Computation and Deduction

Frank Pfenning
Carnegie Mellon University

$\boxed{\text{Draft of January 18, 2001}}$

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 a 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 [Pfe91, 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

$$\begin{array}{ll} \triangleright\, e : \tau & e \text{ has type } \tau \\ e \hookrightarrow v & e \text{ evaluates to } v \end{array}$$

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 **??** 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 **??**). 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 **??**) 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 **??**), 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*.

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

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,

$$\mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$$

denotes the same type as

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

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: $\mathbf{z}$ stands for zero, $\mathbf{s}\; e$ 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, $\mathbf{lam}\; x.\; e$, 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 $\mathbf{fix}\; x.\; e$ 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** $x.\ x$ **z** stands for **lam** $x.\ (x\ \mathbf{z})$ and

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

denotes the same expression as

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

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

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

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.

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

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

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

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.

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

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 **??**).

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* ($\mathbf{s}$ ($\mathbf{s}$ $\mathbf{z}$)) with the earlier definition of *pred*.

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

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.

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

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

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:

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

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*.

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

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

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 **??**.

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**).

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

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

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

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

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.

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

The representation of the evaluation tree for $double\ (\mathbf{s}\ \mathbf{z})$ 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	$double' \hookrightarrow double'$	ev_lam
2	$double \hookrightarrow double'$	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	$double' \hookrightarrow double'$	ev_lam
8	$double \hookrightarrow double'$	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	$(\mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (double\ x'))) \hookrightarrow \mathbf{z}$	ev_case_z $10, 11$
13	$double\ \mathbf{z} \hookrightarrow \mathbf{z}$	ev_app $8, 9, 12$
14	$\mathbf{s}\ (double\ \mathbf{z}) \hookrightarrow \mathbf{s}\ \mathbf{z}$	ev_s 13
15	$\mathbf{s}\ (\mathbf{s}\ (double\ \mathbf{z})) \hookrightarrow \mathbf{s}\ (\mathbf{s}\ \mathbf{z})$	ev_s 14
16	$(\mathbf{case}\ \mathbf{s}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (double\ x'))) \hookrightarrow \mathbf{s}\ (\mathbf{s}\ \mathbf{z})$	ev_case_s $6, 15$
17	$double\ (\mathbf{s}\ \mathbf{z}) \hookrightarrow \mathbf{s}\ (\mathbf{s}\ \mathbf{z})$	ev_app $2, 4, 16$

where

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

The inefficiencies of the rules we alluded to above can be seen clearly in this example: we need two copies of the evaluation of $\mathbf{s}\ \mathbf{z}$, 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 $\mathbf{z}$, 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: $e\ \mathit{Value}$. It is defined by the following inference rules:

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

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

Again, this definition is inductive: an expression e is a value if and only if $e\ \mathit{Value}$ 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 $\mathsf{val\_z}$.

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 $\mathsf{val\_s}$ 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 $\mathsf{ev\_pair}$ 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,

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

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

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

Case:

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

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:

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

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.* 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].

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

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.

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

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

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.

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

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

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 $\triangleright\ \mathbf{lam}\; x.\; \mathbf{lam}\; x.\; x : \alpha \rightarrow (\beta \rightarrow \beta)$, but not for $\triangleright\ \mathbf{lam}\; x.\; \mathbf{lam}\; x.\; x : \alpha \rightarrow (\beta \rightarrow \alpha)$. 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).

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

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:

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

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

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

are derivable. In a type system with explicit polymorphism a more general judgment might be expressed as $\triangleright\ \mathbf{lam}\ x.\ x : \forall\alpha.\ \alpha \to \alpha$ (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 $\mathbf{let\,name}\ x = e_1\ \mathbf{in}\ e_2$. We achieve this by substituting e_1 for x in e_2 and checking only that the result is well-typed.

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

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 $[\mathbf{fix}\ x.\ e/x]e$ in order to evaluate $\mathbf{fix}\ x.\ e$. 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

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

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* $\Gamma \triangleright e : \tau$ *then* $\Gamma, \Gamma' \triangleright e : \tau$ *provided* Γ, Γ' *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 **??**.

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

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

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

Then we construct a deduction

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

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

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

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

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
$\triangleright\, e_1 : \mathsf{nat}$ and $\tau = \mathsf{nat}$	By inversion
$\triangleright\, v_1 : \mathsf{nat}$	By ind. hyp. on $\mathcal{D}_1$
$\triangleright\, \mathbf{s}\ v_1 : \mathsf{nat}$	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: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ e_1 \hookrightarrow \mathbf{s}\, v_1'\end{array} \qquad \begin{array}{c}\mathcal{D}_3 \\ [v_1'/x]e_3 \hookrightarrow v\end{array}}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\, x \Rightarrow e_3) \hookrightarrow v}\,\textsf{ev\_case\_s}.$

$\triangleright (\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\, x \Rightarrow e_3) : \tau$	By assumption
$x{:}\textsf{nat} \triangleright e_3 : \tau$	By inversion
$\triangleright e_1 : \textsf{nat}$	By inversion
$\triangleright \mathbf{s}\, v_1' : \textsf{nat}$	By ind. hyp. on $\mathcal{D}_1$
$\triangleright v_1' : \textsf{nat}$	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: $\mathcal{D} = \dfrac{}{\mathbf{lam}\ x.\ e \hookrightarrow \mathbf{lam}\ x.\ e}\,\textsf{ev\_lam}.$

This case is immediate as for ev_z.

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

$\triangleright e_1\ e_2 : \tau$	By assumption
$\triangleright e_1 : \tau_2 \to \tau$ and $\triangleright e_2 : \tau_2$ for some τ_2	By inversion
$\triangleright \mathbf{lam}\ x.\ e_1' : \tau_2 \to \tau$	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: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ e_1 \hookrightarrow v_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ [v_1/x]e_2 \hookrightarrow v\end{array}}{\mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 \hookrightarrow v}\,\textsf{ev\_letv}.$

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

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

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

$\triangleright\ \textbf{fix}\ x.\ e_1 : \tau$	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,

$$pred_0 = \mathbf{lam}\, x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{fix}\, y.\, y \mid \mathbf{s}\ x' \Rightarrow x'$$

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 $\triangleright\, v : \mathsf{nat}$. 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 $\triangleright\, e : \mathsf{nat} \rightarrow \mathsf{nat}$, 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 $\mathsf{nat} \rightarrow \mathsf{nat}$ 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 $\triangleright\ id : \mathsf{nat} \rightarrow \mathsf{nat}$ and by the reasoning above we can interpret it as a function from natural numbers to natural numbers. We also have $\triangleright\ id : (\mathsf{nat} \rightarrow \mathsf{nat}) \rightarrow (\mathsf{nat} \rightarrow \mathsf{nat})$, 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 $\mathit{plus}_2\ (\mathbf{s}\ \mathbf{z})\ (\mathbf{s}\ \mathbf{z})$, 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 $\mathsf{ev\_fst}$ 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 $\tau_1 + \tau_2$:

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

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

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

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** ...) 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

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

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* : $\mathsf{nat} \to \mathsf{nat}$ 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 $\mathsf{ev\_pair}$, $\mathsf{ev\_fst}$, and $\mathsf{ev\_snd}$.

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 $\textbf{let val}\ x = e_1\ \textbf{in}\ e_2$ and $(\textbf{lam}\ x.\ e_2)\ e_1$ are equivalent in sense that

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

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.

Bibliography

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

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

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

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

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

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

[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.

[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.

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

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

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

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

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

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

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

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

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

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

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

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

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[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.

[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.