1 E2 E3 D3 D2 D1)
               (tp_app E2 T1 E3 T P2 P1) Q3.
```

We skip the two cases for **case**-expressions and only show their implementation. The techniques we need have all been introduced.

```
tps_case_z: tps (ev_case_z D2 D1) (tp_case P3 P2 P1) Q2
             <- tps D2 P2 Q2.

tps_case_s: tps (ev_case_s D3 D1) (tp_case P3 P2 P1) Q3
             <- tps D1 P1 (tp_s Q1')
             <- tps D3 (P3 V1 Q1') Q3.
```

Next, we come to the cases for definitions. For **let val**-expressions, no new considerations arise.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ e_1 \hookrightarrow v_1 & [v_1/x]e_2 \hookrightarrow v\end{array}}{\textbf{let val } x = e_1 \textbf{ in } e_2 \hookrightarrow v}\,\mathsf{ev\_letv}.$

$\mathcal{P} :: \triangleright \textbf{let val } x = e_1 \textbf{ in } e_2 : \tau$ By assumption
$\mathcal{P}_1 :: \triangleright e_1 : \tau_1$ and
$\mathcal{P}_2 :: x{:}\tau_1 \triangleright e_2 : \tau$ for some τ_1 By inversion
$\mathcal{Q}_1 :: \triangleright v_1 : \tau_1$ By ind. hyp. on $\mathcal{D}_1$
$\mathcal{P}_2' :: \triangleright [v_1/x]e_2 : \tau$ By the Substitution Lemma 2.4
$\mathcal{Q}_2 :: \triangleright v : \tau$ By ind. hyp. on $\mathcal{D}_2$

```
tps_letv : tps (ev_letv D2 D1) (tp_letv P2 P1) Q2
            <- tps D1 P1 Q1
            <- tps D2 (P2 V1 Q1) Q2.
```

let name-expressions may at first sight appear to be the most complicated case. However, the substitution at the level of expressions is dealt with via compositionality as in evaluation, so the representation of this case is actually quite simple.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_2 \\ [e_1/x]e_2 \hookrightarrow v\end{array}}{\textbf{let name } x = e_1 \textbf{ in } e_2 \hookrightarrow v}\,\mathsf{ev\_letn}.$

$\mathcal{P} :: \triangleright \textbf{let name } x = e_1 \textbf{ in } e_2 : \tau$ By assumption
$\mathcal{P}_2 :: \triangleright [e_1/x]e_2 : \tau$ By inversion
$\mathcal{Q}_2 :: \triangleright v : \tau$ By ind. hyp. on $\mathcal{D}_2$

```
tps_letn : tps (ev_letn D2) (tp_letn P2 P1) Q2
            <- tps D2 P2 Q2.
```

The case of fixpoint follows the same general pattern as the case for application in that we need to appeal to the substitution lemma. The solution is analogous.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ [\mathbf{fix}\, x.\, e_1/x]e_1 \hookrightarrow v\end{array}}{\mathbf{fix}\, x.\, e_1 \hookrightarrow v}\,\textsf{ev\_fix}.$

$\mathcal{P} :: \triangleright\, \mathbf{fix}\, x.\, e_1 : \tau$	By assumption
$\mathcal{P}_1 :: x : \tau \triangleright e_1 : \tau$	By inversion
$\mathcal{P}_1' :: \triangleright\, [\mathbf{fix}\, x.\, e_1/x]e_1 : \tau$	By the Substitution Lemma 2.4
$\mathcal{Q}_1 :: \triangleright\, v : \tau$	By ind. hyp. on $\mathcal{D}_1$

In the representation,

```
P1 : {x:exp} of x T -> of (E1 x) T
```

and thus

```
(P1 (fix E1)) : of (fix E1) T -> of (E1 (fix E1)) T
```

and

```
(P1 (fix E1) (tp_fix P1)) : of (E1 (fix E1)) T
```

This is the representation of the deduction $\mathcal{P}_1'$, since

$$\ulcorner[\mathbf{fix}\, x.\, e_1/x]e_1\urcorner = [\ulcorner\mathbf{fix}\, x.\, e_1\urcorner/x]\ulcorner e_1\urcorner \equiv (\lambda x{:}\textsf{exp}.\, \ulcorner e_1\urcorner)\ (\textsf{fix}\ (\lambda x{:}\textsf{exp}.\, \ulcorner e_1\urcorner)).$$

```
tps_fix : tps (ev_fix D1) (tp_fix P1) Q1
            <- tps D1 (P1 (fix E1) (tp_fix P1)) Q1.
```

Here is a simple example which illustrates the use of the `tps` type family as a program. First, we abbreviate the expression

$$\mathbf{let\,name}\ f = \mathbf{lam}\ x.\ x\ \mathbf{in}\ \mathbf{let}\ g = f\ f\ \mathbf{in}\ g\ g$$

by `e0`, using the definitional mechanism of Elf. The we generate the typing derivation with a `%solve` declaration and call it `p0`. Next we evaluate `e0` and call the evaluation `d0`. Then we pose a query that will execute the proof of type preservation to generate a substitution for `Q`, the typing derivation for the value of the expression above.

```
e0 : exp = letn (lam [x] x) ([f] letn (app f f) ([g] app g g)).
%solve p0 : of e0 T.
%solve d0 : eval e0 V.
%query 1 * tps d0 p0 Q.
```

Among other information, this will print

```
Q = tp_lam ([x:exp] [u:of x T1] u).
```

Of course, this is a very indirect way to generate a typing derivation of **lam** $x.\ x$, but illustrates the computational content of the type family `tps` we defined.

5.7 Exercises

Exercise 5.1 Carry out three representative cases in the proof of Property 5.4. Where do we require the assumption that Γ must be of a certain form? Construct a counterexample which shows the falsehood of careless generalization of the theorem to admit arbitary contexts Γ.

Exercise 5.2 Carry out three representative cases in the proof of the Adequacy Theorem 5.5.

Exercise 5.3 Modify the natural semantics for Mini-ML such that only closed λ-expressions have a value. How does this affect the proof of type preservation?

Exercise 5.4 Write Elf programs

1. to count the number of occurrences of bound variables in a Mini-ML expression;
2. to remove all vacuous **let**-bindings from a Mini-ML expression;
3. to rewrite all occurrences of expressions of the form $(\mathbf{lam}\ x.\ e_2)\ e_1$ to $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$.

Exercise 5.5 For each of the following statements, prove them informally and represent the proof in Elf, or give a counterexample if the statement is false.

1. For any expressions e_1 and e_2, evaluation of $(\mathbf{lam}\ x.\ e_2)\ e_1$ yields a value v if and only if evaluation of $\mathbf{let\,val}\ x = e_1\ \mathbf{in}\ e_2$ yields v.
2. For any expressions e_1 and e_2, evaluation of $(\mathbf{lam}\ x.\ e_2)\ e_1$ yields a value v if and only if evaluation of $\mathbf{let\,name}\ x = e_1\ \mathbf{in}\ e_2$ yields v.
3. For *values* v_1, the expression $(\mathbf{lam}\ x.\ e_2)\ v_1$ has type τ if and only if the expression $\mathbf{let\,val}\ x = v_1\ \mathbf{in}\ e_2$ has type τ.
4. For *values* v_1, the expression $(\mathbf{lam}\ x.\ e_2)\ v_1$ has type τ if and only if the expression $\mathbf{let\,name}\ x = v_1\ \mathbf{in}\ e_2$ has type τ.
5. Evaluation is deterministic, that is, whenever $e \hookrightarrow v_1$ and $e \hookrightarrow v_2$ then $v_1 = v_2$ (modulo renaming of bound variables, as usual).

Exercise 5.6 Give an LF representation of the fragment of Mini-ML which includes pairing, first and second projection, functions and application, and definitions with **let val** without using hypothetical judgments. Thus the typing judgment should be represented as a ternary type family, say, hastype, indexed by a representation of the context Δ and representations of e and τ. We would then look for a representation function $\ulcorner\cdot\urcorner$ which satisfies

$$\Gamma \vdash \ulcorner\mathcal{P}\urcorner : \mathsf{hastype}\ \ulcorner\Delta\urcorner\ \ulcorner e\urcorner\ \ulcorner\tau\urcorner$$

for a suitable Γ, whenever $\mathcal{P}$ is a valid deduction of $\Delta \triangleright e : \tau$.

Exercise 5.7 Illustrate by means of an example why declaring the type `tp` as dynamic might lead to undesirable backtracking and unexpected answers during type inference for Mini-ML with the program in Section 5.5. Can you construct a situation where the program diverges on a well-typed Mini-ML expression? How about on a Mini-ML expression which is not well-typed?

Exercise 5.8 Extend the implementation of the Mini-ML interpreter, type inference, and proof of type preservation to include

1. unit, void, and disjoint sum types (see Exercise 2.7),
2. lists (see Exercise 2.8).

Exercise 5.9 Consider the call-by-name version of Mini-ML with lazy constructors as sketched in Exercise 2.13. Recall that neither the arguments to functions, nor the arguments to constructors ($\mathbf{s}$ and $\langle\cdot,\cdot\rangle$) should be evaluated.

1. Implement an interpreter for the language and show a few expressions that highlight the differences in the operational semantics.
2. Implement type inference.
3. Define and implement a suitable notion of value.
4. Prove value soundness and implement your proof.
5. Prove type preservation and implement your proof.

Discuss the main differences between the development for Mini-ML and its call-by-name variant.

Exercise 5.10 The definition of the judgment e *Closed* follows systematically from the representation of expressions in higher-order abstract syntax, because object-level variables are represented by meta-level variables. This exercise explores a generalization of this fact. Assume we have a signature Σ_0 in the simply-typed lambda-calculus that declares exactly one type constant a and some unspecified number of object constants $c_1, \ldots, c_n$. Define an LF signature Σ_1 that extends Σ_0 by a new family

$$\mathsf{closed} \quad : \quad a \to \mathsf{type}$$

such that

$$\vdash_{\Sigma_0} N : a$$

if and only if

$$\Gamma \vdash_{\Sigma_1} N : a \quad \text{and} \quad \Gamma \vdash_{\Sigma_1} M : \mathsf{closed}\ N$$

provided Γ no has declaration of the form $u{:}\Pi y_1{:}A_1 \ldots \Pi y_m{:}A_m.\ \mathsf{closed}\ P$.

Exercise 5.11 Write Elf programs to determine if a Mini-ML expression is free of the recursion operation **fix** and at the same time

1. linear (every bound variable occurs exactly once);
2. affine (every bound variable occurs at most once);
3. relevant (every bound variable occurs at least once).

Since only one branch in a case statement will be taken during evaluation, a bound variable must occur exactly once in each branch in a linear expression, may occur at most once in each branch in an affine expression, and must occur at least once in each branch in a relevant expression.

Exercise 5.12 Instead of substituting in the typing rule for **let name**-expressions we could extend contexts to record the definitions for variables bound with **let name**.

$$\text{Contexts} \quad \Delta \quad ::= \quad \cdot \mid \Delta, x{:}\tau \mid \Delta, x = e$$

Variables must still occur at most once in a context (no variable may be declared and defined). We would replace the rule tp_letn by the following two rules.

$$\frac{\Delta \triangleright e_1 : \tau_1 \qquad \Delta, x = e_1 \triangleright e_2 : \tau_1}{\Delta \triangleright \textbf{let name}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\,\mathsf{tp\_letn0} \qquad \frac{x = e \text{ in } \Delta \qquad \Delta \triangleright e : \tau}{\Delta \triangleright x : \tau}\,\mathsf{tp\_var0}$$

There are at least two ways we can view this modification for representation in the framework.

1. We use a new judgment, $x = e$, which is introduced only as a hypothesis into a derivation.
2. We view a hypothesis $x = e$ as the assumption of an *inference rule*. We might write this as

$$\frac{\triangleright e_1 : \tau_1 \qquad \begin{array}{c} \dfrac{\triangleright e_1 : \tau}{\triangleright x : \tau}\,u_\tau \\ \vdots \\ \triangleright e_2 : \tau_2 \end{array}}{\triangleright \textbf{let name}\ x = e_1\ \textbf{in}\ e_2}\,\mathsf{tp\_let}^{x,u}.$$

The subscript τ in the hypothetical rule u indicates that in each application of u we may choose a different type τ. Hypothetical rules have been investigated by Schroeder-Heister [SH84].

1. Show the proper generalization and the new cases in the informal proof of type preservation using rules tp_letn0 and tp_var0.
2. Give the Elf implementation of type inference using alternative 1.
3. Implement the modified proof of type preservation in Elf using alternative 1.
4. Give the Elf implementation of type inference using alternative 2.
5. Implement the modified proof of type preservation in Elf using alternative 2.

Exercise 5.13 [*on the value restriction or its absence*]

Exercise 5.14 [*on interpreting let name as let value, connect to value restriction*]

Exercise 5.15 The typing rules for Mini-ML in Section 2.5 are not a realistic basis for an implementation, since they require e_1 in an expression of the form **let name** $u = e_1$ **in** e_2 to be re-checked at every occurrence of u in e_2. This is because we may need to assign different types to e_1 for different occurrences of u.

Fortunately, all the different types for an expression e can be seen as instances of a most general *type schema* for e. In this exercise we explore an alternative formulation of Mini-ML which uses explicit type schemas.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \alpha \\ \text{Type Schemas} & \sigma & ::= & \tau \mid \forall \alpha.\, \sigma \end{array}$$

Type schemas σ are related to types τ through *instantiation*, written as $\sigma \preceq \tau$. This judgment is defined by

$$\frac{}{\tau \preceq \tau}\,\textsf{inst\_tp} \qquad \frac{[\tau'/\alpha]\sigma \preceq \tau}{\forall \alpha.\, \sigma \preceq \tau}\,\textsf{inst\_all}.$$

We modify the judgment $\Delta \triangleright e : \tau$ and add a second judgment, $\Delta \triangleright\!\!\triangleright e : \sigma$ stating that e has type schema σ. The typing rule for **let name** now no longer employs substitution, but refers to a schematic type for the definition. It must therefore be possible to assign type schemas to variables which are instantiated when we need an actual type for a variable.

$$\frac{\Delta \triangleright\!\!\triangleright e_1 : \sigma_1 \qquad \Delta, x{:}\sigma_1 \triangleright e_2 : \tau_2}{\Delta \triangleright \mathbf{let\,name}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\,\textsf{tp\_letn} \qquad \frac{\Delta(x) = \sigma \qquad \sigma \preceq \tau}{\Delta \triangleright x : \tau}\,\textsf{tp\_var}$$

Type schemas can be derived for expressions by means of quantifying over free type variables.

$$\frac{\Delta \rhd e : \tau}{\Delta \mathrel{\rhd\!\!\rhd} e : \tau}\,\textsf{tpsc\_tp} \qquad \frac{\Delta \mathrel{\rhd\!\!\rhd} e : \sigma}{\Delta \mathrel{\rhd\!\!\rhd} e : \forall\alpha.\,\sigma}\,\textsf{tpsc\_all}^{\alpha}$$

Here the premiss of the $\textsf{tpsc\_all}^{\alpha}$ rule must be parametric in α, that is, α must not occur free in the context Δ.

In the proofs and implementations below you may restrict yourself to the fragment of the language with functions and **let name**, since the changes are orthogonal to the other constructs of the language.

1. Give an example which shows why the restriction on the $\textsf{tpsc\_all}$ rule is necessary.

2. Prove type preservation for this formulation of Mini-ML. Carefully write out and prove any substitution lemmas you might need, but you may take weakening and exchange for granted.

3. State the theorem which asserts the equivalence of the new typing rules when compared to the formulation in Section 2.5.

4. Prove the easy direction of the theorem in item 3. Can you conjecture the critical lemma for the opposite direction?

5. Implement type schemas, schematic instantiation, and the new typing judgments in Elf.

6. Unlike our first implementation, the new typing rules do not directly provide an implementation of type inference for Mini-ML in Elf. Show the difficulty by means of an example.

7. Implement the proof of type preservation from item 2 in Elf.

8. Implement one direction of the equivalence proof from item 3 in Elf.

Exercise 5.16 [*about the Milner-Mycroft calculus with explicit types for polymorphic let and recursion*]

Chapter 6

Compilation

The model of evaluation introduced in Section 2.3 and formalized in Section 3.6 builds only on the expressions of the Mini-ML language itself. This leads very naturally to an *interpreter* in Elf which is given in Section 4.3. Our specification of the operational semantics is in the style of *natural semantics* which very often lends itself to direct, though inefficient, execution. The inefficiency of the interpreter in 4.3 is more than just a practical issue, since it is clearly the wrong model if we would like to reason about the complexity of functions defined in Mini-ML. One can refine the evaluation model in two ways: one is to consider more efficient interpreters (see Exercises 2.12 and 4.2), another is to consider compilation. In this chapter we pursue the latter possibility and describe and prove the correctness of a compiler for Mini-ML.

In order to define a compiler we need a *target language* for compilation, that is, the language into which programs in the source language are translated. This target language has its own operational semantics, and we must show the correctness of compilation with respect to these two languages and their semantics. The ultimate target language for compilation is determined by the architecture and instruction set of the machine the programs are to be run on. In order to insulate compilers from the details of particular machine architectures it is advisable to design an intermediate language and execution model which is influenced by a set of target architectures and by constructs of the source language. We refer to this intermediate level as an *abstract machine*. Abstract machine code can then itself either be interpreted or compiled further to actual machine code. In this chapter we take a stepwise approach to compilation, using two intermediate forms between Mini-ML and a variant of the SECD machine [Lan64] which is also related to the Categorical Abstract Machine (CAM) [CCM87]. This decomposition simplifies the correctness proofs and localizes ideas which are necessary to understand the compiler in its totality.

The material presented in this chapter follows work by Hannan [HM90, Han91], both in general approach and in many details. An extended abstract that also addresses correctness issues and methods of formalization can be found in [HP92]. A different approach to compilation using *continuations* may be found in Section **??**.

6.1 An Environment Model for Evaluation

The evaluation judgment $e \hookrightarrow v$ requires that all information about the state of the computation is contained in the Mini-ML expression e. The application of a function formed by λ-abstraction, **lam** $x.\ e$, to an argument v thus requires the substitution of v for x in e and evaluation of the result. In order to avoid this substitution it may seem reasonable to formulate evaluation as a hypothetical judgment (e is evaluated under the hypothesis that x evaluates to v) but this attempt fails (see Exercise 6.1). Instead, we allow free variables in expressions which are given values in an *environment*, which is explicitly represented as part of a revised evaluation judgment. Variables are evaluated by looking up their value in the environment; previously we always eliminated them by substitution, so no separate rule was required. However, this leads to a problem with the scope of variables. Consider the expression **lam** $y.\ x$ in an environment that binds x to **z**. According to our natural semantics the value of this expression should be **lam** $y.$ **z**, but this requires the substitution of **z** for x. Simply returning **lam** $y.\ x$ is incorrect if this value may later be interpreted in an environment in which x is not bound, or bound to a different value. The practical solution is to return a *closure* consisting of an environment η and an expression **lam** $y.\ e$. η must contain at least all the variables free in **lam** $y.\ e$. We ignore certain questions of efficiency in our presentation and simply pair up the complete current environment with the expression to form the closure.

This approach leads to the question how to represent environments and closures. A simple solution is to represent an environment as a list of values and a variable as a pointer into this list. It was de Bruijn's idea [dB72] to implement such pointers as natural numbers where n refers to the n^{th} element of the environment list. This works smoothly if we also represent bound variables in this fashion: an occurence of a bound variable points backwards to the place where it is bound. This pointer takes the form of a positive integer, where 1 refers to the innermost binder and 1 is added for every binding encountered when going upward through the expression. For example

$$\textbf{lam}\ x.\ \textbf{lam}\ y.\ x\ (\textbf{lam}\ z.\ y\ z)$$

would be written as

$$\Lambda\ (\Lambda\ (2\ (\Lambda\ (2\ 1))))$$

where Λ binds an (unnamed) variable. In this form expressions that differ only in the names of their bound variables are syntactically identical. If we restrict attention to pure λ-terms for the moment, this leads to the definition

$$\begin{array}{rrcl} \text{de Bruijn Expressions} & D & ::= & n \mid \Lambda D \mid D_1\ D_2 \\ \text{de Bruijn Indices} & n & ::= & 1 \mid 2 \mid \ldots \end{array}$$

Instead of using integers and general arithmetic operations on them, we use only the integer 1 to refer to the innermost element of the environment and the operator $\uparrow$ (read: shift, written in post-fix notation) to increment variable references. That is, the integer $n+1$ is represented as

$$1\underbrace{\uparrow\cdots\uparrow}_{n\text{ times}}.$$

But $\uparrow$ can also be applied to other expressions, in effect raising each integer in the expression by 1. For example, the expression

$$\mathbf{lam}\ x.\ \mathbf{lam}\ y.\ x\ x$$

can be represented by

$$\Lambda\ (\Lambda\ ((1\uparrow)\ (1\uparrow)))$$

or

$$\Lambda\ (\Lambda\ ((1\ 1)\uparrow)).$$

This is a very simple form of a λ-calculus with *explicit substitutions* where $\uparrow$ is the only available substitution (see [ACCL91]).

$$\text{Modified de Bruijn Expressions} \quad F \quad ::= \quad 1 \mid F\uparrow \mid \Lambda F \mid F_1\ F_2$$

We use the convention that the postfix operator $\uparrow$ binds stronger than application which in turn binds stronger that the prefix operator Λ. Thus the two examples above can be written as $\Lambda\ \Lambda\ 1\uparrow\ 1\uparrow$ and $\Lambda\ \Lambda\ (1\ 1)\uparrow$, respectively.

The next step is to introduce environments. These depend on *values* and vice versa, since a closure is a pair of an environment and an expression, and an environment is a list of values. This can be carried to the extreme: in the Categorical Abstract Machine (CAM), for example, environments are built as iterated pairs and are thus values. Our representation will not make this identification. Since we have simplified our language to a pure λ-calculus, the only kind of value which can arise is a closure.

$$\begin{array}{rrcl} \text{Environments} & \eta & ::= & \cdot \mid \eta, W \\ \text{Values} & W & ::= & \{\eta; F\} \end{array}$$

We write w for parameters ranging over values. During the course of evaluation, only closures over Λ-expressions will arise, that is, all closures have the form $\{\eta; \Lambda F'\}$ (see Exercise 6.2).

The specification of modified de Bruijn expressions, values, and environments is straightforward. The abstract syntax is now first-order, since the language does not contain any name binding constructs.

