CSE-321 Programming Languages
Course Notes

Sungwoo Park

Spring 2009

Draft of September 30, 2011

Preface

This is a collection of course notes for CSE-321 *Programming Languages* at POSTECH. Any comments and suggestions will be greatly appreciated. I especially welcome feedback from students as to which part is difficult to follow and which part needs to be improved. The less background you have in functional languages and type theory, the more useful your comments will be. So please do not hesitate if you are taking this course!

For POSTECH PhD Qualification Exam, see the following sections:

- Chapter 2: Sections 2.1, 2.2, 2.3, 2.4

- Chapter 3: Sections 3.1, 3.2, 3.3, 3.4, 3.7

- Chapter 4: Sections 4.1, 4.2, 4.3, 4.4

- Chapter 5: Sections 5.1, 5.2, 5.3

- Chapter 6: Sections 6.1, 6.2, 6.3

- Chapter 8: Sections 8.1

- Chapter 9: Sections 9.1, 9.2

- Chapter 11: Sections 11.1, 11.2

- Chapter 13: Sections 13.1, 13.2, 13.4, 13.5, 13.6, 13.8

Contents

Chapter 1

Introduction to Functional Programming

This chapter presents basic ideas underlying functional programming, or programming in functional languages. All examples are written in Standard ML (abbreviated as SML henceforth), but it should be straightforward to translate them in any other functional language because all functional languages share the same design principle at their core. The results of running example programs are all produced in the interactive mode of SML of New Jersey.

Since this chapter is devoted to the discussion of important concepts in functional programming, the reader is referred to other sources for a thorough introduction to SML.

1.1 Functional programming paradigm

In the history of programming languages, there have emerged a few different programming paradigms. Each programming paradigm focuses on different aspects of programming, showing strength in some application areas but weakness in others. Object-oriented programming, for example, exploits the mechanism of extending object classes to express the relationship between different objects. Functional programming, as its name suggests, is unique in its emphasis on the role of functions as the basic component of programs. Combined with proper support for modular development, functional programming proves to be an excellent choice for developing large-scale software.

Functional programming is often compared with imperative programming to highlight its characteristic features. In imperative programming, the basic component is commands. Typically a program consists of a sequence of commands which yield the desired result when executed. Thus a program written in the imperative style is usually a description of "how to compute" — how to sort an array, how to add an element to a linked list, how to traverse a tree, and so on. In contrast, functional programming naturally encourages programmers to concentrate on "what to compute" because every program fragment must have a value associated with it.

To clarify the difference, let us consider an `if-then-else` conditional construct. In an imperative language (*e.g.*, C, Java, Pascal), the following code looks innocent with nothing suspicious; in fact, such code is often inevitable in imperative programming:

```
if (x == 1) then
   x = x + 1;
```

The above code executes the command to increment variable `x` when it is equal to `1`; if it is not equal to `1`, no command is executed — nothing wrong here. Now consider the following code written in a hypothetical functional language:

```
if (x = 1) then x + 1
```

With the `else` branch missing, the above code does not make sense: every program fragment must have a value associated with it, but the above code does not have a value when `x` is different from `1`. For this reason, it is mandatory to provide the `else` branch in every functional language.

1

As with other programming paradigms, the power of functional programming can be truly appreciated only with substantial experience with it. Unlike other programming paradigms, however, functional programming is built on a lot of fascinating theoretical ideas, independently of the issue of its advantages and disadvantages in software development. Thus functional programming should be the first step in your study of programming language theory!

1.2 Expressions and values

In SML, programs consists of *expressions* which range from simple constants to complex functions. Each expression can have a *value* associated with it, and the process of reducing an expression to a value is called *evaluation*. We say that an expression *evaluates* to a value when such a process terminates. Note that a value is a special kind of expression.

Example: integers

An integer constant `1` is an expression which is already a value. An integer expression `1 + 1` is not a value in itself, but evaluates to an integer value `2`. We can try to find the value associated with an expression by typing it and appending a semicolon at the SML prompt:

```
- 1 + 1;
val it = 2 : int
```

The second line above says that the result of evaluating the given expression is a `value 2`. (We ignore the type annotation `: int` until Section 1.5.)

All arithmetic operators in SML have names familiar from other programming languages (*e.g.*, +, −, *, div, mod). The only notable exception is the unary operator for negation, which is not – but ˜. For example, `˜1` is a negative integer, but `−1` does not evaluate to an integer.

Example: boolean values

Boolean constants in SML are `true` and `false`, and a conditional construct has the form `if` e `then` e_1 `else` e_2 where e, e_1, and e_2 are all expressions and e must evaluate to a boolean value. For example:

```
- if 1 = ˜1 then 10 else ˜10;
val it = ˜10 : int
```

Here `1 = ˜1` is an expression which compares two subexpressions `1` and `˜1` for equality. (In fact, = is also an expression — a binary function taking two arguments.) Since the two subexpressions are not equal, `1 = ˜1` evaluates to `false`, which in turn causes the whole expression to evaluate to `˜10`.

Logical operators available in SML include `andalso`, `orelse`, and `not`. The two binary operators implement short-circuiting internally, but short-circuiting makes no difference in pure functional programming because the evaluation of an expression never produces side effects.

Exercise 1.1. Can you simplify `if` e `then true else false`?

1.3 Variables

A *variable* is a container for a value. As an expression, it evaluates to the very value it stores. We use the keyword `val` to initialize a variable. For example, a variable `x` is initialized with an integer expression `1 + 1` as follows:

```
- val x = 1 + 1;
val x = 2 : int
```

Note that we must provide an expression to be used in computing the initial value for `x` because there is no default value for any variable in SML. (In fact, the use of default values for variables does not even conform to the philosophy of functional programming for the reason explained below.) After initializing `x`, we may use it in other expressions:

```
–  val y = x + x;
val y = 4 : int
```

We say that a variable is *bound* to a given value when it is initialized.

Unlike variables in imperative languages, a variable in SML is immutable in that its contents never change. In other words, a variable is bound to a single value for life. (This is the reason why it makes no sense to declare a variable without initializing it or by initializing it with a default value.) In this sense, "variable" is a misnomer because the contents of a variable is not really "variable." Despite their immutability, however, variables are useful in functional programming. Consider an example of a *local declaration* of SML in which zero or more *local variables* are declared before evaluating a final expression:

```
let
  val x = 1
  val y = x + x
in
  y + y
end
```

Here we declare two local variables `x` and `y` before evaluating `y + y`. Since `y` is added twice in the final expression, it saves computation time to declare `y` as a local variable instead of expanding both instances of `y` into `x + x`. The use of the local variable `y` also improves code readability.

While it may come as a surprise to you (especially if you still believe that executing a sequence of commands is the only way to complete a computation, as is typical in imperative programming), immutability of variables is in fact a feature that differentiates functional programming from imperative programming — without such a restriction on variables, there would be little difference between functional programming and imperative programming because commands (*e.g.*, for updating the contents of a variable) become available in both programming paradigms.

1.4 Functions

In the context of functional programming, a function can be thought of as equivalent to a mathematical function, *i.e.*, a black box mapping a given input to a unique output. Thus declaring a function in SML is indeed tantamount to defining a mathematical function, which in turn implies that the definition of a mathematical function is easily transcribed into an SML function. Interesting examples are given in Section 1.6 when we discuss recursion, and the present section focuses on the concept of function and its syntax in SML.

We use the keyword `fun` to declare a function. For example, we declare a function `incr` that returns a given integer incremented by one as follows:

```
–  fun incr x = x + 1;
val incr = fn : int -> int
```

Here `x` is called a *formal argument/parameter* because it serves only as a placeholder for an *actual argument/parameter*. `x + 1` is called a *function body*. We can also create a nameless function with the keyword `fn`. The following code creates the same function as `incr` and stores it in a variable `incr`:

```
–  val incr = fn x => x + 1;
val incr = fn : int -> int
```

The two declarations above are equivalent to each other.

A *function application* proceeds by substituting an actual argument for a formal argument in a function body and then evaluating the resultant expression. For example, a function application `incr 0` (applying function `incr` to `0`) evaluates to integer `1` via the following steps:

```
       incr 0
 ↦   (fn x => x + 1) 0
 ↦   0 + 1
 ↦   1
```

As an expression, a function is already a value. Intuitively a function is a black box whose internal working is hidden from the outside, and thus cannot be further reduced. As a result, a function body is evaluated only when a function application occurs. For example, a nameless function `fn x => 0 + 1` does *not* evaluate to `fn x => 1`; only when applied to an actual argument (which is ignored in this case) does it evaluate its body.

An important feature of functional programming is that functions are treated no differently from primitive values such as boolean values and integers. For example, a function can be stored in a variable (as shown above), passed as an actual argument to another function, and even returned as a return value of another function. Such values are often called *first-class objects* in programming language jargon because they are the most basic element comprising a program. Hence functions are first-class objects in functional languages. In fact, it turns out that a program in a functional language can be thought of as consisting entirely of functions and nothing else, since primitive values can also be encoded in terms of functions (which will be discussed in Chapter 3).

Interesting examples exploiting functions as first-class objects are found in Section 1.10. For now, we will content ourselves with an example illustrating that a function can be a return value of another function. Consider the following code which declares a function `add` taking two integers to calculate their sum:

```
- fun add x y = x + y;
val add = fn : int -> int -> int
```

Then what is a nameless function corresponding to `add`? A naive attempt does not even satisfy the syntax rules:

```
- val add = fn x y => x + y;
<some syntax error message>
```

The reason why the above attempt fails is that every function in SML can have only a single argument! The function `add` above (declared with the keyword `fun`) appears to have two arguments, but it is just a disguised form of a function that take an integer and returns another function:

```
- val add = fn x => (fn y => x + y);
val add = fn : int -> int -> int
```

That is, when applied to an argument `x`, it returns a new function `fn y => x + y` which returns `x + y` when applied to an argument `y`. Thus it is legitimate to apply `add` to a single integer to instantiate a new function as demonstrated below:

```
- val incr = add 1;
val incr = fn : int -> int
- incr 0;
val it = 1 : int
- incr 1;
val it = 2 : int
```

Now it should be clear how the evaluation of `add 1 1` proceeds:

```
     add 1 1
↦    (fn x => (fn y => x + y)) 1 1
↦    (fn y => 1 + y) 1
↦    1 + 1
↦    2
```

1.5 Types

Documentation is an integral part of good programming — without proper documentation, no code is easy to read unless it is self-explanatory. The importance of documentation (which is sometimes overemphasized in an introductory programming language course), however, often misleads students into thinking that long documentations are always better than concise ones. This is certainly untrue! For example, an overly long documentation on the function `add` can be more distracting than helpful to the reader:

```
(* Takes two arguments and returns their sum.
 * Both arguments must be integers.
 * If not, the result is unpredictable.
 * If their sum is too large, an overflow may occur.
 * ...
 *)
```

The problem here is that what is stated in the documentation cannot be formally verified by the compiler and we have to trust whoever wrote it. As an unintended consequence, any mistake in the documentation can leave the reader puzzled about the meaning of the code rather than helping her understand it. (For the simple case of `add`, it is not impossible to formally prove that the result is the sum of the two arguments, but then how can you express this property of `add` as part of the documentation?)

On the other hand, short documentations that can be formally verified by the compiler are often useless. For example, we could extend the syntax of SML so as to annotate each function with the number of its arguments:

```
argnum add 2          (* NOT valid SML syntax! *)
fun add x y = x + y;
```

Here `argnum add 2`, *as part of the code*, states that the function `add` has two arguments. The compiler can certainly verify that `add` has two arguments, but this property of `add` does not seem to be useful.

Types are a good compromise between expressiveness and simplicity: they convey *useful* information on the code (expressiveness) and can be *formally* verified by the compiler (simplicity). Informally a type is a collection of values of the same kind. For example, an expression that evaluates to an integer or a boolean constant has type `int` or `bool`, respectively. The function `add` has a *function type* `int -> (int -> int)` because given an integer of type `int`, it returns another function of type `int -> int` which takes an integer and returns another integer.[1] To exploit types as a means of documentation, we can explicitly annotate any formal argument with its type; the return type of a function and the type of any subexpression in its body can also be explicitly specified:

```
- fun add (x:int) (y:int) : int = (x + y) : int;
val add = fn : int -> int -> int
```

The SML compiler checks if types provided by programmers are valid; if not, it spits out an error message:

```
- fun add (x:int) (y:bool) = x + y;
stdIn:2.23-2.28 Error: <some error message>
```

Perhaps `add` is too simple an example to exemplify the power of types, but there are countless examples in which the type of a function explains what it does.