```
exp'   : type.  %name exp' F f.

1      : exp'.
^      : exp' -> exp'.  %postfix 20 ^.
lam'   : exp' -> exp'.
app'   : exp' -> exp' -> exp'.

% Environments and values

env    : type.  %name env N.
val    : type.  %name val W w.

empty  : env.
,      : env -> val -> env.   %infix left 10 ,.

clo    : env -> exp' -> val.
```

There are two main judgments that achieve compilation: one relates a de Bruijn expression F in an environment η to an ordinary expression e, another relates a value W to an expression v. We also need an evaluation judgment relating de Bruijn expressions and values in a given environment.

$\eta \vdash F \leftrightarrow e$	F translates to e in environment η
$W \Leftrightarrow v$	W translates to v
$\eta \vdash F \hookrightarrow W$	F evaluates to W in environment η

When we evaluate a given expression e using these judgments, we translate it to a de Bruijn expression F in the empty environment, evaluate F in the empty environment to obtain a value W, and then translate W to an expression v in the original language. This is depicted in the following diagram.

$$\begin{tikzcd}
e \arrow[r, "\mathcal{D} :: e \hookrightarrow v"] \arrow[d, "\mathcal{C} :: \cdot \vdash F \leftrightarrow e"'] & v \\
F \arrow[r, "\mathcal{D}' :: \cdot \vdash F \hookrightarrow W"'] & W \arrow[u, "\mathcal{U} :: W \Leftrightarrow v"']
\end{tikzcd}$$

The correctness of this phase of compilation can then be decomposed into two statements. For *completeness*, we assume that $\mathcal{D}$ and therefore e and v are given, and we would like to show that there exist $\mathcal{C}$, $\mathcal{D}'$, and $\mathcal{U}$ completing the diagram. This means that for every evaluation of e to a value v, this value could also have been produced by evaluating the compiled expression and translating the resulting value back to the original language. The dual of this is *soundness*: we assume that

$\mathcal{C}$, $\mathcal{D}'$ and $\mathcal{U}$ are given and we have to show that an evaluation $\mathcal{D}$ exists. That is, every value which can be produced by compilation and evaluation of compiled expressions can also be produced by direct evaluation.

We will continue to restrict ourselves to expressions built up only from abstraction and application. When we generalize this later only the case of fixpoint expressions will introduce an essential complication. First we define evaluation of de Bruijn expressions in an environment η, written as $\eta \vdash F \hookrightarrow W$. The variable 1 refers to the first value in the environment (counting from right to left); its evaluation just returns that value.

$$\frac{}{\eta, W \vdash 1 \hookrightarrow W}\,\textsf{fev\_1}$$

The meaning of an expression $F\!\uparrow$ in an environment η, W is the same as the meaning of F in the environment η. Intuitively, the environment references from F into η are shifted by one. The typical case is one where a reference to the n^{th} value in η is represented by the expression $1\!\uparrow \cdots \uparrow$, where the shift operator is applied $n-1$ times.

$$\frac{\eta \vdash F \hookrightarrow W}{\eta, W' \vdash F\!\uparrow \hookrightarrow W}\,\textsf{fev\_}\!\uparrow$$

A functional abstraction usually immediately evaluates to itself. Here this is insufficient, since an expression ΛF may contain references to the environment η. Thus we need to combine the environment η with ΛF to produce a closed (and self-contained) value.

$$\frac{}{\eta \vdash \Lambda F \hookrightarrow \{\eta; \Lambda F\}}\,\textsf{fev\_lam}$$

In order to evaluate $F_1\ F_2$ in an environment η we evaluate both F_1 and F_2 in that environment, yielding the closure $\{\eta'; \Lambda F_1'\}$ and value W_2, respectively. We then add W_2 to the environment η', in effect binding the variable previously bound by Λ in $\Lambda F_1'$ to W_2 and then evaluate F_1' in the extended environment to obtain the overall value W.

$$\frac{\eta \vdash F_1 \hookrightarrow \{\eta'; \Lambda F_1'\} \qquad \eta \vdash F_2 \hookrightarrow W_2 \qquad \eta', W_2 \vdash F_1' \hookrightarrow W}{\eta \vdash F_1\ F_2 \hookrightarrow W}\,\textsf{fev\_app}$$

Here is the implementation of this judgment as the type family `feval` in Elf.

```
feval : env -> exp' -> val -> type.  %name feval D'.
%mode feval +N +F -W.

% Variables
fev_1 : feval (N , W) 1 W.
fev_^ : feval (N , W') (F ^) W
```

```
           <- feval N F W.

% Functions
fev_lam : feval N (lam' F) (clo N (lam' F)).
fev_app : feval N (app' F1 F2) W
           <- feval N F1 (clo N' (lam' F1'))
           <- feval N F2 W2
           <- feval (N' , W2) F1' W.
```

We have written this signature in a way that emphasizes its operational reading, because it serves as an implementation of an interpreter. As an example, consider the evaluation of the expression $(\Lambda\ (\Lambda\ (1\uparrow)))\ (\Lambda\ 1)$, which is a representation of $(\mathbf{lam}\ x.\ \mathbf{lam}\ y.\ x)\ (\mathbf{lam}\ v.\ v)$.

```
?- D : feval empty (app' (lam' (lam' (1 ^))) (lam' 1)) W.

W = clo (empty , clo empty (lam' 1)) (lam' (1 ^)).
D' = fev_app fev_lam fev_lam fev_lam.
```

The resulting closure, $\{(\cdot, \{\cdot, \Lambda 1\}); \Lambda(1\uparrow)\}$, represents the de Bruijn expressions $\Lambda(\Lambda 1)$, since $(1\uparrow)$ refers to the first value in the environment.

The translation between ordinary and de Bruijn expressions is specified by the following rules which employ a parametric and hypothetical judgment.

$$\frac{\eta \vdash F_1 \leftrightarrow e_1 \qquad \eta \vdash F_2 \leftrightarrow e_2}{\eta \vdash F_1\ F_2 \leftrightarrow e_1\ e_2}\,\textsf{tr\_app} \qquad \frac{\begin{array}{c}\overline{w \Leftrightarrow x}\,u \\ \vdots \\ \eta, w \vdash F \leftrightarrow e\end{array}}{\eta \vdash \Lambda F \leftrightarrow \mathbf{lam}\ x.\ e}\,\textsf{tr\_lam}^{w,x,u}$$

$$\frac{W \Leftrightarrow e}{\eta, W \vdash 1 \leftrightarrow e}\,\textsf{tr\_1} \qquad \frac{\eta \vdash F \leftrightarrow e}{\eta, W \vdash F\uparrow \leftrightarrow e}\,\textsf{tr\_}\!\uparrow$$

where the rule tr_lam is restricted to the case where w and x are new parameters not free in any other hypothesis, and u is a new label. The translation of values is defined by a single rule in this language fragment.

$$\frac{\eta \vdash \Lambda F \leftrightarrow \mathbf{lam}\ x.\ e}{\{\eta; \Lambda F\} \Leftrightarrow \mathbf{lam}\ x.\ e}\,\textsf{vtr\_lam}$$

As remarked earlier this translation can be non-deterministic if η and e are given and F is to be generated. This is the direction in which this judgment would be used for compilation. Here is an example of a translation.

$$
\cfrac{
 \cfrac{\cfrac{\cfrac{\cfrac{}{w \Leftrightarrow x}\,u}{\cdot, w \vdash 1 \leftrightarrow x}\,\mathsf{tr\_1}}{\cdot, w, w' \vdash 1\uparrow \leftrightarrow x}\,\mathsf{tr\_\uparrow}}{\cdot, w \vdash \Lambda 1\uparrow \leftrightarrow \mathbf{lam}\, y.\ x}\,\mathsf{tr\_lam}^{w,',y,u'}}{\cdot \vdash \Lambda\Lambda 1\uparrow \leftrightarrow \mathbf{lam}\, x.\ \mathbf{lam}\, y.\ x}\,\mathsf{tr\_lam}^{w,x,u}
 \quad
 \cfrac{\cfrac{\cfrac{}{w'' \Leftrightarrow v}\,u''}{\cdot, w'' \vdash 1 \leftrightarrow v}\,\mathsf{tr\_1}}{\cdot \vdash \Lambda 1 \leftrightarrow \mathbf{lam}\, v.\ v}\,\mathsf{tr\_lam}^{w'',v,u''}
}{\cdot \vdash (\Lambda\Lambda 1\uparrow)\ (\Lambda 1) \leftrightarrow (\mathbf{lam}\, x.\ \mathbf{lam}\, y.\ x)\ (\mathbf{lam}\, v.\ v)}\,\mathsf{tr\_app}
$$

The representation of the translation judgment relies on the standard technique for representing deductions of hypothetical judgments as functions.

```
trans  : env -> exp' -> exp -> type.  %name trans C.
vtrans : val -> exp -> type.          %name vtrans U.
% can be used in different directions
%mode trans +N +F -E.
%mode vtrans +W -V.

% Functions
tr_lam : trans N (lam' F) (lam E)
            <- ({w:val} {x:exp}
                  vtrans w x -> trans (N , w) F (E x)).
tr_app : trans N (app' F1 F2) (app E1 E2)
            <- trans N F1 E1
            <- trans N F2 E2.

% Variables
tr_1   : trans (N , W) 1 E
          <- vtrans W E.
tr_^   : trans (N , W) (F ^) E
          <- trans N F E.

% Values
vtr_lam : vtrans (clo N (lam' F)) (lam E)
           <- trans N (lam' F) (lam E).
```

The judgment

$$\begin{array}{c}\cfrac{}{w \Leftrightarrow x}\,u\\ \vdots \\ \eta, w \vdash F \leftrightarrow e\end{array}$$

in the premiss of the tr_lam is parametric in the variables w and x and hypothetical in u. It is represented by a function which, when given a value W', an expression e', and a deduction $\mathcal{U}' :: W' \Leftrightarrow e'$ returns a deduction $\mathcal{D}' :: \eta, W' \vdash F \leftrightarrow [e'/x]e$. This property is crucial in the proof of compiler correctness.

The signature above can be executed as a non-deterministic program for translation between de Bruijn and ordinary expressions in both directions. For the compilation of expressions it is important to keep the clauses `tr_1` and `tr_^` in the given order so as to avoid unnecessary backtracking. This non-determinism arises, since the expression `E` in the rules `tr_1` and `tr_^` does not change in the recursive calls. For other possible implementations see Exercise 6.3. Here is an execution which yields the example deduction above.

```
?- C : trans empty F (app (lam [x] lam [y] x) (lam [z] z)).

F = app' (lam' (lam' (1 ^))) (lam' 1).
C =
   tr_app (tr_lam ([w:val] [x:exp] [u3:vtrans w x] tr_1 u3))
     (tr_lam
         ([w:val] [x:exp] [u1:vtrans w x]
             tr_lam ([w1:val] [x1:exp] [u2:vtrans w1 x1]
                         tr_^ (tr_1 u1)))).
```

It is not immediately obvious that every source expression e can in fact be compiled using this judgment. This is the subject of the following theorem.

Theorem 6.1 *For every closed expression e there exists a de Bruijn expression F such that $\cdot \vdash F \leftrightarrow e$.*

Proof: A direct attempt at an induction argument fails—a typical situation when proving properties of judgments which involve hypothetical reasoning. However, the theorem follows immediately from Lemma 6.2 below. □

Lemma 6.2 *Let $w_1, \ldots, w_n$ be parameters ranging over values and let η be the environment $\cdot, w_n, \ldots, w_1$. Furthermore, let $x_1, \ldots, x_n$ range over expression variables. For any expression e with free variables among $x_1, \ldots, x_n$ there exists a de Bruijn expression F and a deduction $\mathcal{C}$ of $\eta \vdash F \leftrightarrow e$ from hypotheses $u_1 :: w_1 \Leftrightarrow x_1, \ldots, u_n :: w_n \Leftrightarrow x_n$.*

Proof: By induction on the structure of e.

Case: $e = e_1\ e_2$. By induction hypothesis on e_1 and e_2, there exist F_1 and F_2 and deductions $\mathcal{C}_1 :: \eta \vdash F_1 \leftrightarrow e_1$ and $\mathcal{C}_2 :: \eta \vdash F_2 \leftrightarrow e_2$. Applying the rule tr_app to $\mathcal{C}_1$ and $\mathcal{C}_2$ yields the desired deduction $\mathcal{C} :: \eta \vdash F_1\ F_2 \leftrightarrow e_1\ e_2$.

Case: $e = \mathbf{lam}\ x.\ e_1$. Here we apply the induction hypothesis to the expression e_1, environment η, w for a new parameter w, and hypotheses $u_1 :: w_1 \Leftrightarrow x_1, \ldots, u_n :: w_n \Leftrightarrow x_n, u :: w \Leftrightarrow x$ to obtain an F_1 and a deduction

$$\begin{array}{c} \overline{\quad w \Leftrightarrow x \quad}\, u \\ \mathcal{C}_1 \\ \eta, w \vdash F_1 \leftrightarrow e_1 \end{array}$$

possibly also using hypotheses labelled $u_1, \ldots, u_n$. Note that e_1 is an expression with free variables among $x_1, \ldots, x_n, x$. Applying the rule tr_lam discharges the hypothesis u and we obtain the desired deduction

$$\mathcal{C} = \dfrac{\begin{array}{c} \overline{\quad w \Leftrightarrow x \quad}\, u \\ \mathcal{C}_1 \\ \eta, w \vdash F_1 \leftrightarrow e_1 \end{array}}{\eta \vdash \Lambda F_1 \leftrightarrow \mathbf{lam}\ x.\ e_1}\,\mathsf{tr\_lam}^u$$

Case: $e = x$. Then $x = x_i$ for some i between 1 and n and we let $F = 1\ \underbrace{\uparrow \cdots \uparrow}_{i-1 \text{ times}}$ and

$$\mathcal{C} = \dfrac{\dfrac{\dfrac{\overline{\quad w_i \Leftrightarrow x_i \quad}\, u_i}{\cdot, w_n, \ldots, w_i \vdash 1 \leftrightarrow x_i}\,\mathsf{tr\_1}}{\cdots}\,\mathsf{tr\_}\!\uparrow}{\cdot, w_n, \ldots, w_1 \vdash 1\!\uparrow \cdots \uparrow \leftrightarrow x_i}\,\mathsf{tr\_}\!\uparrow$$

□

This proof cannot be represented directly in Elf because we cannot employ the usual technique for representing hypothetical judgments as functions. The difficulty is that the order of the hypotheses is important for returning the correct variable $1\!\uparrow \cdots \uparrow$, but hypothetical judgments are generally invariant under reordering of hypotheses. Hannan [Han91] has suggested a different, deterministic translation for which termination is relatively easy to show, but which complicates the proofs of the remaining properties of compiler correctness. Thus our formalization does not capture the desirable property that compilation always terminates. All the remaining parts, however, are implemented. The first property states that translation followed by evaluation leads to the same result as evaluation followed by translation. We generalize this for arbitrary environments η in order to allow a proof by induction.

This property is depicted in the following diagram.

$$\begin{tikzcd}
e \arrow[r, "\mathcal{D} :: e \hookrightarrow v"] \arrow[d, "\mathcal{C} :: \eta \vdash F \leftrightarrow e"'] & v \\
\eta; F \arrow[r, dotted, "\mathcal{D}' :: \eta \vdash F \hookrightarrow W"'] & W \arrow[u, dotted, "\mathcal{U} :: W \Leftrightarrow v"']
\end{tikzcd}$$

The solid lines indicate deductions that are assumed, dotted lines represent the deductions whose existence we assert and prove below.

Lemma 6.3 *For any closed expressions e and v, environment η, de Bruijn expression F, deductions $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, there exist a value W and deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: By induction on the structures of $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$. In this induction we assume the induction hypothesis on the premisses of $\mathcal{D}$ and for arbitrary $\mathcal{C}$ and on the premisses of $\mathcal{C}$, but for the same $\mathcal{D}$. This is sometimes called *lexicographic* induction on the pair consisting of $\mathcal{D}$ and $\mathcal{C}$. It should be intuitively clear that this form of induction is valid. We represent this proof as a judgment relating the four deductions involved in the diagram.

```
map_eval : eval E V -> trans N F E
              -> feval N F W -> vtrans W V -> type.
%mode map_eval +D +C -D' -U.
```

Case: $\mathcal{C}$ ends in an application of the tr_1 rule.

$$\mathcal{C} = \frac{\begin{array}{c}\mathcal{U}_1\\ W_1 \Leftrightarrow e\end{array}}{\eta_1, W_1 \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1}$$

$\mathcal{D} :: e \hookrightarrow v$	Assumption
$\mathcal{C}_1 :: \eta_1' \vdash \Lambda F_1' \leftrightarrow e$ and $W_1 = \{\eta_1'; \Lambda F_1'\}$	By inversion on $\mathcal{U}_1$
$e = \textbf{lam}\ x.\ e_1$	By inversion on $\mathcal{C}_1$
$v = \textbf{lam}\ x.\ e_1 = e$	By inversion on $\mathcal{D}$

Then $W = W_1$, $\mathcal{U} = \mathcal{U}_1 :: W_1 \Leftrightarrow e$ and $\mathcal{D}' = \mathsf{fev\_1} :: \eta_1, W_1 \vdash 1 \hookrightarrow W_1$ satisfy the requirements of the theorem. This case is captured in the clause

```
mp_1 : map_eval (ev_lam) (tr_1 (vtr_lam (tr_lam C2)))
                (fev_1) (vtr_lam (tr_lam C2)).
```

Case: $\mathcal{C}$ ends in an application of the $\mathsf{tr\_}{\uparrow}$ rule.

$$\mathcal{C} = \dfrac{\begin{matrix}\mathcal{C}_1 \\ \eta_1 \vdash F_1 \leftrightarrow e\end{matrix}}{\eta_1, W_1' \vdash F_1{\uparrow} \leftrightarrow e}\ \mathsf{tr\_}{\uparrow}$$

$\mathcal{D} :: e \hookrightarrow v$	Assumption
$\mathcal{D}_1' :: \eta_1 \vdash F_1 \hookrightarrow W_1$	
and $\mathcal{U}_1 :: W_1 \Leftrightarrow v$	By ind. hyp. on $\mathcal{D}$ and $\mathcal{C}_1$

Now we let $W = W_1, \mathcal{U} = \mathcal{U}_1$, and obtain $\mathcal{D}' :: \eta_1, W_1' \vdash F_1{\uparrow} \hookrightarrow W_1$ by $\mathsf{fev\_}{\uparrow}$ from $\mathcal{D}_1'$.

```
mp_^ : map_eval D (tr_^ C1) (fev_^ D1') U1
          <- map_eval D C1 D1' U1.
```

For the remaining cases we assume that the previous two cases do not apply. We refer to this assumption as *exclusion*.

Case: $\mathcal{D}$ ends in an application of the $\mathsf{ev\_lam}$ rule.

$$\mathcal{D} = \dfrac{}{\mathbf{lam}\ x.\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1}\ \mathsf{ev\_lam}$$

$\mathcal{C} :: \eta \vdash F \leftrightarrow \mathbf{lam}\ x.\ e_1$	By assumption
$F = \Lambda F_1$	By inversion and exclusion

Then we let $W = \{\eta; \Lambda F_1\}$, $\mathcal{D}' = \mathsf{fev\_lam} :: \eta \vdash \Lambda F_1 \hookrightarrow \{\eta; \Lambda F_1\}$, and obtain $\mathcal{U} :: \{\eta; \Lambda F_1\} \Leftrightarrow \mathbf{lam}\ x.\ e_1$ by $\mathsf{vtr\_lam}$ from $\mathcal{C}$.

```
mp_lam : map_eval (ev_lam) (tr_lam C1)
                  (fev_lam) (vtr_lam (tr_lam C1)).
```

Case: $\mathcal{D}$ ends in an application of the $\mathsf{ev\_app}$ rule.

$$\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1'\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ e_2 \hookrightarrow v_2\end{matrix} \qquad \begin{matrix}\mathcal{D}_3 \\ [v_2/x]e_1' \hookrightarrow v\end{matrix}}{e_1\ e_2 \hookrightarrow v}\ \mathsf{ev\_app}$$

This is the most interesting case, since it contains the essence of the argument how substitution can be replaced by binding variables to values in an environment.

$\mathcal{C} :: \eta \vdash F \leftrightarrow e_1\ e_2$ By assumption
$F = F_1\ F_2$,
$\mathcal{C}_1 :: \eta \vdash F_1 \leftrightarrow e_1$, and
$\mathcal{C}_2 :: \eta \vdash F_2 \leftrightarrow e_2$ By inversion and exclusion
$\mathcal{D}'_2 :: \eta \vdash F_2 \hookrightarrow W_2$ and
$\mathcal{U}_2 :: W_2 \Leftrightarrow v_2$ By ind. hyp. on $\mathcal{D}_2$ and $\mathcal{C}_2$
$\mathcal{D}'_1 :: \eta \vdash F_1 \hookrightarrow W_1$ and
$\mathcal{U}_1 :: W_1 \Leftrightarrow \textbf{lam}\ x.\ e'_1$ By ind. hyp. on $\mathcal{D}_1$ and $\mathcal{C}_1$
$W_1 = \{\eta_1; \Lambda F'_1\}$ and
$\mathcal{C}'_1 :: \eta_1 \vdash \Lambda F'_1 \leftrightarrow \textbf{lam}\ x.\ e'_1$ By inversion on $\mathcal{U}_1$

Applying inversion again to $\mathcal{C}'_1$ shows that the premiss must be the deduction of a hypothetical judgment. That is,

$$\mathcal{C}'_1 = \begin{array}{c} \overline{\quad w \Leftrightarrow x \quad}\, u \\ \mathcal{C}_3 \\ \eta_1, w \vdash F'_1 \leftrightarrow e'_1 \end{array}$$

where w is a new parameter ranging over values. This judgment is parametric in w and x and hypothetical in u. We can thus substitute W_2 for w, v_2 for x, and $\mathcal{U}_2$ for u to obtain a deduction

$$\mathcal{C}'_3 :: \eta_1, W_2 \vdash F'_1 \leftrightarrow [v_2/x]e'_1.$$

Now we apply the induction hypothesis to $\mathcal{D}_3$ and $\mathcal{C}'_3$ to obtain a W_3 and

$\mathcal{D}'_3 :: \eta_1, W_2 \vdash F'_1 \hookrightarrow W_3$ and
$\mathcal{U}_3 :: W_3 \Leftrightarrow v$.

We let $W = W_3$, $\mathcal{U} = \mathcal{U}_3$, and obtain $\mathcal{D}' :: \eta \vdash F_1\ F_2 \hookrightarrow W$ by fev_app from $\mathcal{D}'_1$, $\mathcal{D}'_2$, and $\mathcal{D}'_3$.

The implementation of this relatively complex reasoning employs again the magic of hypothetical judgments: the substitution we need to carry out to obtain $\mathcal{C}'_3$ from $\mathcal{C}_3$ is implemented as a function application.

```
mp_app : map_eval (ev_app D3 D2 D1) (tr_app C2 C1)
                  (fev_app D3' D2' D1') U3
              <- map_eval D1 C1 D1' (vtr_lam (tr_lam C3))
              <- map_eval D2 C2 D2' U2
              <- map_eval D3 (C3 W2 V2 U2) D3' U3.
```

This completes the proof once we have convinced ourselves that all possible cases have been considered. Note that whenever $\mathcal{C}$ ends in an application of the tr_1 or tr_↑ rules, then the first two cases apply. Otherwise one of the other two cases must apply, depending on the shape of $\mathcal{D}$. □

Theorem 6.1 and Lemma 6.3 together guarantee completeness of the translation.

Theorem 6.4 (Completeness) *For any closed expressions e and v and evaluation $\mathcal{D} :: e \hookrightarrow v$, there exist a de Bruijn expression F, a value W and deductions $\mathcal{C} :: \cdot \vdash F \leftrightarrow e$, $\mathcal{D}' :: \cdot \vdash F \hookrightarrow W$, and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: Lemma 6.3 shows that an evaluation $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and a translation $W \Leftrightarrow v$ exist for any translation $\mathcal{C} :: \eta \vdash F \leftrightarrow e$. Theorem 6.1 shows that a particular F and translation $\mathcal{C} :: \cdot \vdash F \leftrightarrow e$ exist, thus proving the theorem. □

Completeness is insufficient to guarantee compiler correctness. For example, the translation of values $W \Leftrightarrow v$ could relate *any* expression v to any value W, which would make the statement of the previous theorem almost trivially true. We need to check a further property, namely that any value which could be produced by evaluating the compiled code, could also be produced by direct evaluation as specified by the natural semantics. This is shown in the diagram below.

$$\begin{tikzcd}
e \arrow[r, dotted, "\mathcal{D} :: e \hookrightarrow v"] \arrow[d, "\mathcal{C} :: \eta \vdash F \leftrightarrow e"'] & v \\
\eta; F \arrow[r, "\mathcal{D}' :: \eta \vdash F \hookrightarrow W"'] & W \arrow[u, "\mathcal{U} :: W \Leftrightarrow v"']
\end{tikzcd}$$

We call this property *soundness* of the compiler, since it prohibits the compiled code from producing incorrect values. We prove this from a lemma which asserts the existence of an expression v, evaluation $\mathcal{D}$ and translation $\mathcal{U}$, given the translation $\mathcal{C}$ and evaluation $\mathcal{D}'$. This yields the theorem by showing that the translation $\mathcal{U} :: W \Leftrightarrow v$, is uniquely determined from W.

Lemma 6.5 *For any closed expression e, de Bruijn expression F, environment η, value W, deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, there exist an expression v and deductions $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: The proof proceeds by a straightforward induction over the structure of $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$. It heavily employs inversion (as the proof of completeness, Lemma 6.3). Interestingly, this proof can be implemented by literally the same judgment. We leave it as exercise 6.6 to write out the informal proof—its representation from the proof of completeness is summarized below. Using is as a program in this instance means that we assume that second and third arguments are given and the first and last argument are logic variables whose instantiation terms are to be constructed.

```
map_eval' : eval E V -> trans N F E
              -> feval N F W -> vtrans W V -> type.
%mode map_eval' -D +C +D' -U.

mp'_1 : map_eval' (ev_lam) (tr_1 (vtr_lam (tr_lam C2)))
                  (fev_1) (vtr_lam (tr_lam C2)).

mp'_^ : map_eval' D (tr_^ C1) (fev_^ D1') U1
         <- map_eval' D C1 D1' U1.

mp'_lam : map_eval' (ev_lam) (tr_lam C1)
                    (fev_lam) (vtr_lam (tr_lam C1)).

mp'_app : map_eval' (ev_app D3 D2 D1) (tr_app C2 C1)
            (fev_app D3' D2' D1') U3
            <- map_eval' D1 C1 D1' (vtr_lam (tr_lam C3))
            <- map_eval' D2 C2 D2' U2
            <- map_eval' D3 (C3 W2 V2 U2) D3' U3.

%terminates D' (map_eval' _ C D' _).
```

□

Theorem 6.6 (Uniqueness of Translations) *For any value W if there exist a v and a translation $\mathcal{U} :: W \Leftrightarrow v$, then v and $\mathcal{U}$ are unique. Furthermore, for any environment η and de Bruijn expression F, if there exist an e and a translation $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, then e and $\mathcal{C}$ are unique.*

Proof: By simultaneous induction on the structures of $\mathcal{U}$ and $\mathcal{C}$. In each case, either W or F uniquely determine the last inference. Since the translated expressions in the premisses are unique by induction hypothesis, so is the translated value in the conclusion. □

The proof requires no separate implementation in Elf in the same way that appeals to inversion remain implicit in the formulation of higher-level judgments. It is obtained by direct inspection of properties of the inference rules.

Theorem 6.7 (Soundness) *For any closed expressions e and v, de Bruijn expression F, environment η, value W, deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$, $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, and $\mathcal{U} :: W \Leftrightarrow v$, there exists a deduction $\mathcal{D} :: e \hookrightarrow v$.*

Proof: From Lemma 6.5 we infer the existence of a v, $\mathcal{U}$, and $\mathcal{D}$, given $\mathcal{C}$ and $\mathcal{D}'$. Theorem 6.6 shows that v and $\mathcal{U}$ are unique, and thus the property must hold for all v and $\mathcal{U} :: W \Leftrightarrow v$, which is what we needed to show. □

6.2 Adding Data Values and Recursion

In the previous section we treated only a very restricted core language of Mini-ML. In this section we will extend the compiler to the full Mini-ML language as presented in Chapter 2. The main additions to the core language which affect the compiler are data values (such as natural numbers and pairs) and recursion. The language of de Bruijn expressions is extended by allowing constructors that parallel ordinary expressions. We maintain a similar syntax, but mark de Bruijn expression constructors with a prime (′).

Expressions	F	$::=$	$\mid \mathbf{z}' \mid \mathbf{s}'\ F \mid \mathbf{case}'\ F_1\ F_2\ F_3$	*Natural Numbers*
			$\mid \langle F_1, F_2\rangle' \mid \mathbf{fst}'\ F \mid \mathbf{snd}'\ F$	*Pairs*
			$\mid \Lambda F \mid F_1\ F_2$	*Functions*
			$\mid \mathbf{let}'\,\mathbf{val}\ F_1\ \mathbf{in}\ F_2$	*Definitions*
			$\mid \mathbf{let}'\,\mathbf{name}\ F_1\ \mathbf{in}\ F_2$	
			$\mid \mathbf{fix}'\ F$	*Recursion*
			$\mid 1 \mid F\uparrow$	*Variables*

Expressions of the form $F\uparrow$ are not necessarily variables (where F is a sequence of shifts applied to 1), but it may be intuitively helpful to think of them that way. In the representation we need only first-order constants, since this language has no constructs binding variables by name.

```
exp'   : type.  %name exp' F f.

1      : exp'.
^      : exp' -> exp'.  %postfix 20 ^.
z'     : exp'.
s'     : exp' -> exp'.
case'  : exp' -> exp' -> exp' -> exp'.
pair'  : exp' -> exp' -> exp'.
fst'   : exp' -> exp'.
snd'   : exp' -> exp'.
lam'   : exp' -> exp'.
app'   : exp' -> exp' -> exp'.
letv'  : exp' -> exp' -> exp'.
letn'  : exp' -> exp' -> exp'.
fix'   : exp' -> exp'.
```

Next we need to extend the language of values. While data values can be added in a straightforward fashion, **let name** and recursion present some difficulties. Consider the evaluation rule for fixpoints.

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

We introduced the environment model of evaluation in order to eliminate the need for explicit substitution, where an environment is a list of values. In the case of the fixpoint construction we would need to bind the variable x to the expression **fix** $x.\ e$ in the environment in order to avoid substutition, but **fix** $x.\ e$ is not a value. The evaluation rules for de Bruijn expressions take advantage of the invariant that an environment contains only values. In particular, the rule

$$\frac{}{\eta, W \vdash 1 \hookrightarrow W}\,\mathsf{fev\_1}$$

requires that an environment contain only values. We will thus need to add a new environment constructor $\eta + F$ in order to allow unevaluated expressions in the environment. These considerations yield the following mutually recursive definitions of environments and values. We mark data values with a star (*) to distinguish them from expressions and de Bruijn expressions with the same name.

$$\begin{array}{rrcll}
\text{Environments} & \eta & ::= & \cdot \mid \eta, W \mid \eta + F & \\
\text{Values} & W & ::= & \mid \mathbf{z}^* \mid \mathbf{s}^*\ W & \textit{Natural Numbers} \\
& & & \mid \langle W_1, W_2\rangle^* & \textit{Pairs} \\
& & & \mid \{\eta; F\} & \textit{Closures}
\end{array}$$

The Elf representation is direct.

```
env    : type.  %name env N.
val    : type.  %name val W w.

empty  : env.
,      : env -> val -> env.   %infix left 10 ,.
+      : env -> exp' -> env.  %infix left 10 +.

z*     : val.
s*     : val -> val.

pair*  : val -> val -> val.

clo    : env -> exp' -> val.
```

In the extension of the evaluation rule to this completed language, we must exercise care in the treatment of the new environment constructor for unevaluated expression: when such an expression is looked up in the environment, it must be evaluated.

$$\frac{\eta \vdash F \hookrightarrow W}{\eta + F \vdash 1 \hookrightarrow W}\,\mathsf{fev\_1+} \qquad\qquad \frac{\eta \vdash F \hookrightarrow W}{\eta + F' \vdash F\!\uparrow\, \hookrightarrow W}\,\mathsf{fev\_\!\uparrow\!+}$$

The rules involving data values generally follow the patterns established in the natural semantics for ordinary expressions. The main departure from the earlier formulation is the separation of values from expressions. We show only four of the relevant rules.

$$\frac{}{\eta \vdash \mathbf{z}' \hookrightarrow \mathbf{z}^*}\,\textsf{fev\_z} \qquad \frac{\eta \vdash F \hookrightarrow W}{\eta \vdash \mathbf{s}'\ F \hookrightarrow \mathbf{s}^*\ W}\,\textsf{fev\_s}$$

$$\frac{\eta \vdash F_1 \hookrightarrow \mathbf{z}^* \qquad \eta \vdash F_2 \hookrightarrow W}{\eta \vdash \mathbf{case}'\ F_1\ F_2\ F_3 \hookrightarrow W}\,\textsf{fev\_case\_z}$$

$$\frac{\eta \vdash F_1 \hookrightarrow \mathbf{s}^*\ W_1' \qquad \eta, W_1' \vdash F_3 \hookrightarrow W}{\eta \vdash \mathbf{case}'\ F_1\ F_2\ F_3 \hookrightarrow W}\,\textsf{fev\_case\_s}$$

Evaluating a **let val**-expression also binds a variable to value by extending the environment.

$$\frac{\eta \vdash F_1 \hookrightarrow W_1 \qquad \eta, W_1 \vdash F_2 \hookrightarrow W}{\eta \vdash \mathbf{let\,val}'\ F_1\ \mathbf{in}\ F_2 \hookrightarrow W}\,\textsf{fev\_letv}$$

Evaluating a **let name**-expression binds a variable to an expression and thus requires the new environment constructor.

$$\frac{\eta + F_1 \vdash F_2 \hookrightarrow W}{\eta \vdash \mathbf{let\,name}'\ F_1\ \mathbf{in}\ F_2 \hookrightarrow W}\,\textsf{fev\_letn}$$

Fixpoint expressions are similar, except that the variable is bound to the **fix** expression itself.

$$\frac{\eta + \mathbf{fix}'\ F \vdash F \hookrightarrow W}{\eta \vdash \mathbf{fix}'\ F \hookrightarrow W}\,\textsf{fev\_fix}$$

For example, **fix** $x.\ x$ (considered on page 17) is represented by $\mathbf{fix}'\ 1$. Intuitively, evaluation of this expression should not terminate. An attempt to construct an evaluation leads to the sequence

$$\cfrac{\cfrac{\cfrac{\vdots}{\cdot \vdash \mathbf{fix}'\ 1 \hookrightarrow W}\,\textsf{fev\_fix}}{\cdot + \mathbf{fix}'\ 1 \vdash 1 \hookrightarrow W}\,\textsf{fev\_1+}}{\cdot \vdash \mathbf{fix}'\ 1 \hookrightarrow W}\,\textsf{fev\_fix.}$$

The implementation of these rules in Elf poses no particular difficulties. We show only the rules from above.

```
feval : env -> exp' -> val -> type.  %name feval D.
%mode feval +N +F -W.

% Variables
fev_1 : feval (N , W) 1 W.
fev_^ : feval (N , W') (F ^) W
          <- feval N F W.

fev_1+ : feval (N + F) 1 W
           <- feval N F W.
fev_^+ : feval (N + F') (F ^) W
           <- feval N F W.

% Natural Numbers
fev_z : feval N z' z*.
fev_s : feval N (s' F) (s* W)
         <- feval N F W.
fev_case_z : feval N (case' F1 F2 F3) W
              <- feval N F1 z*
              <- feval N F2 W.
fev_case_s : feval N (case' F1 F2 F3) W
              <- feval N F1 (s* W1)
              <- feval (N , W1) F3 W.

% Pairs
fev_pair : feval N (pair' F1 F2) (pair* W1 W2)
             <- feval N F1 W1
             <- feval N F2 W2.
fev_fst  : feval N (fst' F) W1
             <- feval N F (pair* W1 W2).
fev_snd  : feval N (snd' F) W2
             <- feval N F (pair* W1 W2).

% Functions
fev_lam : feval N (lam' F) (clo N (lam' F)).
fev_app : feval N (app' F1 F2) W
           <- feval N F1 (clo N' (lam' F1'))
           <- feval N F2 W2
           <- feval (N' , W2) F1' W.

% Definitions
fev_letv : feval N (letv' F1 F2) W
```

```
            <- feval N F1 W1
            <- feval (N , W1) F2 W.

fev_letn : feval N (letn' F1 F2) W
            <- feval (N + F1) F2 W.

% Recursion
fev_fix  : feval N (fix' F) W
             <- feval (N + (fix' F)) F W.
```

Next we need to extend the translation between expressions and de Bruijn expressions and values. We show a few interesting cases in the extended judgments $\eta \vdash F \leftrightarrow e$ and $W \Leftrightarrow v$. The case for **let val** is handled just like the case for **lam**, since we will always substitute a *value* for the variable bound by the **let** during execution.

$$\frac{}{\eta \vdash \mathbf{z}' \leftrightarrow \mathbf{z}}\,\mathsf{tr\_z} \qquad \frac{\eta \vdash F \leftrightarrow e}{\eta \vdash \mathbf{s}'\ F \leftrightarrow \mathbf{s}\ e}\,\mathsf{tr\_s}$$

$$\frac{\eta \vdash F_1 \leftrightarrow e_1 \qquad \begin{array}{c}\overline{w \Leftrightarrow x}\ u\\ \vdots \\ \eta, w \vdash F_2 \leftrightarrow e_2\end{array}}{\eta \vdash \mathbf{let\,val}'\ F_1\ \mathbf{in}\ F_2 \leftrightarrow \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}\,\mathsf{tr\_letv}^{w,x,u}$$

where the right premiss of tr_let is parametric in w and x and hypothetical in u. In order to preserve the basic structure of the proofs of lemmas 6.3 and 6.5, we must treat the **let name** and **fix** constructs somewhat differently: we extend the environment with an *expression* parameter (not a value parameter) using the new environment constructor $+$.

$$\dfrac{\eta \vdash F_1 \leftrightarrow e_1 \qquad \begin{array}{c}\overline{\eta \vdash f \Leftrightarrow x}\,u \\ \vdots \\ \eta + f \vdash F_2 \leftrightarrow e_2\end{array}}{\eta \vdash \mathbf{let\,name'}\; F_1 \;\mathbf{in}\; F_2 \leftrightarrow \mathbf{let}\; x = e_1 \;\mathbf{in}\; e_2}\,\mathsf{tr\_letn}^{f,x,u}$$

$$\dfrac{\begin{array}{c}\overline{\eta \vdash f \leftrightarrow x}\,u \\ \vdots \\ \eta + f \vdash F \leftrightarrow e\end{array}}{\eta \vdash \mathbf{fix'}\; F \leftrightarrow \mathbf{fix}\; x.\; e}\,\mathsf{tr\_fix}^{f,x,u}$$

$$\dfrac{\eta \vdash F \leftrightarrow e}{\eta + F \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1+} \qquad \dfrac{\eta \vdash F \leftrightarrow e}{\eta + F' \vdash F\!\uparrow\, \leftrightarrow e}\,\mathsf{tr\_\!\uparrow\!+}$$

Finally, the value translation does not have to deal with fixpoint-expressions (they are not values). We only show the three new cases.

$$\dfrac{}{\mathbf{z}^* \Leftrightarrow \mathbf{z}}\,\mathsf{vtr\_z} \qquad \dfrac{W \Leftrightarrow v}{\mathbf{s}^*\; W \Leftrightarrow \mathbf{s}\; v}\,\mathsf{vtr\_s}$$

$$\dfrac{W_1 \Leftrightarrow v_1 \qquad W_2 \Leftrightarrow v_2}{\langle W_1, W_2\rangle^* \Leftrightarrow \langle v_1, v_2\rangle}\,\mathsf{vtr\_pair}$$

Deductions of parametric and hypothetical judgments are represented by functions, as usual.