Another important use of types is as a debugging aid. In imperative programming, successful compilation seldom guarantees absence of errors. Usually we compile a program, run the executable code, and *then* start debugging by examining the result of the execution (be it a segmentation fault or a number different than expected). In functional programming with a rich type system, the story is different: we start debugging a program *before running the executable code* by examining the result of the compilation which is usually a bunch of *type errors*. Of course, neither in functional programming does successful compilation guarantee absence of errors, but programs that successfully compile run correctly *in most cases!* (You will encounter numerous such examples in doing assignments.) Types are such a powerful tool in software engineering.

1.6   Recursion

Many problems in computer science require iterative procedures to reach a solution – adding integers from 1 to 100, sorting an array, searching for an entry in a B-tree, and so on. Because of its prominent role in programming, iterative computation is supported by built-in constructs in all programming languages. The C language, for example, provides constructs for directly implementing iterative computation such as the `for` loop construct:

---

[1] `->` is right associative and thus `int -> int -> int` is equal to `int -> (int -> int)`.

```
    for (i = 1; i <= 10; i++)
        sum += i;
```

The example above uses an index variable `i` which changes its value from `1` to `10`. Surprisingly we cannot translate the above code in the pure fragment of SML (*i.e.*, without mutable references) because no variable in SML is allowed to change its value! Does this mean that SML is inherently inferior to C in its expressive power? The answer is "no:" SML supports *recursive computations* which are equally expressive to iterative computations.

Typically a recursive computation proceeds by decomposing a given problem into smaller problems, solving these smaller problems separately, and then combining their individual answers to produce a solution to the original problem. (Thus it is reminiscent of the divide-and-conquer algorithm which is in fact a particular instance of recursion.) It is important that these smaller problems are also solved recursively using the same method, perhaps by spawning another group of smaller problems to be solved recursively using the same method, and so on. Since such a sequence of decomposition cannot continue indefinitely (causing nontermination), a recursive computation starts to backtrack when it encounters a situation in which no such decomposition is necessary (*e.g.*, when the problem at hand is immediately solvable). Hence a typical form of recursion consists of *base cases* to specify the termination condition and *inductive cases* to specify how to decompose a problem into smaller ones.

As an example, here is a recursive function `sum` adding integers from 1 to a given argument `n`; note that we cannot use the keyword `fn` because we need to call the same function in its function body:

```
- fun sum n =
    if n = 1 then 1              (* base case *)
    else n + sum (n - 1);       (* inductive case *)
val sum = fn : int -> int
```

The evaluation of `sum 10` proceeds as follows:

```
      sum 10
↦     if 10 = 1 then 1 else 10 + sum (10 - 1)
↦     if false then 1 else 10 + sum (10 - 1)
↦     10 + sum (10 - 1)
↦     10 + sum 9
↦*    10 + 9 + ⋯ 2 + sum 1
↦     10 + ⋯ 2 + (if 1 = 1 then 1 else 1 + sum (1 - 1))
↦     10 + ⋯ 2 + (if true then 1 else 1 + sum (1 - 1))
↦     10 + ⋯ 2 + 1
```

As with iterative computations, recursive computations may fall into infinite loops (*i.e.*, non-terminating computations), which occur if the base condition is never reached. For example, if the function `sum` is invoked with a negative integer as its argument, it goes into an infinite loop (usually ending up with a stack overflow). In most cases, however, an infinite loop is due to a design flaw in the function body rather than an invocation with an inappropriate argument. Therefore it is a good practice to design a recursive function *before* writing code. A good way to design a recursive function is to formulate a mathematical equation. For example, a mathematical equation for `sum` would be given as follows:

$$
\begin{aligned}
sum(1) &= 1 \\
sum(n) &= 1 + sum(n-1) \qquad \text{if } n > 1
\end{aligned}
$$

Once such a mathematical equation is formulated, it should take little time to transcribe it into an SML function. (So think a lot before you write code!)

SML also supports mutually recursive functions. The keyword `and` is used to declare two or more mutually recursive functions. The following code declares two mutually recursive functions `even` and `odd` which determine whether a given natural number is even or odd:

```
fun even n =
  if n = 0 then true
  else odd (n - 1)
and odd n =
  if n = 0 then false
  else even (n - 1)
```

Recursion may appear at first to be an awkward device not suited to iterative computations. This may be because iterative approaches, which are in fact intuitively easier to comprehend than recursive approaches, come first to mind (after being indoctrinated with mindless imperative programming!). Once you get used to functional programming, however, you will find that recursion is not an awkward device at all, but the most elegant device you can use in programming. (Note that *elegant* is synonymous with *easy-to-use* in the context of programming.) So the bottom line is: always think recursively!

1.7 Polymorphic types

In a software development process, we often write the same pattern of code repeatedly only with minor differences. As such, it is desirable to write a single common piece of code and then instantiate it with a different parameter whenever a copy of it is needed, thereby achieving a certain degree of code reuse. The utility of such a scheme is obvious. For example, if a bug is discovered in the program, we do not have to visit all the different places only to make the same change.

The question of how to realize code reuse in a safe way is quite subtle, however. For example, the C language provides macros to facilitate code reuse, but macros are notoriously prone to unexpected errors. Templates in the C++ language are safer because their parameters are types, but they are still nothing more than complex macros. In contrast, SML provides a code reuse mechanism, which not only is safe but also has a solid theoretical foundation, called *parametric polymorphism*.[2]

As a simple example, consider an identity function `id`:

```
val id = fn x = x;
```

Since we do not specify the type of `x`, it may accept an argument of any type. Semantically such an invocation of `id` poses no problem because its body does not need to know what type of value `x` is bound to. This observation suggests that a single declaration of `id` is conceptually equivalent to an infinite number of declarations all of which share the same function body:

```
val id_int = fn (x:int) = x;
val id_bool = fn (x:bool) = x;
val id_{int→int} = fn (x:int -> int) = x;
...
```

When `id` is applied to an argument of type A, the SML compiler automatically chooses the right declaration of `id` for type A.

The type system of SML compactly represents all these declarations of the same structure with a single declaration by exploiting a *type variable*:

```
- val id = fn (x:'a) => x;
val id = fn : 'a -> 'a
```

Here type variable `'a` may read *"any type α"*.[3] Then the type of `id` means that given an argument of any type α, it returns a value of type α. We may also explicitly specify type variables that may appear in a variable declaration by listing them before the variable:

```
- val 'a id = fn (x:'a) => x;
val id = fn : 'a -> 'a
```

We refer to types with no type variables as *monotypes* (or *monomorphic types*), and types with some type variables as *polytypes* (or *polymorphic types*). The type system of SML allows a type variable to be replaced by any monotype (but not a polytype).

1.8 Datatypes

We have briefly discussed a few primitive types in SML. Here we give a comprehensive summary of basic types available in SML for future reference:

---

[2]A bit similar to, but not to be confused with the *polymorph* spell in the Warcraft series!

[3]Conventionally type variables are read as Greek letters (*e.g.*, `'a` as alpha, `'b` as beta, and so on).

- `bool`: boolean values `true` and `false`.

- `int`: integers.
  E.g., $0, 1, \tilde{\ }1, \cdots$.

- `real`: floating pointer numbers.
  E.g., $0.0, 1.0, \tilde{\ }1.0$.

- `char`: characters.
  E.g., `#"a", #"b", #" "`.

- `string`: character strings.
  E.g., `"hello", "newline\n", "quote\"", "backslash\\"`.

- A `->` B: functions from type A to type B.

- A `*` B: pairs of types A and B.
  If e_1 has type A and e_2 has type B, then `(`e_1`, `e_2`)` has type A `*` B.
  E.g., `(0, true) : int * bool`.
  A `*` B is called a *product type*.

- A_1 `*` A_2 `*` $\cdots$ `*` A_n: tuples of types A_1 through A_n.
  E.g., `(0, true, 1.0) : int * bool * real`.
  Tuple types are a generalized form of product types.

- `unit`: unit value `()`.
  The only value belonging to type `unit` is `()`. It is useful when declaring a function taking no interesting argument (*e.g.,* of type `unit -> int`).

These types suffice for most problems in which only numerical computations are involved. There are, however, a variety of problems for which *symbolic computations* are more suitable than numerical computations. As an example, consider the problem of classifying images into three categories: circles, squares, and triangles. We can assign integers `1, 2, 3` to these three shapes and use a function of type `image -> int` to classify images (where `image` is the type for images). A drawback of this approach is that it severely reduces the maintainability of the code: there is no direct connection between shapes and type `int`, and programmers should keep track of which variable of type `int` denotes shapes and which denotes integers.

A better approach is to represent each shape with a symbolic constant. In SML, we can use a *datatype* declaration to define a new type `shape` for three symbolic constants:

```
datatype shape = Circle | Square | Triangle
```

Each symbolic constant here has type `shape`. For example:

```
- Circle;
val it = Circle : shape
```

Note that datatypes are a special way of defining new types; hence they form only a subset of types. For example, `int -> int` is a (function) type but not a datatype whereas every datatype is also a type.

A datatype declaration is similar to an enumeration type in the C language, but with an important difference: symbolic constants are not compatible with integers and cannot be substituted for integers. Hence the new approach based on datatypes does not suffer from the same disadvantage of the previous approach. We refer to such symbolic constants as *data constructors*, or simply *constructors* in SML.

There is another feature of SML datatypes that sets themselves apart from C enumeration types: constructors may have arguments. For example, we can augment the above datatype declaration with arguments for constructors:

```
datatype shape =
   Circle of real
| Square of real
| Triangle of real * real * real
```

To create values of type `shape`, then, we have to provide appropriate arguments for constructors:

```
- Circle 1.0;
val it = Circle 1.0 : shape
- Square 1.0;
val it = Square 1.0 : shape
- Triangle (1.0, 1.0, 1.0);
val it = Triangle (1.0,1.0,1.0) : shape
```

Note that each constructor may be seen as a function from its argument type to type `shape`. For example, `Circle` is a function of type `real -> shape`:

```
- Circle;
val it = fn : real -> shape
```

Now we will discuss two important extensions of the datatype declaration mechanism. To motivate the first extension, consider a datatype `pair_bool` for pairing boolean values and another datatype `pair_int` for pairing integers:

```
datatype pair_bool = Pair of bool * bool
datatype pair_int = Pair of int * int
```

The two declarations are identical in structure except their argument types. We have previously seen how declarations of functions with the same structure but with different argument types can be coalesced into a single declaration by exploiting type variables. The situation here is no different: for any type A, a new datatype `pair_A` can be declared exactly in the same way:

```
datatype pair_A = Pair of A * A
```

The SML syntax for parameterizing a datatype declaration with type variables is to place type variables before the datatype name to indicate which type variables are local to the datatype declaration. Here are a couple of examples:

```
datatype 'a pair = Pair of 'a * 'a
datatype ('a, 'b) hetero = Hetero of 'a * 'b

- Pair (0, 1);
val it = Pair (0,1) : int pair
- Pair (0, true);
stdIn:5.1-5.15 Error: <some error message>
- Hetero (0, true);
val it = Hetero (0,true) : (int,bool) hetero
```

The second extension of the datatype declaration mechanism allows a datatype to be used as the type of arguments for its own constructors. As with recursive functions, there must be at least one constructor (corresponding to base cases for recursive functions) that does not use the same datatype for its arguments — without such a constructor, it would be impossible to build values belonging to the datatype. Such datatypes are commonly referred to as *recursive datatypes*, which enable us to implement recursive data structures. As an example, consider a datatype `itree` for binary trees whose nodes store integers:

```
datatype itree =
   Leaf of int
 | Node of itree * int * itree
```

The constructor `Leaf` represents a leaf node storing an integer of type `int`; the constructor `Node` represents an internal node which contains two subtrees as well as an integer. For example,

```
Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4)
```

represents the following binary tree:

```
              7
             / \
            3   4
           / \
          1   2
```
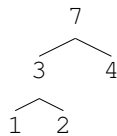
Note that without using `Leaf`, it is impossible to build a value of type `itree`.