```
trans  : env -> exp' -> exp -> type.  %name trans C.
vtrans : val -> exp -> type.          %name vtrans U.
% can be used in different directions
%mode trans +N +F -E.
%mode vtrans +W -V.

% Natural numbers
tr_z     : trans N z' z.
tr_s     : trans N (s' F) (s E)
            <- trans N F E.
```

```
tr_case : trans N (case’ F1 F2 F3) (case E1 E2 E3)
           <- trans N F1 E1
           <- trans N F2 E2
           <- ({w:val} {x:exp}
                 vtrans w x -> trans (N , w) F3 (E3 x)).

% Pairs
tr_pair : trans N (pair’ F1 F2) (pair E1 E2)
             <- trans N F1 E1
             <- trans N F2 E2.
tr_fst  : trans N (fst’ F1) (fst E1)
             <- trans N F1 E1.
tr_snd  : trans N (snd’ F1) (snd E1)
             <- trans N F1 E1.

% Functions
tr_lam : trans N (lam’ F) (lam E)
            <- ({w:val} {x:exp}
                  vtrans w x -> trans (N , w) F (E x)).
tr_app : trans N (app’ F1 F2) (app E1 E2)
            <- trans N F1 E1
            <- trans N F2 E2.

% Definitions
tr_letv: trans N (letv’ F1 F2) (letv E1 E2)
            <- trans N F1 E1
            <- ({w:val} {x:exp}
                  vtrans w x -> trans (N , w) F2 (E2 x)).

tr_letn: trans N (letn’ F1 F2) (letn E1 E2)
            <- trans N F1 E1
            <- ({f:exp’} {x:exp}
                  trans N f x -> trans (N + f) F2 (E2 x)).

% Recursion
tr_fix : trans N (fix’ F) (fix E)
            <- ({f:exp’} {x:exp}
                  trans N f x -> trans (N + f) F (E x)).

% Variables
tr_1  : trans (N , W) 1 E <- vtrans W E.
tr_^  : trans (N , W) (F ^) E <- trans N F E.
```

```
tr_1+ : trans (N + F) 1 E <- trans N F E.
tr_^+ : trans (N + F') (F ^) E <- trans N F E.

% Natural number values
vtr_z : vtrans z* z.
vtr_s : vtrans (s* W) (s V)
         <- vtrans W V.

% Pair values
vtr_pair : vtrans (pair* W1 W2) (pair V1 V2)
            <- vtrans W1 V1
            <- vtrans W2 V2.

% Function values
vtr_lam : vtrans (clo N (lam' F)) (lam E)
           <- trans N (lam' F) (lam E).
```

In order to extend the proof of compiler correctness in Section 6.1 we need to extend various lemmas.

Theorem 6.8 *For every closed expression e there exists a de Bruijn expression F such that $\cdot \vdash F \leftrightarrow e$.*

Proof: We generalize analogously to Lemma 6.2 and prove the modified lemma by induction on the structure of e (see Exercise 6.7). □

Lemma 6.9 *If $W \Leftrightarrow e$ is derivable, then e Value is derivable.*

Proof: By a straightforward induction on the structure of $\mathcal{U} :: W \Leftrightarrow e$. □

Lemma 6.10 *If e Value and $e \hookrightarrow v$ are derivable then $e = v$.*

Proof: By a straightforward induction on the structure of $\mathcal{P} :: e$ *Value*. □

The Elf implementations of the proofs of Lemmas 6.9 and 6.10 is straightforward and can be found in the on-line material that accompanies these notes. The type families are

```
vtrans_val : vtrans W E -> value E -> type.
%mode vtrans_val +U -P.

val_eval  : value E -> eval E E -> type.
%mode val_eval +P -D.
```

The next lemma is the main lemma is the proof of completeness, that is, every value which can obtained by direct evaluation can also be obtained by compilation, evaluation of the compiled code, and translation of the returned value to the original language.

Lemma 6.11 *For any closed expressions e and v, environment η, de Bruijn expression F, deduction $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, there exist a value W and deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: By induction on the structure of $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$. In this induction, as in the proof of Lemma 6.3, we assume the induction hypothesis on the premisses of $\mathcal{D}$ and for arbitrary $\mathcal{C}$, and on the premisses of $\mathcal{C}$ if $\mathcal{D}$ remains fixed. The implementation is an extension of the previous higher-level judgment,

```
map_eval : eval E V -> trans N F E
             -> feval N F W -> vtrans W V -> type.
%mode map_eval +D +C -D' -U.
```

We show only some of the typical cases—the others are straightforward and left to the reader or remain unchanged from the proof of Lemma 6.3

Case: $\mathcal{C}$ ends in an application of the $\mathsf{tr\_1}$ rule.

$$\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{U}_1\\ W_1 \Leftrightarrow e\end{array}}{\eta_1, W_1 \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1}$$

This case changes from the previous proof, since there we applied simple inversion (there was only one possible kind of value) to conclude that $e = v$. Here we need two lemmas from above.

$\mathcal{D} :: e \hookrightarrow v$	Assumption
$\mathcal{P} :: e$ *Value*	By Lemma 6.9 from $\mathcal{U}_1$
$e = v$	By Lemma 6.10 from $\mathcal{P}$

Hence we can let W be W_1, $\mathcal{U}$ be $\mathcal{U}_1$, and $\mathcal{D}'$ be $\mathsf{fev\_1} :: \eta_1, W_1 \vdash 1 \hookrightarrow W_1$. The implementation explicitly appeals to the implementations of the lemmas.

```
mp_1 : map_eval D (tr_1 U1) (fev_1) U1
          <- vtrans_val U1 P
          <- val_eval P D.
```

Case: $\mathcal{C}$ ends in an application of the $\mathsf{tr\_\uparrow}$ rule. This case proceeds as before.

```
mp_^ : map_eval D (tr_^ C1) (fev_^ D1') U1
          <- map_eval D C1 D1' U1.
```

Case: $\mathcal{C}$ ends in an application of the tr_1+ rule.

$$\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1\\ \eta_1 \vdash F_1 \leftrightarrow e\end{array}}{\eta_1 + F_1 \vdash 1 \leftrightarrow e}\,\textsf{tr\_1+}$$

$\mathcal{D} :: e \hookrightarrow v$ Assumption
$\mathcal{D}'_1 :: \eta_1 \vdash F_1 \hookrightarrow W_1$ and
$\mathcal{U}_1 :: W_1 \Leftrightarrow v$ By ind. hyp. on $\mathcal{D}$ and $\mathcal{C}_1$
$\mathcal{D}' :: \eta_1 + F_1 \vdash 1 \hookrightarrow W$ By fev_1+ from $\mathcal{D}'_1$

and we can let $W = W_1$ and $\mathcal{U} = \mathcal{U}_1$.

```
mp_1+ : map_eval D (tr_1+ C1) (fev_1+ D1') U1
           <- map_eval D C1 D1' U1.
```

Case: $\mathcal{C}$ ends in an application of the tr_↑+ rule. This case is just like the tr_↑ case.

```
mp_^+ : map_eval D (tr_^+ C1) (fev_^+ D1') U1
           <- map_eval D C1 D1' U1.
```

For the remaining cases we may assume that none of the four cases above apply. We only show the case for fixpoints.

Case: $\mathcal{D}$ ends in an application of the ev_fix rule.

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ [\textbf{fix}\ x.\ e_1/x]e_1 \hookrightarrow v\end{array}}{\textbf{fix}\ x.\ e_1 \hookrightarrow v}\,\textsf{ev\_fix}$$

$\mathcal{C} :: \eta \vdash F \leftrightarrow \textbf{fix}\ x.\ e_1$ By assumption

By inversion and exclusion (of the previous cases), $\mathcal{C}$ must end in an application of the tr_fix rule and thus $F = \textbf{fix}'\ F_1$ for some F_1 and there is a deduction $\mathcal{C}_1$, parametric in f and x and hypothetical in u, of the form

$$\begin{array}{c}\overline{\eta \vdash f \leftrightarrow x}\ u\\ \mathcal{C}_1\\ \eta + f \vdash F_1 \leftrightarrow e_1\end{array}$$

In this deduction we can substitute $\mathbf{fix}'\ F_1$ for f and $\mathbf{fix}\ x.\ e_1$ for x, and replace the resulting hypothesis $u :: \eta \vdash \mathbf{fix}'\ F_1 \leftrightarrow \mathbf{fix}\ x.\ e_1$ by $\mathcal{C}$! This way we obtain a deduction

$$\mathcal{C}_1' :: \eta + \mathbf{fix}'\ F_1 \vdash F_1 \leftrightarrow [\mathbf{fix}\ x.\ e_1/x]e_1.$$

Now we can apply the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{C}_1'$ which yields a W_1 and deductions

$\mathcal{D}_1' :: \eta + \mathbf{fix}'\ F_1 \vdash F_1 \hookrightarrow W_1$ and
$\mathcal{U}_1 :: W_1 \Leftrightarrow v$ By ind. hyp. on $\mathcal{D}_1$ and $\mathcal{C}_1'$

Applying fev_fix to $\mathcal{D}_1'$ results in a deduction

$$\mathcal{D}' :: \eta \vdash \mathbf{fix}'\ F_1 \hookrightarrow W_1$$

and we let W be W_1 and $\mathcal{U}$ be $\mathcal{U}_1$. In Elf, the substitutions into the hypothetical deduction are implemented by applications of the representing function `C1`.

```
mp_fix : map_eval (ev_fix D1) (tr_fix C1)
                  (fev_fix D1') U1
            <- map_eval D1 (C1 (fix' F1) (fix E1) (tr_fix C1))
                        D1' U1.
```

□

This lemma and the totality of the translation relation in its expression argument (Theorem 6.8) together guarantee completeness of the translation.

Theorem 6.12 (Completeness) *For any closed expressions e and v and evaluation $\mathcal{D} :: e \hookrightarrow v$, there exist a de Bruijn expression F, a value W and deductions $\mathcal{C} :: \cdot \vdash F \leftrightarrow e$, $\mathcal{D}' :: \cdot \vdash F \hookrightarrow W$, and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: As in the proof of Theorem 6.4, but using Lemma 6.11 and Theorem 6.8 instead of Lemma 6.3 and Theorem 6.1. □

Lemma 6.13 *For any closed expression e, de Bruijn expression F, environment η, value W, deduction $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, there exist an expression v and deductions $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: By induction on the structure of $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$. The family `map_eval` which implements the main lemma in the soundness proof, also implements the proof of this lemma without any change. □

Theorem 6.14 (Uniqueness of Translations) *For any value W if there exist a v and a translation $\mathcal{U} :: W \Leftrightarrow v$, then v and $\mathcal{U}$ are unique. Furthermore, for any environment η and de Bruijn expression F, if there exist an e and a translation $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, then e and $\mathcal{C}$ are unique.*

Proof: As before, by a simultaneous induction on the structures of $\mathcal{U}$ and $\mathcal{C}$. □

Theorem 6.15 (Soundness) *For any closed expressions e and v, de Bruijn expression F, environment η, value W, deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$, $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, and $\mathcal{U} :: W \Leftrightarrow v$, there exists a deduction $\mathcal{D} :: e \hookrightarrow v$.*

Proof: From Lemma 6.13 we infer the existence of a v, $\mathcal{U}$, and $\mathcal{D}$, given $\mathcal{C}$ and $\mathcal{D}'$. Theorem 6.14 shows that v and $\mathcal{U}$ are unique, and thus the property must hold for all v and $\mathcal{U} :: W \Leftrightarrow v$, which is what we needed to show. □

6.3 Computations as Transition Sequences

So far, we have modelled evaluation as the construction of a deduction of the evaluation judgment. This is true for evaluation based on substitution in Section 2.3 and for evaluation based on environments in Section 6.1. In an abstract machine (and, of course, in an actual machine) a more natural model for computation is a sequence of states. In this section we will develop the CLS machine, an abstract machine similar in scope to the SECD machine [Lan64]. The CLS machine still interprets expressions, so the step from environment based evaluation to this abstract machine does not involve any compilation. Instead, we flatten evaluation trees to sequences of states that describe the computation. This flattening involves some rather arbitrary decisions about which subcomputations should be performed first. We linearize the evaluation deductions beginning with the deduction of the leftmost premiss.

Throughout the remainder of this chapter, we will drop the prime ($'$) from the expression constructors. This should not lead to any confusion, since we no longer need to refer to the original expressions. Now consider the rule for evaluating pairs as a simple example where an evaluation tree has two branches.

$$\frac{\eta \vdash F_1 \hookrightarrow W_1 \qquad \eta \vdash F_2 \hookrightarrow W_2}{\eta \vdash \langle F_1, F_2 \rangle \hookrightarrow \langle W_1, W_2 \rangle^*}\,\textsf{fev\_pair}$$

An abstract machine would presumably start in a state where it is given the environment η and the expression $\langle F_1, F_2 \rangle$. The final state of the machine should somehow indicate the final value $\langle W_1, W_2 \rangle^*$. The computation naturally decomposes into three phases: the first phase computes the value of F_1 in environment η, the second phase computes the value of F_2 in environment η, and the third phase combines the two values to form a pair. These phases mean that we have to preserve the environment η and also the expression F_2 while we are computing the value of F_1. Similarly, we have to save the value W_1 while computing the value of F_2. A natural data structure for saving components of a state is a stack. The considerations above suggest three stacks: a stack Ξ of environments, a stack of expressions to be evaluated, and a stack S of values. However, we also need to remember that, after the evaluation of F_2 we need to combine W_1 and W_2 into a pair. Thus, instead of a stack of expression to be evaluated, we maintain a *program* which consists of expressions and special instructions (such as: *make a pair* written as *mkpair*).

We will need more instructions later, but so far we have:

$$\begin{array}{rrcl}
\text{Instructions} & I & ::= & F \mid \mathit{mkpair} \mid \ldots \\
\text{Programs} & P & ::= & \mathit{done} \mid I \mathbin{\&} P \\
\text{Environment Stacks} & \Xi & ::= & \cdot \mid \Xi; \eta \\
\text{Value Stacks} & S & ::= & \cdot \mid S, W \\
\text{State} & \mathit{St} & ::= & \langle \Xi, P, S \rangle
\end{array}$$

Note that value stacks are simply environments, so we will not formally distinguish them from environments. The instructions of a program a sequenced with &; the program *done* indicates that there are no further instructions, that is, computation should stop.

A state consists of an environment stack Ξ, a program P and a value stack S, written as $\langle \Xi, P, S \rangle$. We have single-step and multi-step transition judgments:

$$\begin{array}{ll}
\mathit{St} \Longrightarrow \mathit{St}' & \mathit{St} \text{ goes to } \mathit{St}' \text{ in one computation step} \\
\mathit{St} \overset{*}{\Longrightarrow} \mathit{St}' & \mathit{St} \text{ goes to } \mathit{St}' \text{ in zero or more steps}
\end{array}$$

We define the transition judgment so that

$$\langle (\cdot; \eta), F \mathbin{\&} \mathit{done}, \cdot \rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot, W) \rangle$$

corresponds to the evaluation of F in environment η to value W. The free variables of F are therefore bound in the innermost environment, and the value resulting from evaluation is deposited on the top of the value stack, which starts out empty. Global evaluation is expressed in the judgment

$$\eta \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\Rightarrow} W \quad F \text{ computes to } W \text{ in environment } \eta$$

which is defined by the single inference rule

$$\frac{\langle (\cdot; \eta), F \mathbin{\&} \mathit{done}, \cdot \rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot, W) \rangle}{\eta \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\Rightarrow} W}\,\textsf{run}.$$

We prove in Theorem 6.19 that $\eta \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\Rightarrow} W$ iff $\eta \vdash F \hookrightarrow W$. We cannot prove this statement directly by induction (in either direction), since during a computation situations arise where the environment stack consists of more than a single environment, the remaining program is not *done*, *etc.* In one direction we generalize it to

$$\langle(\Xi;\eta), F \,\&\, P, S\rangle \stackrel{*}{\Longrightarrow} \langle\Xi, P, (S, W)\rangle$$

if $\eta \vdash F \hookrightarrow W$. This is the subject of Lemma 6.16. A slightly modified form of the converse is given in Lemma 6.18.

The transition rules and the remaining instructions can be developed systematically from the intuition provided above. First, we reconsider the evaluation of pairing. The first rule decomposes the pair expression and saves the environment η on the environment stack.

$$\mathsf{c\_pair} :: \langle(\Xi;\eta), \langle F_1, F_2\rangle \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, F_2 \,\&\, \mathit{mkpair} \,\&\, P, S\rangle$$

Here c_pair labels the rule and can be thought of as the deduction of the given transition judgment. The evaluation of F_1, if it terminates, leads to a state

$$\langle(\Xi;\eta), F_2 \,\&\, \mathit{mkpair} \,\&\, P, (S, W_1)\rangle,$$

and the further evaluation of F_2 then leads to a state

$$\langle\Xi, \mathit{mkpair} \,\&\, P, (S, W_1, W_2)\rangle.$$

Thus, the *mkpair* instruction should cause the machine to create a pair from the first two elements on the value stack and deposit the result again on the value stack. That is, we need as another rule:

$$\mathsf{c\_mkpair} :: \langle\Xi, \mathit{mkpair} \,\&\, P, (S, W_1, W_2)\rangle \Longrightarrow \langle\Xi, P, (S, \langle W_1, W_2\rangle^*)\rangle.$$

We consider one other construct in detail: application. To evaluate an application $F_1\ F_2$ we first evaluate F_1 and then we evaluate F_2. If the value of F_1 is a closure, we have to bind its variable to the value of F_2 and continue evaluation in an extended environment. The instruction that unwraps the closure and extends the environment is called *apply*.

$$\begin{array}{rcl}
\mathsf{c\_app} & :: & \langle(\Xi;\eta), F_1\ F_2 \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, F_2 \,\&\, \mathit{apply} \,\&\, P, S\rangle \\
\mathsf{c\_apply} & :: & \langle\Xi, \mathit{apply} \,\&\, P, (S, \{\eta'; \Lambda F_1'\}, W_2)\rangle \Longrightarrow \langle(\Xi;(\eta', W_2)), F_1' \,\&\, P, S\rangle
\end{array}$$

The rules for applying zero and successor are straightforward, but they necessitate a new operator *add1* to increment the first value on the stack.

$$\begin{array}{rcl}
\mathsf{c\_z} & :: & \langle(\Xi;\eta), \mathbf{z} \,\&\, P, S\rangle \Longrightarrow \langle\Xi, P, (S, \mathbf{z}^*)\rangle \\
\mathsf{c\_s} & :: & \langle(\Xi;\eta), \mathbf{s}\ F \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta), F \,\&\, \mathit{add1} \,\&\, P, S\rangle \\
\mathsf{c\_add1} & :: & \langle\Xi, \mathit{add1} \,\&\, P, (S, W)\rangle \Longrightarrow \langle\Xi, P, (S, \mathbf{s}^*\ W)\rangle
\end{array}$$

For expressions of the form **case** $F_1\ F_2\ F_3$, we need to evaluate F_1 and then evaluate either F_2 or F_3, depending on the value of F_1. This requires a new instruction, *branch*, which either goes to the next instructions or skips the next instruction. In the latter case it also needs to bind a new variable in the environment to the predecessor of the value of F_1.

$$\begin{array}{rl}
\mathsf{c\_case} :: & \langle(\Xi;\eta), \mathbf{case}\ F_1\ F_2\ F_3 \,\&\, P, S\rangle \\
& \Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, S\rangle \\
\mathsf{c\_branch\_z} :: & \langle(\Xi;\eta), \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, (S, \mathbf{z}^*)\rangle \Longrightarrow \langle(\Xi;\eta), F_2 \,\&\, P, S\rangle \\
\mathsf{c\_branch\_s} :: & \langle(\Xi;\eta), \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, (S, \mathbf{s}^*\ W)\rangle \\
& \Longrightarrow \langle(\Xi;(\eta, W)), F_3 \,\&\, P, S\rangle
\end{array}$$

Rules for **fst** and **snd** require new instructions to extract the first or second element of the value on the top of the stack.

$$\begin{array}{rcl}
\mathsf{c\_fst} & :: & \langle(\Xi;\eta), \mathbf{fst}\ F \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta), F \,\&\, \mathit{getfst} \,\&\, P, S\rangle \\
\mathsf{c\_getfst} & :: & \langle\Xi, \mathit{getfst} \,\&\, P, (S, \langle W_1, W_2\rangle^*)\rangle \Longrightarrow \langle\Xi, P, (S, W_1)\rangle \\
\mathsf{c\_snd} & :: & \langle(\Xi;\eta), \mathbf{snd}\ F \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta), F \,\&\, \mathit{getsnd} \,\&\, P, S\rangle \\
\mathsf{c\_getsnd} & :: & \langle\Xi, \mathit{getsnd} \,\&\, P, (S, \langle W_1, W_2\rangle^*)\rangle \Longrightarrow \langle\Xi, P, (S, W_2)\rangle
\end{array}$$

In order to handle **let val** we introduce another new instruction *bind*, even though it is not strictly necessary and could be simulated with other instructions (see Exercise 6.10).

$$\begin{array}{rl}
\mathsf{c\_let} :: & \langle(\Xi;\eta), \mathbf{let}\ F_1\ \mathbf{in}\ F_2 \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, \mathit{bind} \,\&\, F_2 \,\&\, P, S\rangle \\
\mathsf{c\_bind} :: & \langle(\Xi;\eta), \mathit{bind} \,\&\, F_2 \,\&\, P, (S; W_1)\rangle \Longrightarrow \langle(\Xi;(\eta, W_1)), F_2 \,\&\, P, S\rangle
\end{array}$$

We leave the rules for recursion to Exercise 6.11. The rules for variables and abstractions thus complete the specification of the single-step transition relation.

$$\begin{array}{rcl}
\mathsf{c\_1} & :: & \langle(\Xi;(\eta, W)), 1 \,\&\, P, S\rangle \Longrightarrow \langle\Xi, P, (S, W)\rangle \\
\mathsf{c\_\uparrow} & :: & \langle(\Xi;(\eta, W')), F\uparrow \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta), F \,\&\, P, S\rangle \\
\mathsf{c\_lam} & :: & \langle(\Xi;\eta), \Lambda F \,\&\, P, S\rangle \Longrightarrow \langle\Xi, P, (S, \{\eta; \Lambda F\})\rangle
\end{array}$$

The set of instructions extracted from these rules is

$$\text{Instructions}\quad I \quad ::= \quad F \mid \mathit{add1} \mid \mathit{branch} \mid \mathit{mkpair} \mid \mathit{getfst} \mid \mathit{getsnd} \mid \mathit{apply} \mid \mathit{bind}.$$

We view each of the transition rules for the single-step transition judgment as an axiom. Note that there are no other inference rules for this judgment. A partial computation is defined as a multi-step transition. This is easily defined via the following two inference rules.

$$\frac{}{St \overset{*}{\Longrightarrow} St}\,\mathsf{id} \qquad\qquad \frac{St \Longrightarrow St' \qquad St' \overset{*}{\Longrightarrow} St''}{St \overset{*}{\Longrightarrow} St''}\,\mathsf{step}$$

This definition guarantees that the end state of one transition matches the beginning state of the remaining transition sequence. Without the aid of dependent types we would have to define a computation as a list states and ensure externally that the end state of each transition matches the beginning state of the next. This use of dependent types to express complex constraints is one of the reasons why simple lists do not arise very frequently in Elf programming.

Deductions of the judgment $St \overset{*}{\Longrightarrow} St'$ have a very simple form: They all consist of a sequence of single steps terminated by an application of the id rule. We will follow standard practice and use a linear notation for sequences of steps:

$$St_1 \Longrightarrow St_2 \Longrightarrow \cdots \Longrightarrow St_n$$

Similarly, we will mix multi-step and single-step transitions in sequences, with the obvious meaning. We write $\mathcal{C}_1 \circ \mathcal{C}_2$ for the result of appending computations $\mathcal{C}_1$ and $\mathcal{C}_2$. This only makes sense if the final state of $\mathcal{C}_1$ is the same as the start state of $\mathcal{C}_2$. The $\circ$ operator is associative (see Exercise 6.12).

Recall that a complete computation was defined as a sequence of transitions from an initial state to a final state. The latter is characterized by the program *done*, and empty environment stack, and a value stack containing exactly one value, namely the result of the computation.

$$\frac{\langle (\cdot; \eta), F \mathbin{\&} \mathit{done}, \cdot \rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot, W) \rangle}{\eta \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\!\Rightarrow} W}\,\mathsf{run}$$

The representation of the abstract machine and the computation judgments present no particular difficulties. We begin with the syntax.

```
instruction : type.  %name instruction I.
program     : type.  %name program P.
envstack    : type.  %name envstack Ns.
state       : type.  %name state St.

% Instructions
ev     : exp' -> instruction.
add1   : instruction.
branch : instruction.
mkpair : instruction.
getfst : instruction.
getsnd : instruction.
apply  : instruction.
bind   : instruction.

% Programs
done : program.
&    : instruction -> program -> program.
```

```
%infix right 10 &.

% Environment stacks
emptys : envstack.
;      : envstack -> env -> envstack.
%infix left 10 ;.

% States
st : envstack -> program -> env -> state.
```

The computation rules are also a straightforward transcription of the rules above. The judgment $St \overset{*}{\Longrightarrow} St'$ is represented by a type `St => St'` where `=>` is a type family indexed by two states and written in infix notation. We show only three example rules.

```
=> : state -> state -> type.          %name => R.
%infix none 10 =>.
%mode => +St -St'.

c_z       : st (Ns ; N) (ev z' & P) S => st Ns P (S , z*).

c_app     : st (Ns ; N) (ev (app' F1 F2) & P) S
          => st (Ns ; N ; N) (ev F1 & ev F2 & apply & P) S.
c_apply   : st Ns (apply & P) (S , clo N' (lam' F1') , W2)
          => st (Ns ; (N' , W2)) (ev F1' & P) S.
```

The multi-step transition is defined by the transcription of its two inference rules. We write `~` in infix notation rather than step since it leads to a concise and readable notation for sequences of computation steps.

```
=>*  : state -> state -> type.                %name =>* C.
%infix none 10 =>*.
% no mode---this is not operational.

id   : St =>* St.

~     : St => St'
    -> St' =>* St''
    -> St =>* St''.

%infix right 10 ~.
```

Complete computations appeal directly to the multi-step computation judgment. We write `ceval K F W` for $\eta \vdash F \overset{*}{\Longrightarrow} W$.

```
ceval : env -> exp' -> val -> type.
% no mode---this is not operational

run :     st (emptys ; N) (ev F & done) (empty)
      =>* st (emptys) (done) (empty , W)
    -> ceval N F W.
```

While this representation is declaratively adequate it has a serious operational defect when used for evaluation, that is, when η and F are given and W is to be determined. The declaration for step (written as ~) solves the innermost subgoal first, that is we reduce the goal of finding a computation $\mathcal{C}'' :: St \overset{*}{\Longrightarrow} St''$ to finding a state St' and computation of $\mathcal{C}' :: St' \overset{*}{\Longrightarrow} St''$ and only then a single transition $R :: St \Longrightarrow St'$. This leads to non-termination, since the interpreter is trying to work its way backwards through the space of possible computation sequences. Instead, we can get linear, backtracking-free behavior if we first find the single step $R :: St \Longrightarrow St'$ and then the remainder of the computation $\mathcal{C}' :: St' \overset{*}{\Longrightarrow} St''$. Since there is exactly one rule for any instruction I and id will apply only when the program P is *done*, finding a computation now becomes a deterministic process. Executable versions of the last two judgments are given below. They differ from the one above only in the order of the recursive calls and it is a simple matter to relate the two versions formally.

```
>=>*  : state -> state -> type.
%infix none 10 >=>*.
%mode >=>* +St -St'.

id<   : St >=>* St.
<=<   : St >=>* St''
         <- St => St'
         <- St' >=>* St''.
%infix left 10 <=<.

>ceval : env -> exp' -> val -> type.
%mode >ceval +N +F -W.

>run   : >ceval N F W
          <- st (emptys ; N) (ev F & done) (empty)
             >=>* st (emptys) (done) (empty , W).
```

This example clearly illustrates that Elf should be thought of a uniform language in which one can express specifications (such as the computations above) and implementations (the operational versions below), but that many specifications will not be executable. This is generally the situation in logic programming languages.

In the informal development it is clear (and not usually separately formulated as a lemma) that computation sequences can be concatenated if the final state of the first computation matches the initial state of the second computation. In the formalization of the proofs below, we will need to explicitly implement a type family that appends computation sequences. It cannot be formulated as a function, since such a function would have to be recursive and is thus not definable in LF.

```
append : st Ns P S =>* st Ns' P' S'
      -> st Ns' P' S' =>* st Ns'' P'' S''
      -> st Ns P S =>* st Ns'' P'' S''
      -> type.
%mode append +C +C' -C''.
```

The defining clauses are left as Exercise 6.12.

We now return to the task of proving the correctness of the abstract machine. The first lemma states the fundamental motivating property for this model of computation.

Lemma 6.16 *Let η be an environment, F an expression, and W a value such that $\eta \vdash F \hookrightarrow W$. Then, for any environment stack Ξ, program P and stack S,*

$$\langle(\Xi;\eta), F \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle\Xi, P, (S, W)\rangle$$

Proof: By induction on the structure of $\mathcal{D} :: \eta \vdash F \hookrightarrow W$. We will construct a deduction of $\mathcal{C} :: \langle(\Xi;\eta), F \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle\Xi, P, (S, W)\rangle$. The proof is straightforward and we show only two typical cases. The implementation in Elf takes the form of a higher-level judgment `subcomp` that relates evaluations to computation sequences.

```
subcomp : feval N F W
           -> st (Ns ; N) (ev F & P) S =>* st Ns P (S , W)
           -> type.
%mode +{N:env} +{F:exp'} +{W:val} +{Ns:envstack} +{P:program} +{S:env}
    +{D:feval N F W} -{C:st (Ns ; N) (ev F & P) S =>* st Ns P (S , W)}
    subcomp D C.
```

Note that the mode declaration here is in the "full" form in order to express that the implicitly quantified variables `Ns`, `P`, and `S` are input arguments rather than output arguments. The system would have inferred the latter from the simpler declaration `%mode subcomp +D -C`.

Case: $\mathcal{D}$ ends in an application of the rule $\mathsf{fev\_z}$.

$$\mathcal{D} = \frac{}{\eta \vdash \mathbf{z} \hookrightarrow \mathbf{z}^*}\,\mathsf{fev\_z}.$$

Then the single-step transition

$$\langle(\Xi;\eta), \mathbf{z}\,\&\,P, S\rangle \overset{*}{\Longrightarrow} \langle\Xi, P, (S, \mathbf{z}^*)\rangle$$

satisfies the requirements of the lemma. The clause corresponding to this case:

```
sc_z : subcomp (fev_z) (c_z ~ id).
```

Case: $\mathcal{D}$ ends in an application of the fev_app rule.

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\eta \vdash F_1 \hookrightarrow \{\eta'; \Lambda F_1'\}} \quad \overset{\mathcal{D}_2}{\eta \vdash F_2 \hookrightarrow W_2} \quad \overset{\mathcal{D}_3}{\eta', W_2 \vdash F_1' \hookrightarrow W}}{\eta \vdash F_1\ F_2 \hookrightarrow W}\ \mathsf{fev\_app}$$

Then

$$\begin{array}{lr}
\langle(\Xi;\eta), F_1\ F_2\,\&\,P, S\rangle & \\
\Longrightarrow \langle(\Xi;\eta;\eta), F_1\,\&\,F_2\,\&\,\mathit{apply}\,\&\,P, S\rangle & \text{By rule } \mathsf{c\_app} \\
\overset{*}{\Longrightarrow} \langle(\Xi;\eta), F_2\,\&\,\mathit{apply}\,\&\,P, (S, \{\eta'; \Lambda F_1'\})\rangle & \text{By ind. hyp. on } \mathcal{D}_1 \\
\overset{*}{\Longrightarrow} \langle\Xi, \mathit{apply}\,\&\,P, (S, \{\eta'; \Lambda F_1'\}, W_2)\rangle & \text{By ind. hyp. on } \mathcal{D}_2 \\
\Longrightarrow \langle(\Xi;(\eta', W_2)), F_1'\,\&\,P, S\rangle & \text{By rule } \mathsf{c\_apply} \\
\overset{*}{\Longrightarrow} \langle\Xi, P, (S, W)\rangle & \text{By ind. hyp. on } \mathcal{D}_3.
\end{array}$$

The implementation of this case requires the append family defined above. Note how an appeal to the induction hypothesis is represented as a recursive call.

```
sc_app : subcomp (fev_app D3 D2 D1) C
          <- subcomp D1 C1
          <- subcomp D2 C2
          <- subcomp D3 C3
          <- append (c_app ~ C1) C2 C'
          <- append C' (c_apply ~ C3) C.
```

□

The first direction of Theorem 6.19 is a special case of this lemma. The other direction is more intricate. The basic problem is to extract a tree-structured evaluation from a linear computation. We must then show that this extraction will always succeed for complete computations. Note that it is obviously not possible to extract evaluations from arbitrary incomplete sequences of transitions of the abstract machine.

In order to write computation sequences more concisely, we introduce some notation. Let $R :: St \Longrightarrow St'$ and $\mathcal{C} :: St' \overset{*}{\Longrightarrow} St''$. Then we write

$$R \sim \mathcal{C} :: St \Longrightarrow St''$$

for the computation which begins with R and then proceeds with $\mathcal{C}$. Such a computation exists by the step inference rule. This corresponds directly to the notation in the Elf implementation.

For the proof of the central lemma of this section, we will need a new form of induction often referred to as *complete induction.* During a proof by complete induction we assume the induction hypothesis not only for the immediate premisses of the last inference rule, but for all proper subderivations. Intuitively, this is justified, since all proper subderivations are "smaller" than the given derivation. For a more formal discussion of the complete induction principle for derivations see Section 6.4. The judgment $\mathcal{C} < \mathcal{C}'$ ($\mathcal{C}$ is a *proper subcomputation of* $\mathcal{C}'$) is defined by the following inference rules.

$$\frac{}{\mathcal{C} < R \sim \mathcal{C}}\,\textsf{sub\_imm} \qquad\qquad \frac{\mathcal{C} < \mathcal{C}'}{\mathcal{C} < R \sim \mathcal{C}'}\,\textsf{sub\_med}$$

It is easy to see that the proper subcomputation relation is transitive.

Lemma 6.17 *If* $\mathcal{C}_1 < \mathcal{C}_2$ *and* $\mathcal{C}_2 < \mathcal{C}_3$ *then* $\mathcal{C}_1 < \mathcal{C}_3$.

Proof: By a simple induction (see Exercise 6.12). □

The implementation of this ordering and the proof of transitivity are immediate.

```
< : (st Ns1 P1 S1) =>* (st Ns P S)
     -> (st Ns2 P2 S2) =>* (st Ns P S)
     -> type.
%infix none 8 <.
sub_imm : C < R ~ C.

sub_med : C < C'
       -> C < R ~ C'.
```

The representation of the proof of transitivity is left to Exercise 6.12.

We are now prepared for the lemma that a complete computation with an appropriate initial state can be translated into an evaluation followed by another complete computation.

Lemma 6.18 *If*

$$\mathcal{C} :: \langle(\Xi, \eta), F \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle\cdot, \mathit{done}, (\cdot, W')\rangle$$

there exists a value W, an evaluation

$$\mathcal{D} :: \eta \vdash F \hookrightarrow W,$$

and a computation

$$\mathcal{C}' :: \langle \Xi, P, (S, W)\rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot, W')\rangle$$

such that $\mathcal{C}' < \mathcal{C}$.

Proof: By complete induction on the $\mathcal{C}$. We only show a few cases; the others similar. We use the abbreviation $\mathit{final} = \langle \cdot, \mathit{done}, (\cdot, W')\rangle$. The representing type family `spl` is indexed by three deductions: $\mathcal{C}$, $\mathcal{C}'$, and $\mathcal{D}$. We do not explicitly represent the derivation which shows that $\mathcal{C}' < \mathcal{C}$ since the Twelf system can check this automatically using a `%reduces` declaration, placed after all rules for `spl`.

```
spl : (st (Ns ; N) (ev F & P) S) =>* (st emptys done (empty , W'))
       -> feval N F W
       -> (st Ns P (S , W)) =>* (st emptys done (empty , W'))
       -> type.
%mode spl +C -D -C'.
... clauses for spl ...
%reduces C' < C (spl C _ C').
%terminates C (spl C _ _).
```

Now back to the proof.

Case: $\mathcal{C}$ begins with $\mathsf{c\_z}$, that is, $\mathcal{C} = \mathsf{c\_z} \sim \mathcal{C}_1$. Then $W = \mathbf{z}^*$,

$$\mathcal{D} = \frac{}{K \vdash \mathbf{z} \hookrightarrow \mathbf{z}^*}\,\mathsf{fev\_z},$$

and $\mathcal{C}' = \mathcal{C}_1$. Furthermore, $\mathcal{C}' = \mathcal{C}_1 < \mathsf{c\_z} \sim \mathcal{C}_1$ by rule $\mathsf{sub\_imm}$. The representation in Elf:

```
spl_z : spl (c_z ~ C1) (fev_z) C1.
```

Case: $\mathcal{C}$ begins with $\mathsf{c\_app}$. Then $\mathcal{C} = \mathsf{c\_app} \sim \mathcal{C}_1$ where

$$\mathcal{C}_1 :: \langle (\Xi; \eta; \eta), F_1 \,\&\, F_2 \,\&\, \mathit{apply} \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \mathit{final}.$$

By induction hypothesis on $\mathcal{C}_1$ there exists a W_1, an evaluation

$$\mathcal{D}_1 :: \eta \vdash F_1 \hookrightarrow W_1$$

and a computation

$$\mathcal{C}_2 :: \langle (\Xi; \eta), F_2 \,\&\, \mathit{apply} \,\&\, P, (S, W_1)\rangle \overset{*}{\Longrightarrow} \mathit{final}$$

such that $\mathcal{C}_2 < \mathcal{C}_1$. We can thus apply the induction hypothesis to $\mathcal{C}_2$ to obtain a W_2, an evaluation

$$\mathcal{D}_2 :: \eta \vdash F_2 \hookrightarrow W_2$$

and a computation

$$\mathcal{C}_3 :: \langle \Xi, \mathit{apply} \,\&\, P, (S, W_1, W_2)\rangle \overset{*}{\Longrightarrow} \mathit{final}$$

such that $\mathcal{C}_3 < \mathcal{C}_2$. By inversion, $\mathcal{C}_3 = \mathsf{c\_apply} \sim \mathcal{C}_3'$ and $W_1 = \{\eta'; \Lambda F_1'\}$ where

$$\mathcal{C}_3' :: \langle (\Xi; (\eta', W_2)), F_1' \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \mathit{final}.$$

Then $\mathcal{C}_3' < \mathcal{C}_3$ and by induction hypothesis on $\mathcal{C}_3'$ there is a value W_3, an evaluation

$$\mathcal{D}_3 :: \eta', W_2 \vdash F_1' \hookrightarrow W_3$$

and a compuation

$$\mathcal{C}_4 :: \langle \Xi, P, (S, W_3)\rangle \overset{*}{\Longrightarrow} \mathit{final}.$$

Now we let $W = W_3$ and we construct $\mathcal{D} :: \eta \vdash F_1\ F_2 \hookrightarrow W_3$ by an application of the rule fev_app to the premisses $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$. Furthermore we let $\mathcal{C}' = \mathcal{C}_4$ and conclude by some elementary reasoning concerning the subcomputation relation that $\mathcal{C}' < \mathcal{C}$.

The representation of this subcase of this case requires three explicit appeals to the transitivity of the subcomputation ordering. In order to make this at all intelligible, we use the name C2<C1 (one identifier) for the derivation that $\mathcal{C}_2 < \mathcal{C}_1$ and similarly for other such derivations.