The two extensions of the datatype declaration mechanism may coexist to define *recursive datatypes with type variables*. For example, the datatype `itree` above can be generalized to a datatype `tree` for binary trees of values of *any* type:

```
datatype 'a tree =
  Leaf of 'a
| Node of 'a tree * 'a * 'a tree
```

Now arguments to constructors `Leaf` and `Node` automatically determine the actual type to be substituted for type variable `'a`:

```
- Node (Leaf ~1, 0, Leaf 1);
val it = Node (Leaf ~1,0,Leaf 1) : int tree
- Node (Leaf "L", "C", Leaf "R");
val it = Node (Leaf "L","C",Leaf "R") : string tree
```

Note that once type variable `'a` is instantiated to a specific type (say `int`), values of different types (say `string`) cannot be used. In other words, `tree` can be used only for homogeneous binary trees. The following expression does not compile because it does not determine a unique type for `'a`:

```
- Node (Leaf ~1, "C", Leaf 1);
stdIn:35.1-35.28 Error: <some error message>
```

Recursive data structures are common in both functional and imperative programming. In conjunction with type parameters, recursive datatypes in SML (and any other functional language) enable programmers to implement most of the common recursive data structures in a concise and elegant way. More importantly, a recursive data structure implemented in SML can have the compiler recognize some of its invariants, *i.e.*, certain conditions that must hold for all instances of the data structure. For example, given the definition of the datatype `tree` above, the SML compiler is aware of the invariant that every internal node must have two child nodes. This invariant would not be trivial to enforce in imperative programming. (Can you implement in the C or C++ language a datatype for binary trees in which every internal node has exactly two child nodes?)

We close this section with a brief introduction to the most frequently used datatype in functional programming: lists. They are provided as a built-in datatype `list` with two constructors: a unary constructor `nil` and a binary constructor `::`:

```
datatype 'a list = nil | :: of 'a * 'a list
```

`nil` denotes an empty list of any type (because it has no argument). `::`, called *cons*, is a right-associative infix operator and builds a list of type `'a list` by concatenating its *head* of type `'a` and *tail* of type `'a list`. For example, the following expression denotes a list consisting of `1`, `2`, and `3` in that order:

```
1 :: 2 :: 3 :: nil
```

Another way to create a list in SML is to enumerate its elements separated by commas `,` within brackets (`[` and `]`). For example, `[1, 2, 3]` is an abbreviation of the list given above. The two notations may also appear simultaneously within any list expression. For example, the following expressions are all equivalent:

```
[1, 2, 3]
1 :: [2, 3]
1 :: 2 :: [3]
1 :: 2 :: 3 :: []
```

1.9   Pattern matching

So far we have investigated how to create expressions of various types in SML. Equally important is the question of how to inspect those values that such expressions evaluate to. For simple types such as integers and tuples, the question is easy to answer: we only need to invoke operators already available in SML. For example, we use arithmetic and comparison operators on integers to test if a given integer belongs to a certain interval; for tuple types (including product types), we use the projection operator #*n* to retrieve the *n*-th element of a given tuple (*e.g*, `#2(1, 2, 3)` evaluates to 2). In order to answer the question for datatypes, however, we need a means of testing which constructor has been applied in creating values of a given datatype. What makes this possible in SML is *pattern matching*.

As an example, let us write a function `length` that calculates the length of a given list. The way that `length` works is by simple recursion:

- If the argument is `nil`, return 0.

- If the argument has the form `<head> :: <tail>`, then invoke `length` on `<tail>` and return the result incremented by 1 (to account for `<head>`).

Thus `length` tries to match a given list with `nil` and `::` in either order. Moreover, when the list is matched with `::`, it also needs to retrieve arguments to `::` so as to invoke itself once more. The above definition of `length` is translated into the following code using pattern matching (which remotely resembles the `switch` construct in the C language):

```
fun length l =
  case l of
    nil => 0
  | head :: tail => 1 + length tail
```

Note that `nil` here is *not* a value; rather it is a *pattern* to be compared with `l` (or whatever follows the keyword `case`). Likewise `head :: tail` is a pattern which, when matched, binds `head` to the head of `l` and `tail` to the tail of `l`. If a pattern match occurs, the whole `case` expression reduces to the expression to the right of the corresponding `=>`. We call `nil` and `head :: tail` *constructor patterns* because of their use of datatype constructors.

Exercise 1.2.   What is the type of `length`?

In the case of `length`, we do not need `head` in the second pattern. A *wildcard pattern* _ may be used if no binding is necessary:

```
fun length l =
  case l of
    nil => 0
  | _ :: tail => 1 + length tail
```

_ alone is also a valid pattern, which comes in handy when not every constructor needs to be considered. For example, a function testing if a given list is nil can be implemented as follows:

```
fun testNil l =
  case l of
    nil => true       (* if l is nil *)
  | _ => false        (* for ALL other cases *)
```

Regardless of constructors associated with the datatype `list`, the case analysis above is exhaustive because _ matches with any value.

Pattern matching in SML can be thought of as a generalized version of the `if-then-else` conditional construct. In fact, pattern matching is applicable to *any* type (not just datatypes with constructors). For example, if *e* then *e*$_1$ else *e*$_2$ may be expanded into:

```
 case e of               case e of
    true => e₁     or       true => e₁
  | false => e₂           | _ => e₂
```

It turns out that all variables in SML are also a special form of patterns, which in turn implies that any variable may be replaced by another pattern. We have seen two ways of introducing variables in SML: using the keyword `val` and in function declarations. Thus what immediately follows `val` can be not just a single variable but also any form of pattern; similarly formal arguments in a function declaration can be any form of pattern. As a first example, an easy way to retrieve all individual elements of a tuple is to exploit a tuple pattern (instead of repeatedly using the projection operator `#n`):

```
val (x, y, z) = <some tuple expression>
```

You can even use a constructor pattern `head :: tail` to match with a list:

```
val (head :: tail) = [1, 2, 3];
```

Here `head` becomes bound to `1` and `tail` to `[2, 3]`. Note, however, that the pattern is not exhaustive. For example, if `nil` is given as the right hand side, there is no way to match `head :: tail` with `nil`. (We will see in Section 1.11 how to handle such abnormal cases.) As a second example, we can rewrite the mutually recursive functions `even` and `odd` using pattern matching:

```
fun even 0 = true
  | even n = odd (n - 1)
and odd 0 = false
  | odd n = even (n - 1)
```

1.10   Higher-order functions

We have seen in Section 1.4 that every function in SML is a first-class object — it can be passed as an argument to another function and also returned as the result of a function application. Then a function that takes another function as an argument or returns another function as the result has a function type $A \ \text{->} \ B$ in which A and B themselves may contain function types. We refer to such a function as a *higher-order* function. For example, functions of type `(int -> int) -> int` or `int -> (int -> int)` are all higher-order functions. We may also use type variables in higher-order function types. For example, `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)` is a higher-order function type. (Can you guess what a function of this type is supposed to do?)

Higher-order functions can significantly simplify many programming tasks when properly used. As an example, consider a higher-order function `map` of type `('a -> 'b) -> 'a list -> 'b list`.

Exercise 1.3.   Make an educated guess of what a function of the above type is supposed to do. If you make a correct guess, it typifies the use of types as a means of documentation!

As you might have guessed, `map` takes a function f of type `'a -> 'b` and a list l of type `'a list`, and applies f to each element of l to create another list of type `'b list`. Here is an example of using `map` with the two functions `even` and `odd` defined above:

```
- map even [1, 2, 3, 4];
val it = [false,true,false,true] : bool list
- map odd [1, 2, 3, 4];
val it = [true,false,true,false] : bool list
```

The behavior of `map` is formally written as follows:

$$\text{map } f \ [l_1, l_2, \cdots, l_n] \quad = \quad [f \ l_1, f \ l_2, \cdots, f \ l_n] \qquad (n \geq 0) \tag{1.1}$$

In order to implement `map`, we rewrite the equation (1.1) inductively by splitting it into two cases: a base case $n = 0$ and an inductive case $n > 0$. The base case is easy because the right side is an empty list:

$$\text{map } f \ [] \quad = \quad [] \tag{1.2}$$

The inductive case exploits the observation that $[f\ l_2, \cdots, f\ l_n]$ results from another application of map:[4]

$$\texttt{map}\ f\ [l_1, l_2, \cdots, l_n] \;=\; f\ l_1 \;\texttt{::}\; \texttt{map}\ f\ [l_2, \cdots, l_n] \quad (n > 0) \tag{1.3}$$

The two equations (1.2) and (1.3) derive the following definition of map:

```
fun map f [] = []
  | map f (head :: tail) = f head :: map f tail
```

map processes elements of a given list independently. Another higher-order function foldl (meaning "fold left") processes elements of a given list sequentially by using the result of processing an element when processing the next element. It has type (`'a * 'b -> 'b) -> 'b -> 'a list -> 'b` where `'b` denotes the type of the result of processing an element. Its behavior is formally written as follows:

$$\texttt{foldl}\ f\ a_0\ [l_1, l_2, \cdots, l_n] \;=\; f\ (l_n, \cdots f\ (l_2, f\ (l_1, a_0)) \cdots) \qquad (n \geq 0) \tag{1.4}$$

a_0 can be thought of as an initial value of an accumulator whose value changes as elements in the list are sequentially processed. Thus the equation (1.4) can be expanded to a series of equations as follows:

$$
\begin{aligned}
a_1 &= f\ (l_1,\ a_0) \\
a_2 &= f\ (l_2,\ a_1) \\
&\;\vdots \\
a_n &= f\ (l_n,\ a_{n-1}) = \texttt{foldl}\ f\ a_0\ [l_1, l_2, \cdots, l_n]
\end{aligned}
$$

As with map, we implement foldl by rewriting the equation (1.4) inductively. The base case returns the initial value a_0 of the accumulator:

$$\texttt{foldl}\ f\ a_0\ [] \;=\; a_0 \tag{1.5}$$

The inductive case makes a recursive call with a new value of the accumulator:

$$\texttt{foldl}\ f\ a_0\ [l_1, l_2, \cdots, l_n] \;=\; \texttt{foldl}\ f\ (f\ (l_1, a_0))\ [l_2, \cdots, l_n] \quad (n > 0) \tag{1.6}$$

The two equations (1.5) and (1.6) derive the following definition of foldl:

```
fun foldl f a [] = a
  | foldl f a (head :: tail) = foldl f (f (head, a)) tail
```

As an example, we can obtain the sum of integers in a list with a call to foldl:

```
- foldl (fn (x, y) => x + y) 0 [1, 2, 3, 4];
val it = 10 : int
```

Exercise 1.4. A similar higher-order function is foldr (meaning "fold right") which has the same type as foldl, but processes a given list from its last element to its first element:

$$\texttt{foldr}\ f\ a_0\ [l_1, l_2, \cdots, l_n] \;=\; f\ (l_1, \cdots f\ (l_{n-1}, f\ (l_n, a_0)) \cdots) \qquad (n \geq 0)$$

Give an implementation of foldr.

1.11   Exceptions

Exceptions in SML provide a convenient mechanism for handling erroneous conditions that may arise during a computation. An exception is generated, or *raised*, either by the runtime system when an erroneous condition is encountered, or explicitly by programmers to transfer control to a different part of the program. For example, the runtime system raises an exception when a division by zero occurs, or

---

[4] :: has a lower operator precedence than function applications, so we do not need parentheses.

when no pattern in a `case` expression matches with a given value; programmers may choose to raise an exception when an argument to a function does not satisfy the invariant of the function. An exception can be *caught* by an *exception handler*, which analyzes the exception to decide whether to raise another exception or to resume the computation. Thus not every exception results in aborting the computation.

An exception is a data constructor belonging to a special built-in datatype `exn` whose set of constructors can be freely extended by programmers. An exception declaration consists of a data constructor declaration preceded by the keyword `exception`. For example, we declare an exception `Error` with a string argument as follows:

```
exception Error of string
```

To raise `Error`, we use the keyword `raise`:

```
raise Error "Message for Error"
```

As exceptions are constructors for a special datatype `exn`, the syntax for exception handlers also uses pattern matching:

```
e handle
  <pattern₁> => e₁
| ...
| <patternₙ> => eₙ
```

If an exception is raised during the evaluation of expression e, $<pattern_1>$ through $<pattern_n>$ are tested in that order for a pattern match. If $<pattern_i>$ matches with the exception, e_i becomes a new expression to be evaluated; if no pattern matches, the exception is propagated to the next exception handler, if any.

As a contrived example, consider the following code:

```
exception BadBoy of int;
exception BadGirl of int;
1 + (raise BadGirl ~1) handle
  BadBoy s => (s * s)
| BadGirl s => (s + s)
```

Upon attempting to evaluate the second operand of `+`, an exception `BadGirl` with argument `~1` is raised. Then the whole evaluation is aborted and the exception is propagated to the enclosing exception handler. As the second pattern matches with the exception being propagated, the expression `s + s` to the right of `=>` becomes a new expression to be evaluated. With `s` replaced by the argument `~1` to `BadGirl`, the whole expression evaluates to `~2`.

Exceptions are useful in a variety of situations. Even fully developed programs often exploit the exception mechanism to deal with exceptional cases. For example, when a time consuming computation is interrupted with a division by zero, the exception mechanism comes to rescue to save the partial result accumulated by the time of interruption. Here are a couple of other examples of exploiting exceptions in functional programming:

- You are designing a program in which a function `f` must never be called with a negative integer (which is an invariant of `f`). You raise an exception at the entry point of `f` if its argument is found to be a negative integer.

- All SML programs that you hand in for programming assignments should compile; otherwise you will receive no credit for your hard work. Now you have finished implementing a function `funEasy` but not another function `funHard`, both of which are part of the assignment. Instead of forfeiting points for `funEasy`, you submit the following code for `funHard`, which instantly makes the whole program compile:

  ```
  exception NotImplemented
  fun funHard _ = raise NotImplemented
  ```

  This trick works because `raise NotImplemented` has type `'a`.

1.12 Modules

Modular programming is a methodology for software development in which a large program is partitioned into independent smaller units. Each unit contains a set of related functions, types, *etc.* that can be readily reused in different programming tasks. SML provides a strong support for modular programming with *structures* and *signatures*. A structure, the unit of modular programming in SML, is a collection of declarations satisfying the specification given in a signature.

A structure is a collection of functions, types, exceptions, and other elements enclosed within a struct — end construct; a signature is a collection of specifications on these declarations enclosed within a sig — end construct. For example, the structure in the left conforms to the signature in the right:

```
struct                             sig
  type 'a set = 'a list              type 'a set
  val emptySet : 'a set = nil        val emptySet : 'a set
  fun singleton x = [x]              val singleton : 'a -> 'a set
  fun union s1 s2 = s1 @ s2          val union : 'a set -> 'a set -> 'a set
end                                end
```

The first line in the signature states that a type declaration of 'a set must be given in a structure matching it; any type declaration resulting in a new type 'a set is acceptable. In the example above, we use a type declaration using the keyword *type*, but a datatype declaration like

```
    datatype 'a set = Empty | Singleton of 'a | Union of 'a set * 'a set
```

is also fine, if other elements in the same structure are redefined accordingly. The second line in the signature states that a variable emptySet of type 'a set must be defined in a structure matching it. The structure defines a variable emptySet of type 'a list, which coincides with 'a set under the definition of 'a set. The third line in the signature states that a variable singleton of type 'a -> 'a set, or equivalently a function singleton of type 'a -> 'a set, must be defined in a structure matching it. Again singleton in the structure has type 'a -> 'a list which is equal to 'a -> 'a set under the definition of 'a set. The case for union is similar.[5]

Like ordinary values, structures and signatures can be given names. We use the keywords structure and signature as illustrated below:

```
    structure Set =        signature SET =
    struct                 sig
      ...                    ...
    end                    end
```

Elements of the structure Set can then be accessed using the . notation familiar from the C language (*e.g.*, Set.set, Set.emptySet, ···).

Now how can we specify that the structure Set conforms to the signature SET? One way to do this is to impose a *transparent constraint* between Set and SET using a colon (:):

```
    structure Set : SET = ...
```

The constraint by : says that Set conforms to SET; the program does not compile if Set fails to implement some specification in SET. Another way is to impose an *opaque constraint* between Set and SET using the symbol :>:

```
    structure Set :> SET = ...
```

The constraint by :> says not only that Set conforms to SET but also that only those type declarations explicitly mentioned in SET are visible to the outside. To clarify their difference, consider the following code:

---

[5]@ is an infix operator concatenating two lists.

```
signature S = sig          structure Transparent : S =
   type t                  struct
end                           type t = int
                              val x = 1
                           end

                           structure Opaque :> S =
                           struct
                              type t = int
                              val x = 1
                           end
```

First note that both structures `Transparent` and `Opaque` conform to signature `S`. Since `S` does not declare variable `x`, there is no way to access `Transparent.x` and `Opaque.x`. The difference between `Transparent` and `Opaque` lies in the visibility of the definition of type `t`. In the case of `Transparent`, the definition of `t` as `int` is exported to the outside. Thus the following declaration is accepted because it is known that `Transparent.t` is indeed `int`:

```
- val y : Transparent.t = 1;
val y = 1 : Transparent.t
```

In the case of `Opaque`, however, the definition of `t` remains unknown to the outside, which causes the following declaration to be rejected:

```
- val z : Opaque.t = 1;
stdIn:3.5-3.21 Error: <some error message>
```

An opaque constraint in SML allows programmers to achieve data abstraction by hiding details of the implementation of a structure. In order to use structures given opaque constraints (*e.g.*, those included in the SML basis library or written by other programmers), therefore, you only need to read their signatures to see what values are exported. Often times you will see detailed documentation in signatures but no documentation in structures, for which there is a good reason.

SML also provides an innovative feature called *functors* which can be thought of as functions on structures. A functor takes as input a structure of a certain signature and generates as output a fresh structure specialized for the input structure. Since all structures generated by a functor share the same piece of code found in its definition, it enhances code reuse for modular programming.

To illustrate the use of functors, consider a signature for sets of values with an order relation:

```
datatype order = LESS | EQUAL | GREATER

signature ORD_SET =
sig
  type item                          (* type of elements *)
  type set                           (* type of sets *)
  val compare : item * item -> order  (* order relation *)
  val empty : set                    (* empty set *)
  val add : set -> item -> set       (* add an element *)
  val remove : set -> item -> set    (* remove an element *)
end
```

Function `compare` compares two values of type `item` to determine their relative size (less-than, equal, greater-than), and thus specifies an order relation on type `item`. A structure implementing the signature `ORD_SET` may take advantage of such an order relation on type `item`. For example, it may define `set` as `item list` with an invariant that values in every set are stored in ascending order with respect to `compare`, and exploit the invariant in implementing operations on `set`.

Now let us consider two structures of signature `ORD_SET`:

```
structure IntSet : ORD_SET =        structure StringSet : ORD_SET =
struct                              struct
```

```
   type item = int                    type item = string
   type set = item list               type set = item list
   fun compare (x, y) =               fun compare (x, y) =
     if x < y then LESS                 if String.< (x, y) then LESS
     else if x > y then GREATER         else if String.> (x, y) then GREATER
     else EQUAL                         else EQUAL
   val empty = []                     val empty = []
   fun add s x = ...                  fun add s x = ...
   fun remove s x = ...               fun remove s x = ...
 end                               end
```

If the two structures assume the same invariant on type set (*e.g.,* values are stored in ascending order), code for functions add and remove can be identical in both structures. Then the two structures may share the same piece of code except for the definition of type item and function compare. Functors enhance code reuse in such a case by enabling programmers to write a common piece of code for both structures just once.

Here is a functor that generates IntSet and StringSet when given appropriate structures as input. First we define a signature ORD_KEY for input structures in order to provide types or values specific to IntSet and StringSet:

```
    signature ORD_KEY =
    sig
      type ord_key
      val compare : ord_key * ord_key -> order
    end
```

A functor OrdSet takes a structure OrdKey of signature ORD_KEY and generates a structure of signature ORD_SET:

```
functor OrdSet (OrdKey : ORD_KEY) : ORD_SET =
struct
  type item = OrdKey.ord_key
  type set = item list
  val compare = OrdKey.compare
  val empty = []
  fun add _ _ = ...
  fun remove _ _ = ...
end
```

In order to generate IntSet and StringSet, we need corresponding structures of signature ORD_KEY:

```
structure IntKey : ORD_KEY =        structure StringKey : ORD_KEY =
struct                              struct
  type ord_key = int                  type ord_key = string
  fun compare (x, y) =               fun compare (x, y) =
    if x < y then LESS                 if String.< (x, y) then LESS
    else if x > y then GREATER         else if String.> (x, y) then GREATER
    else EQUAL                         else EQUAL
end                                 end
```

When given IntKey and StringKey as input, OrdSet generates corresponding structures of signature ORD_SET:

```
structure IntSet = OrdSet (IntKey)
structure StringSet = OrdSet (StringKey)
```

Chapter 2

Inductive Definitions

This chapter discusses *inductive definitions* which are an indispensable tool in the study of programming languages. The reason why we need inductive definitions is not difficult to guess: a programming language may be thought of a system that is inhabited by *infinitely* many elements (or programs), and we wish to give a complete specification of it with a *finite* description; hence we need a mechanism of inductive definition by which a finite description is capable of yielding an infinite number of elements in the system. Those techniques related to inductive definitions also play a key role in investigating properties of programming languages. We will study these concepts with a few simple languages.

2.1 Inductive definitions of syntactic categories

An integral part of the definition of a programming language is its *syntax* which answers the question of which program (*i.e.*, a sequence of characters) is recognizable by the parser and which program is not. Typically the syntax is specified by a number of *syntactic categories* such as expressions, types, and patterns. Below we discuss how to define syntactic categories inductively in a few simple languages.

Our first example defines a syntactic category nat of natural numbers:

$$\text{nat} \qquad n \quad ::= \quad \text{O} \mid \text{S}\, n$$

Here nat is the name of the syntactic category being defined, and n is called a *non-terminal*. We read ::= as *"is defined as"* and | as *"or."* O stands for "zero" and S "successor." Thus the above definition is interpreted as:

> A natural number n is either O or S n' where n' is another natural number.

Note that nat is defined inductively: a natural number S n' uses another natural number n', and thus nat uses the same syntactic category in its definition. Now the definition of nat produces an infinite collection of natural numbers such as

$$\text{O, S O, S S O, S S S O, S S S O, } \cdots.$$

Thus nat specifies a language of natural numbers.

A syntactic category may refer to another syntactic category in its definition. For example, given the above definition of nat, the syntactic category tree below uses nat in its inductive definition:

$$\text{tree} \qquad t \quad ::= \quad \text{leaf}\, n \mid \text{node}\, (t, n, t)$$

leaf n represents a leaf node with a natural number n; node (t_1, n, t_2) represents an internal node with a natural number n, a left child t_1, and a right child t_2. Then tree specifies a language of regular binary trees of natural numbers such as

$$\text{leaf}\, n, \text{ node}\, (\text{leaf}\, n_1, n, \text{leaf}\, n_2), \text{ node}\, (\text{node}\, (\text{leaf}\, n_1, n, \text{leaf}\, n_2), n', \text{leaf}\, n''), \cdots.$$

A similar but intrinsically different example is two syntactic categories that are mutually inductively defined. For example, we simultaneously define two syntactic categories even and odd of even and odd numbers as follows:

$$\begin{array}{llll} \text{even} & e & ::= & \mathtt{0} \mid \mathtt{S}\, o \\ \text{odd} & o & ::= & \mathtt{S}\, e \end{array}$$

According to the definition above, even consists of even numbers such as

$$\mathtt{0},\ \mathtt{S\,S\,0},\ \mathtt{S\,S\,S\,S\,0},\ \cdots$$

whereas odd consists of odd numbers such as

$$\mathtt{S\,0},\ \mathtt{S\,S\,S\,0},\ \mathtt{S\,S\,S\,S\,S\,0},\ \cdots .$$

Note that even and odd are *subcategories* of nat because every even number e or odd number o is also a natural number. Thus we may think of even and odd as nat satisfying certain properties.

Exercise 2.1. Define even and odd independently of each other.