```
spl_app : spl (c_app ~ C1)
            (fev_app D3 D2 D1) C4
             <- spl C1 D1 C2
             <- spl C2 D2 (c_apply ~ C3')
             <- spl C3' D3 C4.
```

□

Now we have all the essential lemmas to prove the main theorem.

Theorem 6.19 *$\eta \vdash F \hookrightarrow W$ is derivable iff $\eta \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\!\Rightarrow} W$ is derivable.*

Proof: By definition, $\eta \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\!\Rightarrow} W$ iff there is a computation

$$\mathcal{C} :: \langle (\cdot; \eta), F \,\&\, \mathit{done}, \cdot\rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot, W)\rangle.$$

One direction follows immediately from Lemma 6.16: if $\eta \vdash F \hookrightarrow W$ then

$$\langle (\Xi; \eta), F \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle \Xi, P, (S, W)\rangle$$

for *any* Ξ, P, and S and in particular for $\Xi = \cdot$, $P = \mathit{done}$ and $S = \cdot$. The implementation of this direction in Elf:

```
cev_complete : feval N F W -> ceval N F W -> type.
%mode cev_complete +D -C.
cevc : cev_complete D (run C) <- subcomp D C.
%terminates [] (cev_complete D _).
```

For the other direction, assume there is a deduction $\mathcal{C}$ of the form shown above. By Lemma 6.18 we know that there exist a W', an evaluation

$$\mathcal{D}' :: \eta \vdash F \hookrightarrow W'$$

and a computation

$$\mathcal{C}' :: \langle \cdot, done, (\cdot, W') \rangle \overset{*}{\Longrightarrow} \langle \cdot, done, (\cdot, W) \rangle$$

such that $\mathcal{C}' < \mathcal{C}$. Since there is no transition rule for the program *done*, $\mathcal{C}'$ must be id and $W = W'$. Thus $\mathcal{D} = \mathcal{D}'$ fulfills the requirements of the theorem. This is implemented as follows.

```
cls_sound : ceval N F W -> feval N F W -> type.
%mode cls_sound +C -D.
clss : cls_sound (run C) D <- spl C D (id).
%terminates [] (cls_sound C _).
```

□

6.4 Complete Induction over Computations

Here we briefly justify the principle of complete induction used in the proof of Lemma 6.18. We repeat the definition of proper subcomputations and also define a general subcomputation judgment which will be useful in the proof.

$\mathcal{C} < \mathcal{C}'$ $\mathcal{C}$ is a proper subcomputation of $\mathcal{C}'$, and
$\mathcal{C} \leq \mathcal{C}'$ $\mathcal{C}$ is a subcomputation of $\mathcal{C}'$.

These judgments are defined via the following inference rules.

$$\frac{}{\mathcal{C} < R \sim \mathcal{C}}\,\mathsf{sub\_imm} \qquad \frac{\mathcal{C} < \mathcal{C}'}{\mathcal{C} < R \sim \mathcal{C}'}\,\mathsf{sub\_med}$$

$$\frac{\mathcal{C} < \mathcal{C}'}{\mathcal{C} \leq \mathcal{C}'}\,\mathsf{leq\_sub} \qquad \frac{}{\mathcal{C} \leq \mathcal{C}}\,\mathsf{leq\_eq}$$

We only need one simple lemma regarding the subcomputation judgment.

Lemma 6.20 *If $\mathcal{C} \leq \mathcal{C}'$ is derivable, then $\mathcal{C} < R \sim \mathcal{C}'$ is derivable.*

Proof: By analyzing the two possibilities for the deduction of the premiss and constructing an immediate deduction for the conclusion in each case. □

We call a property P of computations *complete* if it satisfies:

> For every $\mathcal{C}$, the assumption that P holds for all $\mathcal{C}' < \mathcal{C}$ implies that P holds for $\mathcal{C}$.

Theorem 6.21 (Principle of Complete Induction over Computations) *If a property P of computations is complete, then P holds for all computations.*

Proof: We assume that P is complete and then prove by ordinary structural induction that for every $\mathcal{C}$ and for every $\mathcal{C}' \leq \mathcal{C}$, P holds of $\mathcal{C}$.

Case: $\mathcal{C} = id$. By inversion, there is no $\mathcal{C}'$ such that $\mathcal{C}' < id$. Thus P holds for all $\mathcal{C}' < id$. Since P is complete, this implies that P holds for id.

Case: $\mathcal{C} = R \sim \mathcal{C}_1$. The induction hypothesis states that for every $\mathcal{C}_1' \leq \mathcal{C}_1$, P holds of $\mathcal{C}_1'$. We have to show that for every $\mathcal{C}_2 \leq R \sim \mathcal{C}_1$, property P holds of $\mathcal{C}_2$. By inversion, there are two subcases, depending on the evidence for $\mathcal{C}_2 \leq R \sim \mathcal{C}_1$.

Subcase: $\mathcal{C}_2 = R \sim \mathcal{C}_1$. The induction hypothesis and Lemma 6.20 yield that for every $\mathcal{C}_1' < R \sim \mathcal{C}_1$, P holds of $\mathcal{C}_1$. Since P is complete, P must thus hold for $R \sim \mathcal{C}_1 = \mathcal{C}_2$.

Subcase: $\mathcal{C}_2 < R \sim \mathcal{C}_1$. Then by inversion either $\mathcal{C}_1 = \mathcal{C}_2$ or $\mathcal{C}_1 < \mathcal{C}_2$. In either case $\mathcal{C}_2 \leq \mathcal{C}_1$ by one inference. Now we can apply the induction hypothesis to conclude that P holds of $\mathcal{C}_2$.

□

6.5 A Continuation Machine

The natural semantics for Mini-ML presented in Chapter 2 is called a *big-step semantics*, since its only judgment relates an expression to its final value—a "big step". There are a variety of properties of a programming language which are difficult or impossible to express as properties of a big-step semantics. One of the central ones is that "well-typed programs do not go wrong". Type preservation, as proved in Section 2.6, does not capture this property, since it presumes that we are already given a complete evaluation of an expression e to a final value v and then relates the types of e and v. This means that despite the type preservation theorem, it is possible that an attempt to find a value of an expression e leads to an intermediate expression such as **fst z** which is ill-typed and to which no evaluation rule applies. Furthermore, a big-step semantics does not easily permit an analysis of non-terminating computations.

An alternative style of language description is a *small-step semantics.* The main judgment in a small-step operational semantics relates the state of an abstract machine (which includes the expression to be evaluated) to an immediate successor state. These small steps are chained together until a value is reached. This level of description is usually more complicated than a natural semantics, since the current state must embody enough information to determine the next and all remaining computation steps up to the final answer. It is also committed to the order in which subexpressions are evaluated and thus somewhat less abstract than a natural, big-step semantics.

In this section we construct a machine directly from the original natural semantics of Mini-ML in Section 2.3 (and not from the environment-based semantics in Section 6.1). This illustrates the general technique of *continuations* to sequentialize computations. Another application of the technique at the level of expressions (rather than computations) is given in Section **??**.

Our goal now is define a small-step semantics. For this, we isolate an expression e to be evaluated, and a *continuation* K which contains enough information to carry out the rest of the evaluation necessary to compute the overall value. For example, to evaluate a pair $\langle e_1, e_2 \rangle$ we first compute the value of e_1, remembering that the next task will be the evaluation of e_2, after which the two values have to be paired. This also shows the need for intermediate *instructions*, such as "*evaluate the second element of a pair*" or "*combine two values into a pair*". One particular kind of instruction, written as **ev** e, triggers the first step in the computation based on the structure of e.

Because we always fully evaluate one expression before moving on to the next, the continuation has the form of a stack. Because the result of evaluating the current expression must be communicated to the continuation, each item on the stack is a function from values to instructions. Finally, when we have computed a value, we *return* it by applying the first item on the continuation stack. Thus the following structure emerges, to be supplement by further auxiliary instructions as necessary.

$$\begin{array}{rcl} \text{Instructions} & i & ::= & \mathbf{ev}\, e \mid \mathbf{return}\, v \mid \ldots \\ \text{Continuations} & K & ::= & \mathbf{init} \mid K; \lambda x.\, i \\ \text{Machine States} & S & ::= & K \diamond i \mid \mathbf{answer}\, v \end{array}$$

Here, **init** is the initial continuation, indicating that nothing further remains to be done. The machine state **answer** v represents the final value of a computation sequence. Based on the general consideration, we have the following transitions of the abstract machine.

$$S \Longrightarrow S' \quad S \text{ goes to } S' \text{ in one computation step}$$

$$\frac{}{\mathbf{init} \diamond \mathbf{return}\; v \Longrightarrow \mathbf{answer}\; v}\;\textsf{st\_init}$$

$$\frac{}{K; \lambda x.\, i \diamond \mathbf{return}\; v \Longrightarrow K \diamond [v/x]i}\;\textsf{st\_return}$$

Further rules arise from considering each expression constructor in turn, possibly adding new special-purpose intermediate instructions. We will write the rules in the form *label* :: $S \Longrightarrow S'$ as a more concise alternative to the format used above. The meaning, however, remains the same: each rules is an axiom defining the transition judgment. First, the constructors:

$$\begin{array}{lllllll}
\textsf{st\_z} & :: & K & \diamond & \mathbf{ev}\;\mathbf{z} & \Longrightarrow & K & \diamond & \mathbf{return}\;\mathbf{z} \\
\textsf{st\_s} & :: & K & \diamond & \mathbf{ev}\;(\mathbf{s}\;e) & \Longrightarrow & K; \lambda x.\, \mathbf{return}\;(\mathbf{s}\;x) & \diamond & e
\end{array}$$

Second, the corresponding destructor:

$$\begin{array}{llllll}
\textsf{st\_case} & :: & K & \diamond & \mathbf{ev}\;(\mathbf{case}\; e_1 \;\mathbf{of}\; \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\; x \Rightarrow e_3) & \\
 & & & & \quad\Longrightarrow K; \lambda x_1.\, \mathbf{case}_1\; x_1 \;\mathbf{of}\; \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\; x \Rightarrow e_3 \;\diamond\; \mathbf{ev}\; e_1 & \\
\textsf{st\_case}_1\textsf{\_z} & :: & K & \diamond & \mathbf{case}_1\; \mathbf{z} \;\mathbf{of}\; \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\; x \Rightarrow e_3 \Longrightarrow K \diamond \mathbf{ev}\; e_2 & \\
\textsf{st\_case}_1\textsf{\_s} & :: & K & \diamond & \mathbf{case}_1\; \mathbf{s}\; v_1 \;\mathbf{of}\; \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\; x \Rightarrow e_3 \Longrightarrow K \diamond \mathbf{ev}\; [v_1'/x]e_3 &
\end{array}$$

We can see that the **case** construct requires a new instruction of the form $\mathbf{case}_1\; v_1 \;\mathbf{of}\; \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\; x \Rightarrow e_3$. This is distinct from $\mathbf{case}\; e_1 \;\mathbf{of}\; \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\; x \Rightarrow e_3$ in that the case subject is known to be a value. Without an explicit new construct, computation could get into an infinite loop since every value is also an expression which evaluates to itself. It should now be clear how pairs and projections are computed; the new instructions are $\langle v_1, e_2\rangle_1$, $\mathbf{fst}_1$, and $\mathbf{snd}_1$.

$$\begin{array}{lllllllll}
\textsf{st\_pair} & :: & K & \diamond & \mathbf{ev}\;\langle e_1, e_2\rangle & \Longrightarrow & K; \lambda x_1.\, \langle x_1, e_2\rangle_1 & \diamond & \mathbf{ev}\; e_1 \\
\textsf{st\_pair}_1 & :: & K & \diamond & \langle v_1, e_2\rangle_1 & \Longrightarrow & K; \lambda x_1.\, \mathbf{return}\,\langle v_1, x_2\rangle & \diamond & \mathbf{ev}\; e_2 \\
\textsf{st\_fst} & :: & K & \diamond & \mathbf{ev}\;(\mathbf{fst}\; e) & \Longrightarrow & K; \lambda x.\, \mathbf{fst}_1\; x & \diamond & \mathbf{ev}\; e \\
\textsf{st\_fst}_1 & :: & K & \diamond & \mathbf{fst}_1\;\langle v_1, v_2\rangle & \Longrightarrow & K & \diamond & \mathbf{return}\; v_1 \\
\textsf{st\_snd} & :: & K & \diamond & \mathbf{ev}\;(\mathbf{snd}\; e) & \Longrightarrow & K; \lambda x.\, \mathbf{snd}_1\; x & \diamond & \mathbf{ev}\; e \\
\textsf{st\_snd}_1 & :: & K & \diamond & \mathbf{snd}_1\;\langle v_1, v_2\rangle & \Longrightarrow & K & \diamond & \mathbf{return}\; v_2
\end{array}$$

Neither functions, nor definitions or recursion introduce any essentially new ideas. We add two new forms of instructions, $\mathbf{app}_1$ and $\mathbf{app}_2$, for the intermediate forms while evaluating applications.

$$\begin{array}{lllllll}
\textsf{st\_lam} & :: & K & \diamond & \mathbf{ev}\;(\mathbf{lam}\; x.\, e) & \Longrightarrow & K \diamond \mathbf{return}\;\mathbf{lam} x.\, e \\
\textsf{st\_app} & :: & K & \diamond & \mathbf{ev}\;(e_1\; e_2) & \Longrightarrow & K; \lambda x_1.\, \mathbf{app}_1\; x_1\; e_2 \diamond \mathbf{ev}\; e_1 \\
\textsf{st\_app}_1 & :: & K & \diamond & \mathbf{app}_1\; v_1\; e_2 & \Longrightarrow & K; \lambda x_2.\, \mathbf{app}_2\; v_1\; x_2 \diamond \mathbf{ev}\; e_2 \\
\textsf{st\_app}_2 & :: & K & \diamond & \mathbf{app}_2\;(\mathbf{lam}\; x.\, e_1')\; v_2 & \Longrightarrow & K \diamond \mathbf{ev}\;([v_2/x]e_1') \\
\\
\textsf{st\_letv} & :: & K & \diamond & \mathbf{let\,val}\; x = e_1 \;\mathbf{in}\; e_2 & \Longrightarrow & K; \lambda x_1.\, [x_1/x]e_2 \diamond \mathbf{ev}\; e_1 \\
\textsf{st\_letn} & :: & K & \diamond & \mathbf{let\,name}\; u = e_1 \;\mathbf{in}\; e_2 & \Longrightarrow & K \diamond \mathbf{ev}\;([e_1/x]e_2) \\
\textsf{st\_fix} & :: & K & \diamond & \mathbf{fix}\; u.\, e & \Longrightarrow & K \diamond \mathbf{ev}\;([\mathbf{fix}\; u.\, e/u]e)
\end{array}$$

The complete set of instructions as extracted from the transitions above:

$$\begin{array}{lcll}
\text{Instructions} & i \ ::= & \mathbf{ev}\ e \mid \mathbf{return}\ v & \\
 & & \mid \mathbf{case}_1\ v_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2\ \mid \mathbf{s}\ x \Rightarrow e_3 & \text{Natural numbers} \\
 & & \mid \langle v_1, e_2 \rangle_1 \mid \mathbf{fst}_1\ v \mid \mathbf{snd}_1\ v & \text{Pairs} \\
 & & \mid \mathbf{app}_1\ v_1\ e_2 \mid \mathbf{app}_2\ v_1\ v_2 & \text{Functions}
\end{array}$$

The implementation of instructions, continuations, and machine states in Elf uses infix operations to make continuations and states more readable.

```
% Machine Instructions
inst : type.  %name inst I.

ev : exp -> inst.
return : exp -> inst.

case1 : exp -> exp -> (exp -> exp) -> inst.
pair1 : exp -> exp -> inst.
fst1 : exp -> inst.
snd1 : exp -> inst.
app1 : exp -> exp -> inst.
app2 : exp -> exp -> inst.

% Continuations
cont : type.  %name cont K.

init : cont.
;    : cont -> (exp -> inst) -> cont.
%infix left 8 ;.

% Continuation Machine States
state : type.  %name state S.

# : cont -> inst -> state.
answer : exp -> state.
%infix none 7 #.
```

The following declarations constitute a direct translation of the transition rules above.

```
=> : state -> state -> type.          %name => C.
%infix none 6 =>.
%mode => +S -S'.
```

```
% Natural Numbers
st_z : K # (ev z) => K # (return z).
st_s : K # (ev (s E')) => (K ; [x'] return (s x')) # (ev E').
st_case : K # (ev (case E1 E2 E3)) => (K ; [x1] case1 x1 E2 E3) # (ev E1).
st_case1_z : K # (case1 (z) E2 E3) => K # (ev E2).
st_case1_s : K # (case1 (s V1') E2 E3) => K # (ev (E3 V1')).

% Pairs
st_pair : K # (ev (pair E1 E2)) => (K ; [x1] pair1 x1 E2) # (ev E1).
st_pair1 : K # (pair1 V1 E2) => (K ; [x2] return (pair V1 x2)) # (ev E2).
st_fst : K # (ev (fst E')) => (K ; [x'] fst1 x') # (ev E').
st_fst1 : K # (fst1 (pair V1 V2)) => K # (return V1).
st_snd : K # (ev (snd E')) => (K ; [x'] snd1 x') # (ev E').
st_snd1 : K # (snd1 (pair V1 V2)) => K # (return V2).

% Functions
st_lam : K # (ev (lam E')) => K # (return (lam E')).
st_app : K # (ev (app E1 E2)) => (K ; [x1] app1 x1 E2) # (ev E1).
st_app1 : K # (app1 V1 E2) => (K ; [x2] app2 V1 x2) # (ev E2).
st_app2 : K # (app2 (lam E1') V2) => K # (ev (E1' V2)).

% Definitions
st_letv : K # (ev (letv E1 E2)) => (K ; [x1] ev (E2 x1)) # (ev E1).
st_letn : K # (ev (letn E1 E2)) => K # (ev (E2 E1)).

% Recursion
st_fix : K # (ev (fix E')) => K # (ev (E' (fix E'))).

% Return Instructions
st_return : (K ; [x] I x) # (return V) => K # (I V).
st_init : (init) # (return V) => (answer V).
```

Multi-step computation sequences could be represented as lists of single step transitions. However, we would like to use dependent types to guarantee that, in a valid computation sequence, the result state of one transition matches the start state of the next transition. This is difficult to accomplish using a generic type of lists; instead we introduce specific instances of this type which are structurally just like lists, but have strong internal validity conditions.

$S \overset{*}{\Longrightarrow} S'$	S goes to S' in zero or more steps
$e \overset{c}{\hookrightarrow} v$	e evaluates to v using the continuation machine

$$\frac{}{S \overset{*}{\Longrightarrow} S}\,\mathsf{stop} \qquad \frac{S \Longrightarrow S' \qquad S' \overset{*}{\Longrightarrow} S''}{S \overset{*}{\Longrightarrow} S''}\,\mathsf{step}$$

$$\frac{\mathbf{init} \diamond \mathbf{ev}\ e \overset{*}{\Longrightarrow} \mathbf{answer}\ v}{e \overset{c}{\hookrightarrow} v}\,\mathsf{cev}$$

We would like the implementation to be operational, that is, queries of the form `?- ceval` $\ulcorner e \urcorner$ `V.` should compute the value `V` of a given e. This means the $S \Longrightarrow S'$ should be the first subgoal and hence the second argument of the step rule. In addition, we employ a visual trick to display computation sequences in a readable format by representing the step rule as a left associative infix operator.

```
=>* : state -> state -> type.        %name =>* C*.
%infix none 5 =>*.
%mode =>* +S -S'.

stop : S =>* S.
<< : S =>* S''
      <- S => S'
      <- S' =>* S''.
%infix left 5 <<.
% Because of evaluation order, computation sequences are displayed
% in reverse, using "<<" as a left-associative infix operator.

ceval : exp -> exp -> type.          %name ceval CE.
%mode ceval +E -V.

cev : ceval E V
       <- (init) # (ev E) =>* (answer V).
```

We then get a reasonable display of the sequence of computation steps which must be read from right to left.