Let us consider another example of defining a syntactic subcategory. First we define a syntactic category paren of strings of parentheses:

$$\begin{array}{llll} \text{paren} & s & ::= & \epsilon \mid (s\,)s \end{array}$$

ϵ stands for the empty string (*i.e.*, $\epsilon s = s = s\epsilon$). paren specifies a language of strings of parentheses with no constraint on the use of parentheses. Now we define a subcategory mparen of paren for those strings of matched parentheses:

$$\begin{array}{llll} \text{mparen} & s & ::= & \epsilon \mid (s) \mid s\,s \end{array}$$

mparen generates such strings as

$$\epsilon\,,()\,,()()\,,(())\,,(())()\,,()()()\,,\cdots .$$

mparen is ambiguous in the sense that a string belonging to mparen may not be decomposed in a unique way (according to the definition of mparen). For example, ()()() may be thought of as either ()() concatenated with () or () concatenated with ()(). The culprit is the third case $s\,s$ in the definition: for a sequence of substrings of matched parentheses, there can be more than one way to split it into two substrings of matched parentheses. An alternative definition of lparen below eliminates ambiguity in mparen:

$$\begin{array}{llll} \text{lparen} & s & ::= & \epsilon \mid (s)\,s \end{array}$$

The idea behind lparen is that the first parenthesis in a non-empty string s is a left parenthesis "(" which is paired with a unique occurrence of a right parenthesis ")". For example, $s = (())()$ can be written as $(s_1)s_2$ where $s_1 = ()$ and $s_2 = ()$, both strings of matched parentheses, are uniquely determined by s. ()) and (()(), however, are not strings of matched parentheses and cannot be written as $(s_1)s_2$ where both s_1 and s_2 are strings of matched parentheses.

An inductive definition of a syntactic category is a convenient way to specify a language. Even the syntax of a full-scale programming language (such as SML) uses essentially the same machinery. It is, however, not the best choice for *investigating properties of languages.* For example, how can we formally express that n belongs to nat if $\mathtt{S}\,n$ belongs to nat, let alone prove it? Or how can we show that a string belonging to mparen indeed consists of matched parentheses? The notion of *judgment* comes into play to address such issues arising in inductive definitions.

2.2 Inductive definitions of judgments

A judgment is an object of knowledge, or simply a statement, that may or may not be provable. Here are a few examples:

- "$1 - 1$ *is equal to* 0" is a judgment which is always provable.

- "$1 + 1$ *is equal to* 0" is also a judgment which is never provable.

- *"It is raining"* is a judgment which is sometimes provable and sometimes not.

- "`S S 0` *belongs to the syntactic category* `nat`" is a judgment which is provable if `nat` is defined as shown in the previous section.

Then how do we prove a judgment? For example, on what basis do we assert that "$1 - 1$ *is equal to* 0" is always provable? We implicitly use arithmetic to prove "$1 - 1$ *is equal to* 0", but strictly speaking, arithmetic rules are not given for free — we first have to reformulate them as *inference rules*.

An inference rule consists of premises and a conclusion, and is written in the following form (where J stands for a judgment):

$$\frac{J_1 \quad J_2 \quad \cdots \quad J_n}{J} \ R$$

The inference rule, whose name is R, states that if J_1 through J_n (premises) hold, then J (conclusion) also holds. As a special case, an inference rule with no premise (*i.e.*, $n = 0$) is called an *axiom*. Here are a few examples of inference rules and axioms where we omit their names:

$$\frac{m \text{ is equal to } l \quad l \text{ is equal to } n}{m \text{ is equal to } n} \qquad \frac{m \text{ is equal to } n}{m + 1 \text{ is equal to } n + 1}$$

$$\frac{}{n \text{ is equal to } n} \qquad \frac{}{0 \text{ is a natural number}} \qquad \frac{My \text{ coat is wet}}{It \text{ is raining}}$$

Judgments are a general concept that covers any form of knowledge: knowledge about weather, knowledge about numbers, knowledge about programming languages, and so on. Note that judgments alone are inadequate to justify the knowledge being conveyed — we also need inference rules for proving or refuting judgments. In other words, the definition of a judgment is complete *only when there are inference rules for proving or refuting it*. Without inference rules, there can be no meaning in the judgment. For example, without arithmetic rules, the statement "$1 - 1$ *is equal to* 0" is nothing more than nonsense and thus cannot be called a judgment.

Needless to say, judgments are a concept strong enough to express membership in a syntactic category. As an example, let us recast the inductive definition of nat as a system of judgments and inference rules. We first introduce a judgment n nat:

$$n \text{ nat} \qquad \Leftrightarrow \qquad n \text{ is a natural number}$$

We use the following two inference rules to prove the judgment n nat where their names, *Zero* and *Succ*, are displayed:

$$\frac{}{\texttt{0 nat}} \ Zero \qquad \frac{n \text{ nat}}{\texttt{S } n \text{ nat}} \ Succ$$

n in the rule *Succ* is called a *metavariable* which is just a placeholder for another sequence of `0` and `S` and is thus *not* part of the language consisting of `0` and `S`. That is, n is just a (meta)variable which ranges over the set of sequences of `0` and `S`; n itself (before being replaced by `S 0`, for example) is not tested for membership in `nat`.

The notion of metavariable is similar to the notion of variable in SML. Consider an SML expression `x = 1` where `x` is a variable of type `int`. The expression makes sense only because we read `x` as a variable that ranges over integer values and is later to be replaced by an actual integer constant. If we literally read `x` as an (ill-formed) integer, `x = 1` would always evaluate to `false` because `x`, as an integer constant, is by no means equal to another integer constant `1`.

The judgment n nat is now defined inductively by the two inference rules. The rule *Zero* is a base case because it is an axiom, and the rule *Succ* is an inductive case because the premise contains a judgment smaller in size than the one (of the same kind) in the conclusion. Now we can prove, for example, that `S S 0` nat holds with the following *derivation tree*, in which `S S 0` nat is the root and `0` nat is the only leaf (*i.e.*, it is an inverted tree):

$$\frac{\dfrac{\dfrac{}{\texttt{0 nat}} \ Zero}{\texttt{S 0 nat}} \ Succ}{\texttt{S S 0 nat}} \ Succ$$

Similarly we can rewrite the definition of the syntactic category tree in terms of judgments and inference rules:

$$t \text{ tree} \quad \Leftrightarrow \quad t \text{ is a regular binary tree of natural numbers}$$

$$\frac{n \text{ nat}}{\texttt{leaf } n \text{ tree}} \; Leaf \qquad \frac{t_1 \text{ tree} \quad n \text{ nat} \quad t_2 \text{ tree}}{\texttt{node } (t_1, n, t_2) \text{ tree}} \; Node$$

A slightly more complicated example is a judgment that isolates full regular binary trees of natural numbers, as shown below. Note that there is no restriction on the form of judgment as long as its meaning is clarified by inference rules. We may even use English sentences as a valid form of judgment!

$$t \text{ ctree}\langle d \rangle \quad \Leftrightarrow \quad t \text{ is a full regular binary tree of natural numbers of depth } d$$

$$\frac{n \text{ nat}}{\texttt{leaf } n \text{ ctree}\langle 0 \rangle} \; Cleaf \qquad \frac{t_1 \text{ ctree}\langle d \rangle \quad n \text{ nat} \quad t_2 \text{ ctree}\langle d \rangle}{\texttt{node } (t_1, n, t_2) \text{ ctree}\langle S\ d \rangle} \; Cnode$$

The following derivation tree proves that

$$
\begin{array}{c}
\texttt{0} \\
\diagup \quad \diagdown \\
\texttt{0} \qquad \texttt{0} \\
\diagup \diagdown \quad \diagup \diagdown \\
\texttt{0} \quad \texttt{0} \quad \texttt{0} \quad \texttt{0}
\end{array}
$$
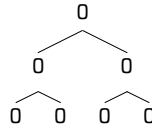
is a full regular binary tree of depth S S 0:

$$\frac{\dfrac{\overline{\texttt{0 nat}}\; Zero}{\texttt{leaf 0 ctree}\langle 0 \rangle}\; Cleaf \quad \overline{\texttt{0 nat}}\; Zero \quad \dfrac{\overline{\texttt{0 nat}}\; Zero}{\texttt{leaf 0 ctree}\langle 0 \rangle}\; Cleaf}{\texttt{node (leaf 0, 0, leaf 0) ctree}\langle S\ 0 \rangle}\; Cnode \quad \overline{\texttt{0 nat}} \quad (omitted)}{\texttt{node (node (leaf 0, 0, leaf 0), 0, node (leaf 0, 0, leaf 0)) ctree}\langle S\ S\ 0 \rangle}\; Cnode$$

We can also show that $t = \texttt{node (leaf 0, 0, node (leaf 0, 0, leaf 0))}$ is not a full regular binary tree as we cannot prove $t \text{ ctree}\langle d \rangle$ for any natural number d:

$$\frac{\dfrac{\overline{\texttt{0 nat}} \quad d' = \texttt{0}}{\texttt{leaf 0 ctree}\langle d' \rangle}\; Cleaf \quad \overline{\texttt{0 nat}}\; Zero \quad \dfrac{\cdots \quad d' = \texttt{S } d''}{\texttt{node (leaf 0, 0, leaf 0) ctree}\langle d' \rangle}\; Cnode}{\texttt{node (leaf 0, 0, node (leaf 0, 0, leaf 0)) ctree}\langle S\ d' \rangle}\; Cnode$$

It is easy to see why the proof fails: the left subtree of t requires $d' = \texttt{0}$ while the right subtree of t requires $d' = \texttt{S } d''$, and there is no way to solve two conflicting equations on d'.

As with the syntactic categories even and odd, multiple judgments can be defined simultaneously. For example, here is the translation of the definition of even and odd into judgments and inference rules:

$$\begin{array}{ccl} n \text{ even} & \Leftrightarrow & n \text{ is an even number} \\ n \text{ odd} & \Leftrightarrow & n \text{ is an odd number} \end{array}$$

$$\frac{}{\texttt{0 even}}\; ZeroE \qquad \frac{n \text{ odd}}{\texttt{S } n \text{ even}}\; SuccE \qquad \frac{n \text{ even}}{\texttt{S } n \text{ odd}}\; SuccO$$

The following derivation tree proves that S S 0 is an even number:

$$\frac{\dfrac{\dfrac{}{\texttt{0 even}}\; ZeroE}{\texttt{S 0 odd}}\; SuccO}{\texttt{S S 0 even}}\; SuccE$$

Exercise 2.2. Translate the definition of paren, mparen, and lparen into judgments and inference rules.

2.3 Derivable rules and admissible rules

As shown in the previous section, judgments are defined with a certain (fixed) number of inference rules. When put together, these inference rules justify new inference rules which may in turn be added to the system. The new inference rules do *not* change the characteristics of the system because they can all be justified by the original inference rules, but may considerably facilitate the study of the system. For example, when multiplying two integers, we seldom employ the basic arithmetic rules, which can be thought of as original inference rules; instead we mostly use the rules of the multiplication table, which can be thought of as new inference rules.

There are two ways to introduce new inference rules: as *derivable rules* and as *admissible rules*. A derivable rule is one in which the gap between the premise and the conclusion can be bridged by a derivation tree. In other words, there always exists a sequence of inference rules that use the premise to prove the conclusion. As an example, consider the following inference rule which states that if n is a natural number, so is $\mathsf{S}\ \mathsf{S}\ n$:

$$\frac{n\ \mathsf{nat}}{\mathsf{S}\ \mathsf{S}\ n\ \mathsf{nat}}\ Succ2$$

The rule $Succ2$ is derivable because we can justify it with the following derivation tree:

$$\frac{\dfrac{n\ \mathsf{nat}}{\mathsf{S}\ n\ \mathsf{nat}}\ Succ}{\mathsf{S}\ \mathsf{S}\ n\ \mathsf{nat}}\ Succ$$

Now we may use the rule $Succ2$ as if it was an original inference rule; when asked to justify its use, we can just present the above derivation tree.

An admissible rule is one in which the premise implies the conclusion. That is, whenever the premise holds, so does the conclusion. A derivable rule is certainly an admissible rule because of the derivability of the conclusion from the premise. There are, however, admissible rules that are not derivable rules. (Otherwise why would we distinguish between derivable and admissible rules?) Consider the following inference rule which states that if $\mathsf{S}\ n$ is a natural number, so is n:

$$\frac{\mathsf{S}\ n\ \mathsf{nat}}{n\ \mathsf{nat}}\ Succ^{-1}$$

First observe that the rule $Succ^{-1}$ is not derivable: the only way to derive n nat from $\mathsf{S}\ n$ nat is by the rule $Succ$, but the premise of the rule $Succ$ is smaller than its conclusion whereas $\mathsf{S}\ n$ nat is larger than n nat. That is, there is no derivation tree like

$$\frac{\dfrac{\mathsf{S}\ n\ \mathsf{nat}}{\vdots}\ Succ^{-1}}{n\ \mathsf{nat}}\ Succ^{-1}\ .$$

Now suppose that the premise $\mathsf{S}\ n$ nat holds. Since the only way to prove $\mathsf{S}\ n$ nat is by the rule $Succ$, $\mathsf{S}\ n$ nat must have been derived from n nat as follows:

$$\frac{\dfrac{\vdots}{n\ \mathsf{nat}}}{\mathsf{S}\ n\ \mathsf{nat}}\ Succ$$

Then we can extract a smaller derivation tree $\displaystyle\frac{\vdots}{n\ \mathsf{nat}}$ which proves n nat. Hence the rule $Succ^{-1}$ is justified as an admissible rule.

An important property of derivable rules is that they remain valid even when the system is augmented with new inference rules. For example, the rule $Succ2$ remains valid no matter how many new inference rules are added to the system because the derivation of $\mathsf{S}\ \mathsf{S}\ n$ nat from n nat is always possible thanks to the rule $Succ$ (which is not removed from the system). In contrast, admissible rules may become invalid when new inference rules are introduced. For example, suppose that the system introduces a new (bizarre) inference rule:

$$\frac{n\ \mathsf{tree}}{\mathsf{S}\ n\ \mathsf{nat}}\ Bizarre$$

The rule *Bizarre* invalidates the previously admissible rule $Succ^{-1}$ because the rule $Succ$ is no longer the only way to prove S n nat and thus S n nat fails to guarantees n nat. Therefore the validity of an admissible rule must be checked each time a new inference rule is introduced.

Exercise 2.3. Is the rule $\dfrac{n \text{ even}}{\text{S S } n \text{ even}}$ $SuccE^2$ derivable or admissible? What about the rule $\dfrac{\text{S S } n \text{ even}}{n \text{ even}}$ $SuccE^{-2}$?

2.4 Inductive proofs

We have learned how to specify systems using inductive definitions of syntactic categories or judgments, or *inductive systems* of syntactic categories or judgments. While it is powerful enough to specify even full-scale programming languages (*i.e.*, their syntax and semantics), the mechanism of inductive definition alone is hardly useful unless the resultant system is shown to exhibit desired properties. That is, we cannot just specify a system using an inductive definition and then immediately use it without proving any interesting properties. For example, our intuition says that every string in the syntactic category mparen has the same number of left and right parentheses, but the definition of mparen itself does not automatically prove this property; hence we need to formally prove this property ourselves in order to use mparen as a language of strings of matched parentheses. As another example, consider the inductive definition of the judgments n even and n odd. The definition seems to make sense, but it still remains to formally prove that n in n even indeed represents an even number and n in n odd an odd number.

There is another important reason why we need to be able to prove properties of inductive systems. An inductive system is often so complex that its soundness, *i.e.*, its definition being devoid of any inconsistencies, may not be obvious at all. In such a case, we usually set out to prove a property that is supposed to hold in the system. Then each flaw in the definition that destroys the property, if any, manifests itself at some point in the proof (because it is impossible to complete the proof). For example, an expression in a functional language is supposed to evaluate to a value of the same type, but this property (called *type preservation*) is usually not obvious at all. By attempting to prove type preservation, we can either locate flaws in the definition or partially ensure that the system is sound. Thus proving properties of an inductive system is the most effective aid in fixing errors in the definition.

First we will study a principle called *structural induction* for proving properties of inductive systems of syntactic categories. Next we will study another principle called *rule induction* for proving properties of inductive systems of judgments. Since an inductive system of syntactic category is a simplified presentation of a corresponding inductive system of judgments, structural induction is in fact a special case of rule induction. Nevertheless structural induction deserves separate treatment because of the role of syntactic categories in the study of programming languages.

2.4.1 Structural induction

The principle of structural induction states that a property of a syntactic category may be proven inductively by analyzing the structure of its definition: for each base case, we show that the property holds without making any assumption; for each inductive case, we first assume that the property holds for each smaller element in it and then prove the property holds for the entire case.

A couple of examples will clarify the concept. Consider the syntactic category nat of natural numbers. We wish to prove that $P(n)$ holds for every natural number n. Examples of $P(n)$ are:

- n has a successor.

- n is 0 or has a predecessor n' (*i.e.*, S $n' = n$).

- n is a product of prime numbers (where definitions of products and prime numbers are assumed to be given).

By structural induction, we prove the following two statements:

- $P(0)$ holds.

- If $P(n)$ holds, then $P(\text{S } n)$ also holds.

The first statement is concerned with the base case in which 0 has no smaller element in it; hence we prove $P(0)$ without any assumption. The second statement is concerned with the inductive case in which S n has a smaller element n in it; hence we first assume, as an *induction hypothesis*, that $P(n)$ holds and then prove that $P(\text{S } n)$ holds. The above instance of structural induction is essentially the same as the principle of mathematical induction.

As another example, consider the syntactic category tree of regular binary trees. In order to prove that $P(t)$ holds for every regular binary tree t, we need to prove the following two statements:

- $P(\text{leaf } n)$ holds.

- If $P(t_1)$ and $P(t_2)$ hold as induction hypotheses, then $P(\text{node } (t_1, n, t_2))$ also holds.

The above instance of structural induction is usually called *tree induction*.

As a concrete example of an inductive proof by structural induction, let us prove that every string belonging to the syntactic category mparen has the same number of left and right parentheses. (Note that we are not proving that mparen specifies a language of strings of matched parentheses.) We first define two auxiliary functions *left* and *right* to count the number of left and right parentheses. For visual clarity, we write $left[s]$ and $right[s]$ instead of $left(s)$ and $right(s)$. (We do not define *left* and *right* on the syntactic category paren because the purpose of this example is to illustrate structural induction rather than to prove an interesting property of mparen.)

$$
\begin{aligned}
left[\epsilon] &= 0 \\
left[(s)] &= 1 + left[s] \\
left[s_1 \ s_2] &= left[s_1] + left[s_2] \\
right[\epsilon] &= 0 \\
right[(s)] &= 1 + right[s] \\
right[s_1 \ s_2] &= right[s_1] + right[s_2]
\end{aligned}
$$

Now let us interpret $P(s)$ as "$left[s] = right[s]$." Then we want to prove that if s belongs to mparen, written as $s \in$ mparen, then $P(s)$ holds.

Theorem 2.4. *If $s \in$ mparen, then $left[s] = right[s]$.*

Proof. By structural induction on s.

Each line below corresponds to a single step in the proof. It is written in the following format:

conclusion   *justification*

This format makes it easy to read the proof because in most cases, we want to see the conclusion first rather than its justification.

Case $s = \epsilon$:
$left[\epsilon] = 0 = right[\epsilon]$

Case $s = (s')$:
$left[s'] = right[s']$ by induction hypothesis on s'
$left[s] = 1 + left[s'] = 1 + right[s'] = right[s]$ from $left[s'] = right[s']$

Case $s = s_1 \ s_2$:
$left[s_1] = right[s_1]$ by induction hypothesis on s_1
$left[s_2] = right[s_2]$ by induction hypothesis on s_2
$left[s_1 \ s_2] = left[s_1] + left[s_2] = right[s_1] + right[s_2] = right[s_1 \ s_2]$
                         from $left[s_1] = right[s_1]$ and $left[s_2] = right[s_2]$
$\square$

In the proof above, we may also say "by induction on the structure of s" instead of "by structural induction on s."

2.4.2 Rule induction

The principle of rule induction is similar to the principle of structural induction except that it is applied to derivation trees rather than definitions of syntactic categories. Consider an inductive definition of a judgment J with two inference rules:

$$\frac{}{J_b}\ R_{\text{base}} \qquad \frac{J_1 \quad J_2 \quad \cdots \quad J_n}{J_i}\ R_{\text{ind}}$$

We want to show that whenever J holds, another judgment $P(J)$ holds where $P(J)$ is a new form of judgment parameterized over J. For example, when J is "n nat", $P(J)$ may be "either n even or n odd." To this end, we prove the following two statements:

- $P(J_b)$ holds.

- If $P(J_1)$, $P(J_2)$, $\cdots$, and $P(J_n)$ hold as induction hypotheses, then $P(J_i)$ holds.

By virtue of the first statement, the following inference rule makes sense because we can always prove $P(J_b)$:

$$\frac{}{P(J_b)}\ R'_{\text{base}}$$

The following inference rule also makes sense because of the second statement: it states that if $P(J_1)$ through $P(J_n)$ hold, then $P(J_i)$ also holds, which is precisely what the second statement proves:

$$\frac{P(J_1) \quad P(J_2) \quad \cdots \quad P(J_n)}{P(J_i)}\ R'_{\text{ind}}$$

Now, for any derivation tree for J using the rules R_{base} and R_{ind}, we can prove $P(J)$ using the rules R'_{base} and R'_{ind}:

$$\frac{}{J_b}\ R_{\text{base}} \qquad \Longrightarrow \qquad \frac{}{P(J_b)}\ R'_{\text{base}}$$

$$\frac{\vdots \quad \vdots \quad \cdots \quad \vdots}{\dfrac{J_1 \quad J_2 \qquad J_n}{J_i}}\ R_{\text{ind}} \qquad \Longrightarrow \qquad \frac{\vdots \qquad \vdots \quad \cdots \qquad \vdots}{\dfrac{P(J_1) \quad P(J_2) \qquad P(J_n)}{P(J_i)}}\ R'_{\text{ind}}$$

In other words, J always implies $P(J)$. A generalization of the above strategy is the principle of rule induction.

As a trivial example, let us prove that n nat implies either n even or n odd. We let $P(n$ nat$)$ be "either n even or n odd" and apply the principle of rule induction. The two rules *Zero* and *Succ* require us to prove the following two statements:

- $P(0$ nat$)$ holds. That is, for the case where the rule *Zero* is used to prove n nat, we have $n = 0$ and thus prove $P(0$ nat$)$.

- If $P(n'$ nat$)$ holds, $P(\text{S } n'$ nat$)$ holds. That is, for the case where the rule *Succ* is used to prove n nat, we have $n = \text{S } n'$ and thus prove $P(\text{S } n'$ nat$)$ using the induction hypothesis $P(n'$ nat$)$.

According to the definition of $P(J)$, the two statements are equivalent to:

- Either 0 even or 0 odd holds.

- If either n' even or n' odd holds, then either S n' even or S n' odd holds.

A formal inductive proof proceeds as follows:

Theorem 2.5. *If n nat, then either n even or n odd.*

Proof. By rule induction on the judgment n nat.

It is of utmost importance that we apply the principle of rule induction to the *judgment* n nat rather than the natural number n. In other words, we analyze the structure of the proof of n nat, *not the structure of n*. If we analyze the structure of n, the proof degenerates to an example of structural induction! Hence we may also say "by induction on the structure of the proof of n nat" instead of "by rule induction on the judgment n nat."

Case $\dfrac{}{\text{0 nat}}\ Zero$ (where n happens to be equal to 0):

(This is the case where n nat is proven by applying the rule *Zero*. It is not obtained as a case where n is equal to 0, since we are not analyzing the structure of n. Note also that we do *not* apply the induction hypothesis because the premise has no judgment.)

0 even   by the rule $ZeroE$

Case $\dfrac{n' \text{ nat}}{\text{S } n' \text{ nat}}\ Succ$ (where n happens to be equal to S n'):

(This is the case where n nat is proven by applying the rule *Succ*.)

n' even or n' odd   by induction hypothesis
S n' odd or S n' even   by the rule $SuccO$ or $SuccE$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

Rule induction can also be applied simultaneously to two or more judgments. As an example, let us prove that n in n even represents an even number and n in n odd an odd number. We use the rules $ZeroE$, $SuccE$, and $SuccO$ in Section 2.2 along with the following inference rules using a judgment n double n':

$$\frac{}{\text{0 double 0}}\ Dzero \qquad \frac{n \text{ double } n'}{\text{S } n \text{ double S S } n'}\ Dsucc$$

Intuitively n double n' means that n' is a double of n (*i.e.*, $n' = 2 \times n$). The properties of even and odd numbers are stated in the following theorem:

Theorem 2.6.
  If n even, then there exists n' such that n' double n.
  If n odd, then there exist n' and n'' such that n' double n'' and S $n'' = n$.

The proof of the theorem follows the same pattern of rule induction as in previous examples except that $P(J)$ distinguishes between the two cases $J = n$ even and $J = n$ odd:

- $P(n \text{ even})$ is "there exists n' such that n' double n."

- $P(n \text{ odd})$ is "there exist n' and n'' such that n' double n'' and S $n'' = n$."