```
?- C* : init # (ev (app (lam [x] x) z)) =>* answer V.
Solving...
V = z.
C* =
 stop << st_init << st_z << st_app2 << st_return << st_z << st_app1
      << st_return << st_lam << st_app.
```

The overall task now is to prove that $e \hookrightarrow v$ if and only if $e \overset{c}{\hookrightarrow} v$. In one direction we have to find a translation from tree-structured derivations $\mathcal{D} :: e \hookrightarrow v$ to sequential computations $\mathcal{C} :: \mathbf{init} \diamond \mathbf{ev}\ e \overset{*}{\Longrightarrow} \mathbf{answer}\ v$. In the other direction we have to find a way to chop a sequential computation into pieces which can be reassembled into a tree-structured derivation.

We start with the easier of the two proofs. We assume that $e \hookrightarrow v$ and try to show that $e \overset{c}{\hookrightarrow} v$. This immediately reduces to showing that $\mathbf{init} \diamond \mathbf{ev}\ e \overset{*}{\Longrightarrow} \mathbf{answer}\ v$. This does not follow directly by induction, since subcomputations will neither start from the initial computation nor return the final answer. If we generalize the claim to state that for all continuations K we have that $K \diamond \mathbf{ev}\ e \overset{*}{\Longrightarrow} K \diamond \mathbf{return}\ v$, then it follows directly by induction, using some simple lemmas regarding the concatenation of computation sequences (see Exercise 6.17).

We can avoid explicit concatenation of computation sequences and obtain a more direct proof (and more efficient program) if we introduce an *accumulator argument.* This argument contains the remainder of the computation, starting from the state $K \diamond \mathbf{return}\ v$. To the front of this given computation we add the computation from $K \diamond \mathbf{ev}\ e \overset{*}{\Longrightarrow} K \diamond \mathbf{return}\ v$, passing the resulting computation as the next value of the accumulator argument. Translating this intuition to a logical statement requires explicitly universally quantifying over the accumulator argument.

Lemma 6.22 *For any closed expression e, value v and derivation $\mathcal{D} :: e \hookrightarrow v$, if $\mathcal{C}' :: K \diamond \mathbf{return}\ v \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ for any K and w, then $\mathcal{C} :: K \diamond \mathbf{ev}\ e \overset{*}{\Longrightarrow} \mathbf{answer}\ w$.*

Proof: The proof proceeds by induction on the structure of $\mathcal{D}$. Since the accumulator argument must already hold the remainder of the overall computation upon appeal to the induction hypothesis, we apply the induction hypothesis on the immediate subderivations of $\mathcal{D}$ in right-to-left order.

The proof is implemented by a type family

```
ccp : eval E V
       -> K # (return V) =>* (answer W)
       -> K # (ev E) =>* (answer W)
       -> type.
%mode ccp +D +C' -C.
```

Operationally, the first argument is the induction argument, the second argument the accumulator, and the last the output argument.

We only show a couple of cases in the proof; the others follow in a similar manner.

Case:

$$\mathcal{D} = \frac{}{\mathbf{lam}\ x.\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1}\ \textsf{ev\_lam}$$

$\mathcal{C}' :: K \diamond \mathbf{return}\ \mathbf{lam}\ x.\ e_1 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$	Assumption
$\mathcal{C} :: K \diamond \mathbf{ev}(\mathbf{lam}\ x.\ e_1) \Longrightarrow \mathbf{answer}\ w$	By st_lam followed by $\mathcal{C}'$

In this case we have added a step st_lam to a computation; in the implementation, this will be an application of the step rule for the $S \overset{*}{\Longrightarrow} S'$ judgment, which is written as << in infix notation. Recall that the reversal of the evaluation order means that computations (visually) proceed from right to left.

```
ccp_lam : ccp (ev_lam) C' (C' << st_lam).
```

Case:

$$\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1'\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ e_2 \hookrightarrow v_2\end{matrix} \qquad \begin{matrix}\mathcal{D}_3 \\ [v_2/x]e_1' \hookrightarrow v\end{matrix}}{e_1\ e_2 \hookrightarrow v}\ \mathsf{ev\_app}$$

$\mathcal{C}' :: K \diamond \mathbf{return}\ v \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ Assumption

$\mathcal{C}_3 :: K \diamond \mathbf{ev}\ ([v_2/x]e_1') \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By ind. hyp. on $\mathcal{D}_3$ and $\mathcal{C}'$

$\mathcal{C}_2' :: K; \lambda x_2.\ \mathbf{app}_2\ (\mathbf{lam}\ x.\ e_1')\ x_2 \diamond \mathbf{return}\ v_2 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$
By st_return and st_app2 followed by $\mathcal{C}_3$

$\mathcal{C}_2 :: K; \lambda x_2.\ \mathbf{app}_2\ (\mathbf{lam}\ x.\ e_1')\ x_2 \diamond \mathbf{ev}\ e_2 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$
By ind. hyp. on $\mathcal{D}_2$ and $\mathcal{C}_2'$

$\mathcal{C}_1' :: K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{return}\ \mathbf{lam}\ x.\ e_1' \overset{*}{\Longrightarrow} \mathbf{answer}\ w$
By st_return and st_app1 followed by $\mathcal{C}_2$.

$\mathcal{C}_1 :: K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{ev}\ e_1 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By ind. hyp. on $\mathcal{D}_1$ and $\mathcal{C}_1'$

$\mathcal{C} :: K \diamond \mathbf{ev}\ (e_1\ e_2) \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By st_app followed by $\mathcal{C}_1$.

The implementation threads the accumulator argument, adding steps concerned with application as in the proof above.

```
ccp_app : ccp (ev_app D3 D2 D1) C' (C1 << st_app)
           <- ccp D3 C' C3
           <- ccp D2 (C3 << st_app2 << st_return) C2
           <- ccp D1 (C2 << st_app1 << st_return) C1.
```

□

From this, the completeness of the abstract machine follows directly.

Theorem 6.23 (Completeness of the Continuation Machine) *For any closed expression e and value v, if $e \hookrightarrow v$ then $e \overset{c}{\hookrightarrow} v$.*

Proof: We use Lemma 6.22 with $K = \mathbf{init}$, $w = v$, and $\mathcal{C}'$ the computation with st_init as the only step, to conclude that there is a computation $\mathcal{C} :: \mathbf{init} \diamond \mathbf{ev}\ e \overset{*}{\Longrightarrow} \mathbf{answer}\ v$. Therefore, by rule cev, $e \overset{c}{\hookrightarrow} v$.

The implementation is straightforward, using ccp, the implementation of the main lemma above.

```
cpm_complete : eval E V -> ceval E V -> type.
%mode cpm_complete +D -C.
cpmcp : cpm_complete D (cev C)
         <- ccp D (stop << st_init) C.
```

□

Now we turn our attention to the soundness of the continuation machine: whenever it produces a value v then the natural semantics can also produce the value v from the same expression. This is more difficult to prove than completeness. The reason is that in the completeness proof, every subderivation of $\mathcal{D} :: e \hookrightarrow v$ can inductively be translated to a sequence of computation steps, but not every sequence of computation steps corresponds to an evaluation. For example, the partial computation

$$K \diamond \mathbf{ev}\ (e_1\ e_2) \overset{*}{\Longrightarrow} K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{ev}\ e_1$$

represents only a fragment of an evaluation. In order to translate a computation sequence we must ensure that it is sufficiently long. A simple way to accomplish this is to require that the given computation goes all the way to a final answer. Thus, we have a state $K \diamond \mathbf{ev}\ e$ at the beginning of a computation sequence $\mathcal{C}$ to a final answer w, there must be some initial segment of $\mathcal{C}'$ which corresponds to an evaluation of e to a value v, while the remaining computation goes from $K \diamond \mathbf{return}\ v$ to the final answer w. This can then be proved by induction.

Lemma 6.24 *For any continuation K, closed expression e and value w, if $\mathcal{C} :: K \diamond \mathbf{ev}\ e \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ then there is a value v a derivation $\mathcal{D} :: e \hookrightarrow v$, and a subcomputation $\mathcal{C}'$ of $\mathcal{C}$ of the form $K \diamond \mathbf{return}\ v \overset{*}{\Longrightarrow} \mathbf{answer}\ w$.*

Proof: By complete induction on the structure of $\mathcal{C}$. Here *complete induction*, as opposed to a simple structural induction, means that we can apply the induction hypothesis to any subderivation of $\mathcal{C}$, not just to the immediate subderivations. It should be intuitively clear that this is a valid induction principle (see also Section 6.4).

In the implementation we have chosen not to represent the evidence for the assertion that $\mathcal{C}'$ is a subderivation of $\mathcal{C}$. This can be added, either directly to the implementation or as a higher-level judgment (see Exercise **??**). This information is not required to execute the proof on specific computation sequences, although it is critical for seeing that it always terminates.

```
csd : K # (ev E) =>* (answer W)
       -> eval E V
       -> K # (return V) =>* (answer W)
       -> type.
%mode csd +C -D -C'.
```

We only show a few typical cases; the others follow similarly.

Case: The first step of $\mathcal{C}$ is st_lam followed by $\mathcal{C}_1 :: K \diamond \textbf{return lam } x.\ e \overset{*}{\Longrightarrow} \textbf{answer } w$.

In this case we let $\mathcal{D} = $ ev_lam and $\mathcal{C}' = \mathcal{C}_1$. The implementation (where step is written as << in infix notation):

```
csd_lam : csd (C' << st_lam) (ev_lam) C'.
```

Case: The first step of $\mathcal{C}$ is st_app followed by $\mathcal{C}_1 :: K; \lambda x_1.\ \textbf{app}_1\ x_1\ e_2 \diamond \textbf{ev}\ e_1 \overset{*}{\Longrightarrow} \textbf{answer } w$, where $e = e_1\ e_2$.

$\mathcal{D}_1 :: e_1 \hookrightarrow v_1$ for some v_1 and
$\mathcal{C}_1' :: K; \lambda x_1.\ \textbf{app}_1\ x_1\ e_2 \diamond \textbf{return}\ v_1 \overset{*}{\Longrightarrow} \textbf{answer } w$ By ind. hyp. on $\mathcal{C}_1$
$\mathcal{C}_1'' :: K \diamond \textbf{app}_1\ v_1\ e_2 \overset{*}{\Longrightarrow} \textbf{answer } w$ By inversion on $\mathcal{C}_1'$
$\mathcal{C}_2 :: K; \lambda x_2.\ \textbf{app}_2\ v_1\ x_2 \diamond \textbf{ev}\ e_2 \overset{*}{\Longrightarrow} \textbf{answer } w$ By inversion on $\mathcal{C}_1''$
$\mathcal{D}_2 :: e_2 \hookrightarrow v_2$ form some v_2 and
$\mathcal{C}_2' :: K; \lambda x_2.\ \textbf{app}_2\ v_1\ x_2 \diamond \textbf{return}\ v_2 \overset{*}{\Longrightarrow} \textbf{answer } w$ By ind. hyp. on $\mathcal{C}_2$
$\mathcal{C}_2'' :: K \diamond \textbf{app}_2\ v_1\ v_2 \overset{*}{\Longrightarrow} \textbf{answer } w$ By inversion on $\mathcal{C}_2'$
$v_1 = \textbf{lam}\ x.\ e_1'$ and
$\mathcal{C}_3 :: K \diamond \textbf{ev}\ ([v_2/x]e_1') \overset{*}{\Longrightarrow} \textbf{answer } w$ By inversion on $\mathcal{C}_2''$
$\mathcal{D}_3 :: [v_2/x]e_1' \hookrightarrow v$ for some v and
$\mathcal{C}' :: K \diamond \textbf{return}\ v \overset{*}{\Longrightarrow} \textbf{answer } w$ By ind. hyp on $\mathcal{C}_3$
$\mathcal{D} :: e_1\ e_2 \hookrightarrow v$ By rule ev_app from $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$.

The evaluation $\mathcal{D}$ and computation sequence $\mathcal{C}'$ now satisfy the requirements of the lemma. The appeals to the induction hypothesis are all legal, since $\mathcal{C} > \mathcal{C}_1 > \mathcal{C}_1'' > \mathcal{C}_2 > \mathcal{C}_2' > \mathcal{C}_2'' > \mathcal{C}_3 > \mathcal{C}'$, where $>$ is the subcomputation judgment. Each of the subcomputation judgments in this chain follows either immediately, or by induction hypothesis.

The implementation:

```
csd_app : csd (C1 << st_app) (ev_app D3 D2 D1) C'
           <- csd C1 D1 (C2 << st_app1 << st_return)
           <- csd C2 D2 (C3 << st_app2 << st_return)
           <- csd C3 D3 C'.
```

□

Once again, the main theorem follows directly from the lemma.

Theorem 6.25 (Soundness of the Continuation Machine) *For any closed expression e and value v, if $e \overset{c}{\hookrightarrow} v$ then $e \hookrightarrow v$.*

Proof: By inversion, $\mathcal{C} :: \textbf{init} \diamond \textbf{ev}\ e \overset{*}{\Longrightarrow} \textbf{answer}\ v$. By Lemma 6.24 there is a derivation $\mathcal{D} :: e \hookrightarrow v'$ and $\mathcal{C}' :: \textbf{init} \diamond \textbf{return}\ v' \overset{*}{\Longrightarrow} \textbf{answer}\ v$ for some v'. By inversion on $\mathcal{C}'$ we see that $v = v'$ and therefore $\mathcal{D}$ satisfies the requirements of the theorem.

```
cpm_sound : ceval E V -> eval E V -> type.
%mode cpm_sound +C -D.

cpmsd : cpm_sound (cev C) D
         <- csd C D (stop << st_init).

%terminates {} (cpm_sound C D).
```

□

6.6 Type Preservation and Progress

So far we have concentrated on operational aspects of the translation from a big-step to a small-step semantics; now we turn to issues of typing. The first property is type preservation—a reprise of the same property for the big-step semantics. But the reformulation of the semantics also allows us to express new language properties, still at a high level of abstraction. One of the most important ones is *progress.* It states that in any valid abstract machine state we can have one of two situations: either we have already computed the final answer of the program, or we can make progress by taking a further step.

Type preservation and progress together express that *well-typed programs cannot go wrong*, a phrase coined by Milner [Mil78]. In our setting, "*going wrong*" corresponds to reaching a machine state in which no further transition rules applicable. Note that type preservation for the big-step semantics does *not* express this, since it only talks about completed evaluations, not about intermediate states. Of course, even in the small-step semantics an expression can fail to have a value, but from the progress theorem we know that this is only due to non-termination.

We begin by giving the typing rules for the continuation-passing machine. One complication as compared to the typing rules for expressions is that certain machine states only make sense when components of instructions are values. For example, the instruction $\textbf{app}_2\ v_1\ v_2$ requires both arguments to be values. If this restriction is not enforced, the progress theorem clearly fails, because there is no transition for a state $K \diamond \textbf{app}_2\ ((\textbf{lam}\ x.\ x)\,(\textbf{lam}\ y.\ y))\ \textbf{z}$ even though the arguments to $\textbf{app}_2$ are correctly typed. This introduces a further complication: since continuations are composed of functions from values to instructions, we need to record that the argument of the continuation is indeed a function. For this we introduce a new context Υ, which is either empty or contains a simple hypothesis $x\ \mathit{Value}$. The value judgment is appropriately generalized so that $x\ \mathit{Value} \rhd x\ \mathit{Value}$.

$\Upsilon ::= \cdot \mid x\ \mathit{Value}$	Continuation argument
$\Upsilon; \Delta \rhd i : \tau$	Instruction i has type τ in context Δ.
$\rhd K : \tau \Rightarrow \sigma$	Continuation K maps values of type τ to answers of type σ
$\rhd S : \sigma$	State S has type σ

Typing for continuations keeps track of two types: the type of the value that will be passed to it, and the type of the final answer it produces. States on the other hand record only the type of final answer it may produce. The judgments for continuations and state do not depend on a context because they never contain free variables.

$$\frac{\Delta \rhd e : \tau}{\Upsilon; \Delta \rhd \mathbf{ev}\ e : \tau}\,\mathsf{vi\_ev} \qquad \frac{\Delta \rhd v : \tau \qquad \Upsilon \rhd v\ \mathit{Value}}{\Upsilon; \Delta \rhd \mathbf{return}\ v : \tau}\,\mathsf{vi\_return}$$

$$\frac{\Delta \rhd v_1 : \mathbf{nat} \qquad \Delta \rhd e_2 : \tau \qquad \Delta, x{:}\mathbf{nat} \rhd e_2 : \tau \qquad \Upsilon \rhd v_1\ \mathit{Value}}{\Upsilon; \Delta \rhd (\mathbf{case}_1\ v_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2\ \mid \mathbf{s}\ x \Rightarrow e_3) : \tau}\,\mathsf{vi\_case_1}$$

$$\frac{\Delta \rhd v_1 : \tau_1 \qquad \Delta \rhd e_2 : \tau_2 \qquad \Upsilon \rhd v_1\ \mathit{Value}}{\Upsilon; \Delta \rhd \langle v_1, e_2\rangle_1 : \tau_1 \times \tau_2}\,\mathsf{vi\_pair_1}$$

$$\frac{\Delta \rhd v' : \tau_1 \times \tau_2 \qquad \Upsilon \rhd v'\ \mathit{Value}}{\Upsilon; \Delta \rhd \mathbf{fst}_1\ v' : \tau_1}\,\mathsf{vi\_fst_1} \qquad \frac{\Delta \rhd v' : \tau_1 \times \tau_2 \qquad \Upsilon \rhd v'\ \mathit{Value}}{\Upsilon; \Delta \rhd \mathbf{snd}_1\ v' : \tau_2}\,\mathsf{vi\_snd_1}$$

$$\frac{\Delta \rhd v_1 : \tau_2 \to \tau_1 \qquad \Delta \rhd e_2 : \tau_2 \qquad \Upsilon \rhd v_1\ \mathit{Value}}{\Upsilon; \Delta \rhd \mathbf{app}_1\ v_1\ e_2 : \tau_1}\,\mathsf{vi\_app_1}$$

$$\frac{\Delta \rhd v_1 : \tau_2 \to \tau_1 \qquad \Delta \rhd v_2 : \tau_2 \qquad \Upsilon \rhd v_1\ \mathit{Value} \qquad \Upsilon \rhd v_2\ \mathit{Value}}{\Upsilon; \Delta \rhd \mathbf{app}_2\ v_1\ v_2 : \tau_1}\,\mathsf{vi\_app_2}$$

In the typing rules for continuations, we have to make sure that the parts of the continuation are composed properly: the value returned by the last instruction of a continuation matches the type accepted by the remaining continuation. The initial continuation just returns its argument as the final answer and therefore has type $\tau \Rightarrow \tau$.

$$\frac{}{\triangleright\ \mathbf{init} : \tau \Rightarrow \tau}\ \textsf{vk\_init}$$

$$\frac{x\ \mathit{Value}; \cdot, x{:}\tau \triangleright i : \tau' \qquad \triangleright K : \tau' \Rightarrow \sigma}{\triangleright K; \lambda x.\, i : \tau \Rightarrow \sigma}\ \textsf{vk\_;}$$

For a state we verify that the type of the instruction to be executed matches the one expected by the continuation, and assign the type of the final answer to the state.

$$\frac{\cdot; \cdot \triangleright i : \tau \qquad \triangleright K : \tau \Rightarrow \sigma}{\triangleright (K \diamond i) : \sigma}\ \textsf{vs\_}\diamond$$

$$\frac{\cdot \triangleright v : \sigma \qquad \cdot \triangleright v\ \mathit{Value}}{\triangleright\ \mathbf{answer}\ v : \sigma}\ \textsf{vs\_answer}$$

The typing judgments for instructions, continuations, and states admit more states as valid than can be reached from an initial state of the form **init** $\diamond$ **ev** e (see Exercise 6.19). However, they are accurate enough to permit proof of preservation and progress and is therefore appropriate for our purposes.

The implementation of the judgments in the logical framework is straightforward. The fact that we need at most one value variable is not explicitly represented. Instead, we use the usual techniques for parametric and hypothetical judgments by assuming `value x` for a new parameter `x`. The declarations below can be executed in Elf in order to check the validity of instructions, continuations and states and infer their most general types.

```
%%% Instructions
valid : inst -> tp -> type.          %name valid VL.
%mode valid +I *T.

% Evaluation and return
vi_ev : valid (ev E) T
         <- of E T.
vi_return : valid (return V) T
             <- of V T
             <- value V.

% Natural Numbers
vi_case1 : valid (case1 V1 E2 E3) T
            <- of V1 nat
            <- of E2 T
            <- ({x:exp} of x nat -> of (E3 x) T)
            <- value V1.

% Pairs
vi_pair1 : valid (pair1 V1 E2) (cross T1 T2)
            <- of V1 T1
            <- of E2 T2
            <- value V1.
vi_fst1 : valid (fst1 V') T1
           <- of V' (cross T1 T2)
           <- value V'.
vi_snd1 : valid (snd1 V') T2
           <- of V' (cross T1 T2)
           <- value V'.

% Functions
vi_app1 : valid (app1 V1 E2) T1
           <- of V1 (arrow T2 T1)
           <- of E2 T2
           <- value V1.
vi_app2 : valid (app2 V1 V2) T1
           <- of V1 (arrow T2 T1)
           <- of V2 T2
           <- value V1
           <- value V2.

%%% Continuations
validk : cont -> tp -> tp -> type.    %name validk VK.
%mode validk +K *T *S.

vk_init : validk (init) T T.
vk_; : validk (K ; [x] I x) T S
        <- ({x:exp} value x -> of x T -> valid (I x) T')
        <- validk K T' S.