An inductive proof of the theorem proceeds as follows:

Proof of Theorem 2.6. By simultaneous rule induction on the judgments n even and n odd.

Case $\dfrac{}{\text{0 even}}\ ZeroE$ where $n = 0$:

0 double 0   by the rule $Dzero$
We let $n' = 0$.

Case $\dfrac{n_p \text{ odd}}{\text{S } n_p \text{ even}}\ SuccE$ where $n = \text{S } n_p$:

n_p' double n_p'' and S $n_p'' = n_p$ by induction hypothesis
S n_p' double S S n_p'' by the rule $Dsucc$ with n_p' double n_p''
S n_p' double n from S S $n_p'' = \text{S } n_p = n$
We let $n' = \text{S } n_p'$.

Case $\dfrac{n_p \text{ even}}{\text{S } n_p \text{ odd}}\ SuccO$ where $n = \text{S } n_p$:

n_p' double n_p by induction hypothesis
We let $n' = n_p'$ and $n'' = n_p$ from $n = \text{S } n_p$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

2.5 Techniques for inductive proofs

An inductive proof is not always as straightforward as the proof of Theorem 2.5. For example, the theorem being proven may be simply false! In such a case, the proof attempt (which will eventually fail) may help us to extract a counterexample of the theorem. If the theorem is indeed provable (or is believed to be provable) but a direct proof attempt fails, we can try a common technique for inductive proofs. Below we illustrate three such techniques: introducing a lemma, generalizing the theorem, and proving by the principle of inversion.

2.5.1 Using a lemma

We recast the definition of the syntactic categories mparen and lparen as a system of judgments and inference rules:

$$\frac{}{\epsilon \text{ mparen}} \; Meps \qquad \frac{s \text{ mparen}}{(s) \text{ mparen}} \; Mpar \qquad \frac{s_1 \text{ mparen} \quad s_2 \text{ mparen}}{s_1 \, s_2 \text{ mparen}} \; Mseq$$

$$\frac{}{\epsilon \text{ lparen}} \; Leps \qquad \frac{s_1 \text{ lparen} \quad s_2 \text{ lparen}}{(s_1) \, s_2 \text{ lparen}} \; Lseq$$

Our goal is to show that s mparen implies s lparen. It turns out that a direct proof attempt by rule induction fails and that we need a lemma. To informally explain why we need a lemma, consider the case where the rule $Mseq$ is used to prove s mparen. We may write $s = s_1 \, s_2$ with s_1 mparen and s_2 mparen. By induction hypothesis on s_1 mparen and s_2 mparen, we may conclude s_1 lparen and s_2 lparen. From s_1 lparen, there are two subcases to consider:

- If $s_1 = \epsilon$, then $s = s_1 \, s_2 = s_2$ and s_2 lparen implies s lparen.

- If $s_1 = (s_1') \, s_1''$ with s_1' lparen and s_1'' lparen, then $s = (s_1') \, s_1'' \, s_2$.

In the second subcase, it is necessary to prove $s_1'' \, s_2$ lparen from s_1'' lparen and s_2 lparen, which is not addressed by what is being proven (and is not obvious). Thus the following lemma needs to be proven first:

Lemma 2.7. *If s lparen and s' lparen, then $s \, s'$ lparen.*

Then how do we prove the above lemma by rule induction? The lemma does not seem to be provable by rule induction because it does not have the form *"If J holds, then $P(J)$ holds"* — the *If* part contains two judgments! It turns out, however, that rule induction can be applied exactly in the same way. The trick is to interpret the statement in the lemma as:

If s lparen, then s' lparen implies $s \, s'$ lparen.

Then we apply rule induction to the judgment s lparen with $P(s \text{ lparen})$ being "s' lparen implies $s \, s'$ lparen." An inductive proof of the lemma proceeds as follows:

Proof of Lemma 2.7. By rule induction on the judgment s lparen. Keep in mind that the induction hypothesis on s lparen yields "s' lparen implies $s \, s'$ lparen." Consequently, if s' lparen is already available as an assumption, the induction hypothesis on s lparen yields $s \, s'$ lparen.

Case $\dfrac{}{\epsilon \text{ lparen}} \; Leps$ where $s = \epsilon$:

s' lparen	assumption
$s \, s' = \epsilon \, s' = s'$	
$s \, s'$ lparen	from s' lparen

Case $\dfrac{s_1 \text{ lparen} \quad s_2 \text{ lparen}}{(s_1) \, s_2 \text{ lparen}} \; Lseq$ where $s = (s_1) \, s_2$:

s' lparen	assumption
$s \, s' = (s_1) \, s_2 \, s'$	
"s' lparen implies $s_2 \, s'$ lparen"	by induction hypothesis on s_2 lparen

s_2 s' lparen

(s_1) s_2 s' lparen

from the assumption s' lparen

by the rule $Lseq$ with s_1 lparen and s_2 s' lparen

$\square$

Exercise 2.8. Can you prove Lemma 2.7 by rule induction on the judgment s' lparen?

Now we are ready to prove that s mparen implies s lparen.

Theorem 2.9. *If s mparen, then s lparen.*

Proof. By rule induction on the the judgment s mparen.

Case $\dfrac{}{\epsilon \text{ mparen}}$ $Meps$ where $s = \epsilon$:

ϵ lparen $\hfill$ by the rule $Leps$

Case $\dfrac{s' \text{ mparen}}{(s') \text{ mparen}}$ $Mpar$ where $s = (s')$:

s' lparen $\hfill$ by induction hypothesis

(s') lparen $\hfill$ from $\dfrac{s' \text{ lparen} \quad \dfrac{}{\epsilon \text{ lparen}} Leps}{(s') \ \epsilon \text{ lparen}} Lseq$ and $(s') = (s') \ \epsilon$

Case $\dfrac{s_1 \text{ mparen} \quad s_2 \text{ mparen}}{s_1 \ s_2 \text{ mparen}}$ $Mseq$ where $s = s_1 \ s_2$:

s_1 lparen $\hfill$ by induction hypothesis on s_1 mparen

s_2 lparen $\hfill$ by induction hypothesis on s_2 mparen

$s_1 \ s_2$ lparen $\hfill$ by Lemma 2.7

$\square$

2.5.2 Generalizing a theorem

We have seen in Theorem 2.4 that if a string s belongs to the syntactic category mparen, or if s mparen holds, s has the same number of left and right parentheses, *i.e.*, $left[s] = right[s]$. The result, however, does not prove that s is a string of matched parentheses because it does not take into consideration positions of matching parentheses. For example, $s =)($ satisfies $left[s] = right[s]$, but is not a string of matched parentheses because the left parenthesis appears after its corresponding right parenthesis.

In order to be able to recognize strings of matched parentheses, we introduce a new judgment $k \rhd s$ where k is a non-negative integer:

$$k \rhd s \quad \Leftrightarrow \quad k \text{ left parentheses concatenated with } s \text{ form a string of matched parentheses}$$
$$\Leftrightarrow \quad \underbrace{((\cdots(}_{k} s \text{ is a string of matched parentheses}$$

The idea is that we scan a given string from left to right and keep counting the number of left parentheses that have not yet been matched with corresponding right parentheses. Thus we begin with $k = 0$, increment k each time a left parenthesis is encountered, and decrement k each time a right parenthesis is encountered:

$$\dfrac{}{0 \rhd \epsilon} Peps \qquad \dfrac{k+1 \rhd s}{k \rhd (s} Pleft \qquad \dfrac{k-1 \rhd s \quad k > 0}{k \rhd)s} Pright$$

The second premise $k > 0$ in the rule $Pright$ ensures that in any prefix of a given string, the number of right parentheses may not exceed the number of left parentheses. Now a judgment $0 \rhd s$ expresses that s is a string of matched parentheses. Here are a couple of examples:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0 \rhd \epsilon} Peps \quad 1 > 0}{1 \rhd)} Pright \quad 2 > 0}{2 \rhd))} Pright}{\dfrac{1 \rhd ())}{0 \rhd (())} Pleft} Pleft$$

$(\text{the rule } Pright \text{ is not applicable because } 0 \not> 0)$

$$\dfrac{\dfrac{0 \rhd)(}{1 \rhd))(} Pright}{0 \rhd ())(} Pleft$$

Note that while an inference rules is usually read from the premise to the conclusion, *i.e.*, "if the premise holds, then the conclusion follows," the above rules are best read from the conclusion to the premise: "in order to prove the conclusion, we prove the premise instead." For example, the rule *Peps* may be read as "in order to prove $0 \triangleright \epsilon$, we do not have to prove anything else," which implies that $0 \triangleright \epsilon$ automatically holds; the rule *Pleft* may be read as "in order to prove $k \triangleright (s$, we only have to prove $k + 1 \triangleright s$." This bottom-up reading of the rules corresponds to the left-to-right direction of scanning a string. For example, a proof of $0 \triangleright (())$ would proceed as the following sequence of judgments in which the given string is scanned from left to right:

$$0 \triangleright (()) \quad \longrightarrow \quad 1 \triangleright ()) \quad \longrightarrow \quad 2 \triangleright)) \quad \longrightarrow \quad 1 \triangleright) \quad \longrightarrow \quad 0 \triangleright \epsilon$$

Exercise 2.10. Rewrite the inference rules for the judgment $k \triangleright s$ so that they are best read from the premise to the conclusion.

Now we wish to prove that a string s satisfying $0 \triangleright s$ indeed belongs to the syntactic category mparen:

Theorem 2.11. *If $0 \triangleright s$, then s mparen.*

It is easy to see that a direct proof of Theorem 2.11 by rule induction fails. For example, when $0 \triangleright (s$ follows from $1 \triangleright s$ by the rule *Pleft*, we cannot apply the induction hypothesis to the premise because it does not have the form $0 \triangleright s'$. What we need is, therefore, a generalization of Theorem 2.11 that covers all cases of the judgment $k \triangleright s$ instead of a particular case $k = 0$:

Lemma 2.12. *If $k \triangleright s$, then $\underbrace{((\cdots (}_{k} s$ mparen.*

Lemma 2.12 formally verifies the intuition behind the general form of the judgment $k \triangleright s$. Then Theorem 2.11 is obtained as a corollary of Lemma 2.12.

The proof of Lemma 2.12 requires another lemma whose proof is left as an exercise (see Exercise 2.18):

Lemma 2.13. *If $\underbrace{((\cdots (}_{k} s$ mparen, then $\underbrace{((\cdots (()}_{k} s$ mparen.*

Proof of Lemma 2.12. By rule induction on the judgment $k \triangleright s$.

Case $\dfrac{}{0 \triangleright \epsilon}$ *Peps* where $k = 0$ and $s = \epsilon$:

ϵ mparen $\hfill$ by the rule *Meps*

$\underbrace{((\cdots (}_{k} s$ mparen $\hfill$ from $\underbrace{((\cdots (}_{k} s = \epsilon$

Case $\dfrac{k + 1 \triangleright s'}{k \triangleright (s'}$ *Pleft* where $s = (s'$:

$\underbrace{((\cdots (}_{k+1} s'$ mparen $\hfill$ by induction hypothesis on $k + 1 \triangleright s'$

$\underbrace{((\cdots (}_{k} s$ mparen $\hfill$ from $\underbrace{((\cdots (}_{k+1} s' = \underbrace{((\cdots ((}_{k} s' = \underbrace{((\cdots (}_{k} s$

Case $\dfrac{k - 1 \triangleright s' \quad k > 0}{k \triangleright)s'}$ *Pright* where $s =)s'$:

$\underbrace{((\cdots (}_{k-1} s'$ mparen $\hfill$ by induction hypothesis on $k - 1 \triangleright s'$

$\underbrace{((\cdots (()}_{k-1} s'$ mparen $\hfill$ by Lemma 2.13

$\underbrace{((\cdots (}_{k} s$ mparen $\hfill$ from $\underbrace{((\cdots (()}_{k-1} s' = \underbrace{((\cdots ()}_{k} s' = \underbrace{((\cdots (}_{k} s$ $\hfill\square$

It is important that generalizing a theorem is different from introducing a lemma. We introduce a lemma when the induction hypothesis is applicable to all premises in an inductive proof, but the conclusion to be drawn is not a direct consequence of induction hypotheses. Typically such a lemma,

which fills the gap between induction hypotheses and the conclusion, requires another inductive proof and is thus proven separately. In contrast, we generalize a theorem when the induction hypothesis is not applicable to some premises and an inductive proof does not even work. Introducing a lemma is to no avail here, since the induction hypothesis is applicable only to premises of inference rules and nothing else (*e.g.*, judgments proven by a lemma). Thus we generalize the theorem so that a direct inductive proof works. (The proof of the generalized theorem may require us to introduce a lemma, of course.)