%%% States
valids : state -> tp -> type.         %name valids VS.

vs_# : valids (K # I) S
        <- valid I T
        <- validk K T S.
vs_answer : valids (answer V) S
             <- of V S
             <- value V.
```

With this preparation, we can now prove preservation and progress. The proofs are quite straightforward, but somewhat tedious.

Theorem 6.26 (One-Step Type Preservation) *If* $\triangleright S : \sigma$ *and* $S \Longrightarrow S'$ *then* $\triangleright S' : \sigma$.

Proof: By cases on the derivation of $\mathcal{C} :: S \Longrightarrow S'$, applying several levels of inversion to the given typing derivation for S. The implementation is via a type family

```
vps : valids S T -> S => S' -> valids S' T -> type.
%mode vps +VS +C -VS'.
```

that relates the three derivation involved in the theorem. We show only a few representative cases; the full implementation can be found in the on-line course material.

Case: $\mathcal{C}$ is st_app:

$$K \diamond \mathbf{ev}\ (e_1\ e_2) \Longrightarrow K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{ev}\ e_1.$$

$\triangleright K \diamond \mathbf{ev}\ (e_1\ e_2) : \sigma$	Assumption
$\triangleright K : \tau \Rightarrow \sigma$ and	
$\cdot; \cdot \triangleright \mathbf{ev}\ (e_1\ e_2) : \tau$ for some τ	By inversion
$\cdot \triangleright e_1\ e_2 : \tau$	By inversion
$\cdot \triangleright e_1 : \tau_2 \rightarrow \tau$ and	
$\cdot \triangleright e_2 : \tau_2$ for some τ_2	By inversion
$x_1\ \mathit{Value}; x_1 : \tau_2 \rightarrow \tau \triangleright \mathbf{app}_1\ x_1\ e_2 : \tau$	By rule (vi_app$_1$)
$\triangleright (K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2) : (\tau_2 \rightarrow \tau) \Rightarrow \sigma$	By rule (vk_;)
$\triangleright (K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2) \diamond \mathbf{ev}\ e_1 : \sigma$	By rule (vs_ $\diamond$)

The inversion steps are represented by expanding the structure of the first argument, obtaining access to the subderivations VK of $\triangleright K : \tau \rightarrow \sigma$, P2 of $\cdot \triangleright e_2 : \tau_2$, and P1 of $\cdot \triangleright e_1 : \tau_2 \rightarrow \tau$. The needed derivation for S' is readily constructed from these variables.

```
vps_app : vps (vs_# VK (vi_ev (tp_app P2 P1))) (st_app)
            (vs_# (vk_; VK ([x1] [q1:value x1]
                              [p1:of x1 (arrow T2 T1)]
                                vi_app1 q1 P2 p1))
              (vi_ev P1)).
```

Case: $\mathcal{C}$ is st_app$_1$:

$$K \diamond \mathbf{app}_1\ v_1\ e_2 \Longrightarrow K; \lambda x_2.\ \mathbf{app}_2\ v_1\ x_2 \diamond \mathbf{ev}\ e_2.$$

$\triangleright K \diamond \mathbf{app}_1\ v_1\ e_2 : \sigma$	Assumption
$\triangleright K : \tau \Rightarrow \sigma$ and	
$\cdot ; \cdot \triangleright \mathbf{app}_1\ v_1\ e_2 : \tau$ for some τ	By inversion
$\cdot \triangleright v_1 : \tau_2 \rightarrow \tau$ and	
$\cdot \triangleright e_2 : \tau_2$ form some τ_2 and	
$\cdot \triangleright v_1\ \mathit{Value}$	By inversion
$x_2\ \mathit{Value}; x_2 : \tau_2 \triangleright\!\!\triangleright \mathbf{app}_2\ v_1\ x_2 : \tau$	By rule (vi_app$_2$)
$\triangleright (K; \lambda x_2.\ \mathbf{app}_1\ v_1\ e_2) : \tau_2 \Rightarrow \sigma$	By rule (vk_;)
$\triangleright (K; \lambda x_1.\ \mathbf{app}_2\ v_1\ x_2) \diamond \mathbf{ev}\ e_2 : \sigma$	By rule vs_cpm

In the representation we decompose the first argument as before. This now gives as also `Q1` which is the derivation of v_1 *Value*.

```
vps_app1 : vps (vs_# VK (vi_app1 Q1 P2 P1)) (st_app1)
            (vs_# (vk_; VK ([x2] [q2:value x2] [p2:of x2 T2]
                                  vi_app2 q2 Q1 p2 P1))
               (vi_ev P2)).
```

Case: $\mathcal{C}$ is st_app$_2$:

$$K \diamond \mathbf{app}_2\ (\mathbf{lam}\ x.\ e_1')\ v_2 \Longrightarrow K \diamond \mathbf{ev}\ ([v_2/x]e_1').$$

$\triangleright (K \diamond \mathbf{app}_2\ (\mathbf{lam}\ x.\ e_1')\ v_2) : \sigma$	Assumption
$\triangleright K : \tau_1 \Rightarrow \sigma$ and	
$\cdot ; \cdot \triangleright \mathbf{app}_2\ (\mathbf{lam}\ x.\ e_1')\ v_2) : \tau_1$ for some τ_1	By inversion
$\cdot \triangleright (\mathbf{lam}\ x.\ e_1') : \tau_2 \rightarrow \tau_1$ and	
$\cdot \triangleright v_2 : \tau_2$ for some τ_2 and	
$\cdot \triangleright (\mathbf{lam}\ x.\ e_1')\ \mathit{Value}$ and	
$\cdot \triangleright v_2\ \mathit{Value}$	By inversion
$x{:}\tau_2 \triangleright e_1' : \tau_1$	By inversion
$\cdot \triangleright [v_2/x]e_1' : \tau_1$	By substitution property (2.4)
$\cdot ; \cdot \triangleright \mathbf{ev}\ ([v_2/x]e_1') : \tau_1$	By rule (vi_ev)
$\triangleright K \diamond \mathbf{ev}\ ([v_2/x]e_1') : \sigma$	By rule (vs_cpm)

By applying inversion as above we obtained

$$P_1' :: (x{:}\tau_2 \triangleright e_1' : \tau_1)$$

which is represented by the variable

```
P1' : {x:exp} of x T2 -> of (E1' x) T1.
```

The appeal to the substitution principle is implemented by applying this function to the representation of v_2 and the typing derivation $\mathcal{P}_2 :: (\cdot \triangleright v_2 : \tau_2)$. Note that we do not need the derivation $\mathcal{Q}_2$ which is evidence that v_2 is a value.

```
vps_app2 : vps (vs_# VK (vi_app2 Q2 (val_lam) P2 (tp_lam P1')))
            (st_app2) (vs_# VK (vi_ev (P1' V2 P2))).
```

□

The multi-step type preservation theorem is a direct consequence of the one-step preservation.

Theorem 6.27 (Multi-Step Type Preservation) *If* $\triangleright S : \sigma$ *and* $S \overset{*}{\Longrightarrow} S'$ *then* $\triangleright S' : \sigma$

Proof: By straightforward induction on the structure of the derivation $\mathcal{C}^* :: (S \overset{*}{\Longrightarrow} S')$. We only show the implementation.

```
vps* : valids S T -> S =>* S' -> valids S' T -> type.
%mode vps* +VS +C* -VS'.

vps*_stop : vps* VS (stop) VS.
vps*_<< : vps* VS (C2* << C1) VS2
           <- vps VS C1 VS1
           <- vps* VS1 C2* VS2.
```

□

Finally, we come to the progress theorem: we can make a transition from every state that is not a final state. Recall that the only final states are of the form **answer** v.

Theorem 6.28 (Progress) *If* $\triangleright (K \diamond i) : \sigma$ *then there is a state* S' *such that* $K \diamond i \Longrightarrow S'$.

Proof: We know by inversion that

$\triangleright K : \tau \Rightarrow \sigma$ and
$\triangleright i : \tau$ for some τ.

We apply case analysis on i. In each case we can either directly make a transition, or we need to apply several inversions on an available typing or value derivation until each subcase can be seen to be impossible or a transition rule applies. For a **return** instruction, we also need to distinguish cases on the shape of K. We show only a few cases.

Case: $i = \mathbf{ev}\ (e_1\, e_2)$. Then $\mathsf{st\_app}$ applies.

Case: $i = \mathbf{app}_1\ v_1\, e_2$. Then $\mathsf{st\_app}_1$ applies.

Case: $i = \mathbf{app}_2\ v_1\ v_2$. Then

$\cdot \rhd v_1$ *Value* and
$\cdot \rhd v_2$ *Value* and
$\cdot \rhd v_1 : \tau_2 \to \tau$ and
$\cdot \rhd v_2\ : \tau_2$ By inversion

Now we distinguish subcases on $\cdot \rhd v_2$ *Value*

Subcase: $v_1 = \mathbf{z}$. This is impossible, since there is no rule to conclude $\cdot \rhd \mathbf{z} : \tau_2 \to \tau$.

Subcase: $v_1 = \mathbf{s}\ v_1'$. This is impossible, since there is no rule to conclude $\cdot \rhd \mathbf{s}\ v_1' : \tau_2 \to \tau$.

Subcase: $v_1 = \langle v_1', v_1'' \rangle$. This is impossible, since there is not rule to conclude $\cdot \rhd \langle v_1', v_1'' \rangle : \tau_2 \to \tau$

Subcase: $v_1 = \mathbf{lam}x.\ e_1'$. Then $\mathsf{st\_app}_2$ applies.

The implementation is very similar to progress, except that we don't need to construct the resulting typing derivation. Note that impossible cases are not represented. Also, for simplicity of implementation, we distinguish the cases on the typing derivation rather than the instruction, which is possible since the typing judgment is syntax-directed. We show only the three cases from above.

```
pgs : valids S T -> S => S' -> type.
%mode pgs +VS -C.

pgs_app : pgs (vs_# VK (vi_ev (tp_app P2 P1))) (st_app).

pgs_app1 : pgs (vs_# VK (vi_app1 Q1 P2 P1)) (st_app1).

pgs_app2 : pgs (vs_# VK (vi_app2 Q2 (val_lam) P2 (tp_lam P1')))
               (st_app2).
```

Note that some applications of inversion may be redundant for the sake of uniformity. For example, we could have replaced the last clause by

```
pgs_app2' : pgs (vs_# VK (vi_app2 Q2 (val_lam) P2 P1))
                (st_app2).
```

Nonetheless, the inversion on the value derivation is necessary, and

```
pgs_app2'' : pgs (vs_# VK (vi_app2 Q2 Q1 P2 P1))
                 (st_app2).
```

would be incorrect as a proof case because it is not apparent from the first argument that $\mathsf{st\_app}_2$ indeed applies. □

6.7 Contextual Semantics

[*This section discusses a contextual semantics as an alternative small-step machine to the CPM machine. This still has to be revised from an older version.*]

6.8 Exercises

Exercise 6.1 If we replace the rule ev_app in the natural semantics of Mini-ML (see Section 2.3) by

$$\dfrac{e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1' \qquad e_2 \hookrightarrow v_2 \qquad \begin{array}{c}\overline{x \hookrightarrow v_2}\ u \\ \vdots \\ e_1' \hookrightarrow v\end{array}}{e_1\ e_2 \hookrightarrow v}\ \mathsf{ev\_app}'^{x,u}$$

in order to avoid explicitly substituting v_2 for x, something goes wrong. What is it? Can you suggest a way to fix the problem which still employs hypothetical judgments?

(Note: We assume that the third premiss of the modified rule is parametric in x and hypothetical in u which is discharged as indicated. This implies that we assume that x is not already free in any other hypothesis and that all labels for hypotheses are distinct—so this is *not* the problem you are asked to detect.)

Exercise 6.2 Define the judgment W *RealVal* which restricts closures W to Λ-abstractions. Prove that $\cdot \vdash F \hookrightarrow W$ then W *RealVal* and represent this proof in Elf.

Exercise 6.3 In this exercise we try to eliminate some of the non-determinism in compilation.

1. Define a judgment F *std* which should be derivable if the de Bruijn expression F is in the standard form in which the $\uparrow$ operator is not applied to applications or abstractions.

2. Rewrite the translation from ordinary expressions e such that only standard forms can be related to any expression e.

3. Prove the property in item 2.

4. Implement the judgments in items 1, 2, and the proof in item 3.

Exercise 6.4 Restrict yourself to the fragment of the language with variables, abstraction, and application, that is,

$$F \quad ::= \quad 1 \mid F\uparrow \mid \Lambda F \mid F_1\ F_2$$

1. Define a judgment F *Closed* that is derivable iff the de Bruijn expression F is closed, that is, has no free variables at the *object level*.

2. Define a judgment for conversion of de Bruijn expressions F to standard form (as in Exercise 6.3, item 1) in a way that preserves meaning (as given by its interpretation as an ordinary expression e).

3. Prove that, under appropriate assumptions, this conversion results in a de Bruijn expression in standard form equivalent to the original expression.

4. Implement the judgments and correctness proofs in Elf.

Exercise 6.5 Restrict yourself to the same fragment as in Exercise 6.4 and define the operation of substitution as a judgment *subst* $F_1\ F_2\ F$. It should be a consequence of your definition that if ΛF_1 represents **lam** $x.\ e_1$, F_2 represents e_2, and *subst* $F_1\ F_2\ F$ is derivable then F should represent $[e_2/x]e_1$. Furthermore, such an F should always exist if F_1 and F_2 are as indicated. With appropriate assumptions about free variables or indices (see Exercise 6.4) prove these properties, thereby establishing the correctness of your implementation of substitution.

Exercise 6.6 Write out the informal proof of Theorem 6.7.

Exercise 6.7 Prove Theorem 6.8 by appropriately generalizing Lemma 6.2.

Exercise 6.8 Standard ML [MTH90] and many other formulations do not contain a **let name** construct. Disregarding problems of polymorphic typing for the moment, it is quite simple to simulate **let name** with **let val** operationally using so-called *thunks*. The idea is that we can prohibit the evaluation of an arbitrary expression by wrapping it in a vacuous **lam**-abstraction. Evaluation can be forced by applying the function to some irrelevant value (we write **z**, most presentations use a unit element). That is, instead of

$$l \quad = \quad \textbf{let name}\ x = e_1\ \textbf{in}\ e_2$$

we write

$$l' \quad = \quad \textbf{let val}\ x' = \textbf{lam}\ y.\ e_1\ \textbf{in}\ [x'\ \mathbf{z}/x]e_2$$

where y is a new variable not free in e_1.

1. Show a counterexample to the conjecture "*If l is closed, $l \hookrightarrow v$, and $l' \hookrightarrow v'$ then $v = v'$ (modulo renaming of bound variables)*".

2. Show a counterexample to the conjecture "$\triangleright\ l : \tau$ *iff* $\triangleright\ l' : \tau$".

3. Define an appropriate congruence $e \cong e'$ such that $l \cong l'$ and if $e \cong e'$, $e \hookrightarrow v$ and $e' \hookrightarrow v'$ then $v \cong v'$.

4. Prove the properties in item 3.

5. Prove that if the values v and v' are natural numbers, then $v \cong v'$ iff $v = v'$.

We need a property such as the last one to make sure that the congruence we define does not identify all expressions. It is a special case of a so-called *observational equivalence* (see **??**).

Exercise 6.9 The rules for evaluation in Section 6.2 have the drawback that looking up a variable in an environment and evaluation are mutually recursive, since the environment contains unevaluated expressions. Such expressions may be added to the environment during evaluation of a **let name** or **fix** construct. In the definition of Standard ML [MTH90] this problem is avoided by disallowing **let name** (see Exercise 6.8) and by syntactically restricting occurrences of the **fix** construct. When translated into our setting, this restriction states that all occurrences of fixpoint expressions must be of the form **fix** x. **lam** $y.\ e$. Then we can dispense with the environment constructor $+$ and instead introduce a constructor $*$ that builds a recursive environment. More precisely, we have

$$\text{Environments} \quad \eta \quad ::= \quad \cdot \mid \eta, W \mid \eta * F$$

The evaluation rules fev_1+, fev_↑+, and fev_fix on page 161 are replaced by

$$\frac{}{K \vdash \mathbf{fix}'\ F \hookrightarrow \{K * F; F\}}\ \textsf{fev\_fix}*$$

$$\frac{}{K * F \vdash 1 \hookrightarrow \{K * F; F\}}\ \textsf{fev\_1}*$$

$$\frac{K \vdash F \hookrightarrow W}{K * F' \vdash F\uparrow \hookrightarrow W}\ \textsf{fev\_}\!\uparrow\!*$$

1. Implement this modified evaluation judgment in Elf.

2. Prove that under the restriction that all occurrences of $\mathbf{fix}'$ in de Bruijn expressions have the form $\mathbf{fix}'\ \Lambda F$ for some F, the two sets of rules define an equivalent operational semantics. Take care to give a precise definition of the notion of equivalence you are considering and explain why it is appropriate.

3. Represent the equivalence proof in Elf.

4. Exhibit a counterexample which shows that some restriction on fixpoint expressions (as, for example, the one given above) is necessary in order to preserve equivalence.

5. Under the syntactic restriction from above we can also formulate a semantics which requires no new constructor for environments by forming closures over fixpoint expressions. Then we need to add another rule for application of an expression which evaluates to a closure over a fixpoint expression. Write out the rules and prove its equivalence to either the system above or the original evaluation judgment for de Bruijn expressions (under the appropriate restriction).

Exercise 6.10 Show how the effect of the *bind* instruction can be simulated in the CLS machine using the other instructions. Sketch the correctness proof for this simulation.

Exercise 6.11 Complete the presentation of the CLS machine by adding recursion. In particular

1. Complete the computation rules on page 173.

2. Add appropriate cases to the proofs of Lemmas 6.16, and 6.18.

Exercise 6.12 Prove the following carefully.

1. The concatenation operation "$\circ$" on computations is associative.

2. The subcomputation relation "$<$" is transitive (Lemma 6.17).

Show the implementation of your proofs as type families in Elf.

Exercise 6.13 The machine instructions from Section 6.3 can simply quote expressions in de Bruijn form and consider them as instructions. As a next step in the (abstract) compilation process, we can convert the expressions to lower-level code which simulates the effect of instructions on the environment and value stacks in smaller steps.

1. Design an appropriate language of operations.

2. Specify and implement a compiler from expressions to code.

3. Prove the correctness of this step of compilation.

4. Implement your correctness proof in Elf.

Exercise 6.14 Types play an important role in compilation, which is not reflected in the some of the development of this chapter. Ideally, we would like to take advantage of type information as much as possible in order to produce more compact and more efficient code. This is most easily achieved if the type information is embedded directly in expressions (see Section **??**), but at the very least, we would expect that types can be assigned to intermediate expressions in the compiler.

1. Define typing judgments for de Bruijn expressions, environments, and values for the language of Section 6.2. You may assume that values are always closed.

2. Prove type preservation for your typing judgment and the operational semantics for de Bruijn expressions.

3. Prove type preservation under compilation, that is, well-typed Mini-ML expressions are mapped to well-typed de Bruijn expressions under the translation of Section 6.2.

4. What is the converse of type preservation under compilation. Does your typing judgment satisfy it?

5. Implement the judgments above in Elf.

6. Implement the proofs above in Elf.

Exercise 6.15 As in Exercise 6.14:

1. Define a typing judgment for evaluation contexts. It should only hold for valid evaluation contexts.

2. Prove that splitting a well-typed expression which is not a value always succeeds and produces a unique context and redex.

3. Prove that splitting a well-typed expression results in a valid evaluation context and valid redex.

4. Prove the correctness of contextual evaluation with respect to the natural semantics for Mini-ML.

5. Implement the judgments above in Elf. Evaluation contexts should be represented as functions from expressions to expressions satisfying an additional judgment.

6. Implement the proofs above in Elf.

Exercise 6.16 Show that the purely expression-based natural semantics of Section 2.3 is equivalent to the one based on a separation between expressions and values in Section 6.5. Implement your proof, including all necessary lemmas, in Elf.

Exercise 6.17 Carry out the alternative proof of completeness of the continuation machine sketched on page 189. Implement the proof and all necessary lemmas in Elf.

Exercise 6.18 Do the equivalence proof in Lemma 6.22 and the alternative in Exercise 6.17 define the same relation between derivations? If so, exhibit the bijection in the form of a higher-level judgment relating the Elf implementations. Be careful to write out necessary lemmas regarding concatenation. You may restrict yourself to functional abstraction, application, and the necessary computation rules.

Exercise 6.19 Not every valid state of the CPM machine (according to the typing judgments in Section 6.6) can be reached by a computation starting from some initial state of the form $\mathbf{init} \diamond \mathbf{ev}\ e$ where $\cdot \vdash e : \tau$.

1. Exhibit a valid, but unreachable state.
2. Modify the validity judgments so that every valid machine state can in fact be reached from some initial state.
3. Prove this property.
4. Implement your proof in Elf.

Bibliography

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

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

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

[CCM87] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8, May 1987.

[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.

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[HM90] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332, Nice, France, 1990.

[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.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

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

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

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

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

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

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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