To generalize a theorem is essentially to find a theorem that is harder to prove than, but immediately implies the original theorem. (In this regard, we can also say that we "strengthen" the theorem.) There is no particular recipe for generalizing a theorem, and some problem requires a deep insight into the judgment to which the induction hypothesis is to be applied. In many cases, however, identifying an invariant on the judgment under consideration gives a clue on how to generalize the theorem. For example, Theorem 2.11 deals with a special case of the judgment $k \triangleright s$, and its generalization in Lemma 2.12 precisely expresses what the judgment $k \triangleright s$ means.

2.5.3 Proof by the principle of inversion

Consider an inference rule $\dfrac{J_1 \quad J_2 \quad \cdots \quad J_n}{J} \ R$. In order to apply the rule R, we first have to establish proofs of all the premises J_1 through J_n, from which we may judge that the conclusion J also holds. An alternative way of reading the rule R is that in order to prove J, it suffices to prove $J_1, \cdots, J_n$. In either case, it is the premises, not the conclusion, that we have to prove first.

Now assume the existence of a proof of the conclusion J. That is, we assume that J is provable, but we may not have a concrete proof of it. Since the rule R is applied in the top-down direction, the existence of a proof of J does not license us to conclude that the premises $J_1, \cdots, J_n$ are also provable. For example, there may be another rule, say $\dfrac{J_1' \quad J_2' \quad \cdots \quad J_m}{J} \ R'$, that deduces the same conclusion, but using different premises. In this case, we cannot be certain that the rule R has been applied at the final step of the proof of J, and the existence of proofs of $J_1, \cdots, J_n$ is not guaranteed.

If, however, the rule R is the *only* way to prove the conclusion J, we may safely "invert" the rule R and deduce the premises $J_1, \cdots, J_n$ from the existence of a proof of J. That is, since the rule R is the only way to prove J, the existence of a proof of J is subject to the existence of proofs of all the premises of the rule R. Such a use of an inference rule in the bottom-up direction is called the *principle of inversion.*

As an example, let us prove that if S n is a natural number, so is n:

Proposition 2.14. *If* S n nat, *then* n nat.

We begin with an assumption that S n nat holds. Since the only way to prove S n nat is by the rule *Succ*, S n nat must have been derived from n nat by the principle of inversion:

$$\frac{n \ \mathsf{nat}}{\mathsf{S} \ n \ \mathsf{nat}} \ Succ$$

Thus there must be a proof of n nat whenever there exists a proof of S n nat, which completes the proof of Proposition 2.14.

2.6 Exercises

Exercise 2.15. Suppose that we represent a binary number as a sequence of digits **0** and **1**. Give an inductive definition of a syntactic category bin for positive binary numbers without a leading **0**. For example, **10** belongs to bin whereas **00** does not. Then define a function num which takes a sequence b belonging to bin and returns its corresponding decimal number. For example, we have $num(\mathbf{10}) = 2$ and $num(\mathbf{110}) = 6$. You may use ϵ for the empty sequence.

Exercise 2.16. Prove the converse of Theorem 2.9: if s lparen, then s mparen.

Exercise 2.17. Given a judgment t tree, we define two functions $numLeaf(t)$ and $numNode(t)$ for calculating the number of leaves and the number of nodes in t, respectively:

$$
\begin{aligned}
numLeaf(\texttt{leaf } n) &= 1 \\
numLeaf(\texttt{node } (t_1, n, t_2)) &= numLeaf(t_1) + numLeaf(t_2) \\
numNode(\texttt{leaf } n) &= 0 \\
numNode(\texttt{node } (t_1, n, t_2)) &= numNode(t_1) + numNode(t_2) + 1
\end{aligned}
$$

Use rule induction to prove that if t tree, then $numLeaf(t) - numNode(t) = 1$.

Exercise 2.18. Prove a lemma: if $\underbrace{((\cdots(}_{k} s \text{ lparen}$, then $\underbrace{((\cdots(()}_{k} s \text{ lparen}$. Use this lemma to prove Lemma 2.13. Your proof needs to exploit the equivalence between s mparen and s lparen as stated in Theorem 2.9 and Exercise 2.16.

Exercise 2.19. Proof the converse of Theorem 2.11: if s mparen, then $0 \triangleright s$.

Exercise 2.20. Consider an SML implementation of the factorial function:

```
fun fact' 0 a = a
  | fact' n a = fact' (n - 1) (n * a)
fun fact n = fact' n 1
```

We wish to prove that `fact` $\widehat{n}$ evaluates to $\widehat{n!}$ by mathematical induction on $n \geq 0$, where $\widehat{n}$ stands for an SML constant expression for a mathematical integer n. Since `fact` $\widehat{n}$ reduces to `fact'` $\widehat{n}$ `1`, we try to prove a lemma that `fact'` $\widehat{n}$ `1` evaluates to $\widehat{n!}$. Unfortunately it is impossible to prove the lemma by mathematical induction on n. How would you generalize the lemma so that mathematical induction works on n?

Exercise 2.21. The principle of mathematical induction states that for any natural number n, a judgment $P(n)$ holds if the following two conditions are met:

1. $P(0)$ holds.

2. $P(k)$ implies $P(k + 1)$ where $k \geq 0$.

There is another principle, called *complete induction*, which allows stronger assumptions in proving $P(k + 1)$:

1. $P(0)$ holds.

2. $P(0), P(1), \cdots, P(k)$ imply $P(k + 1)$ where $k \geq 0$.

It turns out that complete induction is not a new principle; rather it is a derived principle which can be justified by the principle of mathematical induction. Use mathematical induction to show that if the two conditions for complete induction are met, $P(n)$ holds for any natural number n.

Exercise 2.22. Consider the following inference rules for comparing two natural number for equality:

$$
\frac{}{0 \doteq 0} \; EqZero \qquad \frac{n \doteq m}{\texttt{S } n \doteq \texttt{S } m} \; EqSucc
$$

Show that the following inference rule is admissible:

$$
\frac{n \doteq m \quad n \text{ double } n' \quad m \text{ double } m'}{n' \doteq m'} \; EqDouble
$$

Exercise 2.23. Consider yet another inductive definition of strings of matched parentheses where we use a new judgment s tparen:

$$
\frac{}{\epsilon \text{ tparen}} \; Teps \qquad \frac{s_1 \text{ tparen} \quad s_2 \text{ tparen}}{s_1 \, (s_2) \text{ tparen}} \; Tseq
$$

Give a proof of the following lemma:

Lemma 2.24. *If s tparen and s' tparen, then $s\ s'$ tparen.*

Use Lemma 2.24 to prove the following theorem:

Theorem 2.25. *If s mparen, then s tparen.*

In conjunction with the result from Exercise 2.16, Theorem 2.25 proves that s lparen implies s tparen. Now we wish to prove this result directly, *i.e.*, without using the judgment s mparen:

Theorem 2.26. *If s lparen, then s tparen.*

Try to prove Theorem 2.26 by rule induction on s lparen. If you can complete the proof, write it. If you need a lemma to complete the proof, state the lemma, prove it, and use it to complete the proof.

Exercise 2.27. Consider two SML implementations of the Fibonacci function. `fib1` is an ordinary recursive function whereas `fib2` is a tail-recursive function. All arguments to both functions are assumed to be natural numbers (which include zero).

```
fun fib1 0 = 0
  | fib1 1 = 1
  | fib1 n = fib1 (n - 1) + fib1 (n - 2)      // n ≥ 2

fun fib2 m n 0 = m
  | fib2 m n p = fib2 n (m + n) (p - 1)        // p ≥ 1
```

Use mathematical induction or complete induction to prove the following properties. We assume that every variable (p, z, m, n) ranges over natural numbers.

- `fib2 (fib1 p) (fib1 (p + 1)) z = fib1 (z + p).`

- `fib2 m n (p + 1) = m * fib1 p + n * fib1 (p + 1).`

- `fib1 (2 * p) + fib1 p * fib1 p = 2 * fib1 p * fib1 (p + 1).`

- `fib1 (2 * p + 1) = fib1 p * fib1 p + fib1 (p + 1) * fib1 (p + 1).`

Chapter 3

λ-Calculus

This chapter presents the λ-*calculus*, a core calculus for functional languages (including SML of course). It captures the essential mechanism of computation in functional languages, and thus serves as an excellent framework for investigating basic concepts in functional languages. According to the Church-Turing thesis, the λ-calculus is equally expressive as Turing machines, but its syntax is deceptively simple. We first discuss the syntax and semantics of the λ-calculus and then show how to write programs in the λ-calculus.

Before we proceed, we briefly discuss the difference between *concrete syntax* and *abstract syntax*. Concrete syntax specifies which string of characters is accepted as a valid program (causing no syntax errors) or rejected as an invalid program (causing syntax errors). For example, according to the concrete syntax of SML, a string `~1` is interpreted as an integer -1, but a string `-1` is interpreted as an infix operator `-` applied to an integer argument `1` (which later causes a type error). A parser implementing concrete syntax usually translates source programs into tree structures. For example, a source program `1 + 2 * 3` is translated into

```
        +
      /   \
     1     *
          / \
         2   3
```
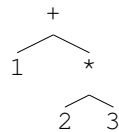
after taking into account operator precedence rules. Such tree structures are called *abstract syntax trees* which abstract away from details of parsing (such as operator precedence/associativity rules) and focus on the structure of source programs; abstract syntax is just the syntax for such tree structures.

While concrete syntax is an integral part of designing a programming language, we will not discuss it in this course. Instead we will work with abstract syntax to concentrate on computational aspects of programming languages. For example, we do not discuss why `1 + 2 * 3` and `1 + (2 * 3)`, both written in concrete syntax, are translated by the parser into the same abstract syntax tree shown above. For the purpose of understanding how their computation proceeds, the abstract syntax tree alone suffices.

3.1 Abstract syntax for the λ-calculus

The abstract syntax for the λ-calculus is given as follows:

$$\text{expression} \qquad e \quad ::= \quad x \mid \lambda x.\, e \mid e\, e$$

- An expression x is called a *variable*. We may use other names for variables (*e.g.*, z, s, t, f, arg, $accum$, and so on). Strictly speaking, therefore, x itself in the inductive definition of expression is a metavariable.

- An expression $\lambda x.\, e$ is called a λ-*abstraction*, or just a function, which denotes a mathematical function whose formal argument is x and whose body is e. We may think of $\lambda x.\, e$ as an internal representation of a nameless SML function `fn x => e` in abstract syntax.

We say that a variable x is *bound* in the λ-abstraction $\lambda x.\, e$ (just like a variable x is bound in an SML function `fn x => e`). Alternatively we can say that x is a bound variable in the body e.

- An expression $e_1\, e_2$ is called a *λ-application* or an *application* which denotes a function application (if e_1 is shown to be equivalent to a λ-abstraction somehow). We may think of $e_1\, e_2$ as an internal representation of an SML function application in abstract syntax. As in SML, applications are left-associative: $e_1\, e_2\, e_3$ means $(e_1\, e_2)\, e_3$ instead of $e_1\, (e_2\, e_3)$.

The scope of a λ-abstraction extends as far to the right as possible. Here are a few examples:

- $\lambda x.\, x\, y$ is the same expression as a λ-abstraction $\lambda x.\, (x\, y)$ whose body is $x\, y$. It should not be understood as an application $(\lambda x.\, x)\, y$.

- $\lambda x.\, \lambda y.\, x\, y$ is the same expression as $\lambda x.\, \lambda y.\, (x\, y) = \lambda x.\, (\lambda y.\, (x\, y))$. It should not be understood as an application $(\lambda x.\, \lambda y.\, x)\, y$.

As it turns out, every expression in the λ-calculus denotes a mathematical function. That is, the denotation of every expression in the λ-calculus is a mathematical function. Section 3.2 discusses how to determine unique mathematical functions corresponding to expressions in the λ-calculus, and in the present section, we develop the intuition behind the λ-calculus by considering a few examples of λ-abstractions.

Our first example is an identity function:

$$\mathsf{id} \quad = \quad \lambda x.\, x$$

id is an identity function because when given an argument x, it returns x without any further computation. Like higher-order functions in SML, a λ-abstraction may return another λ-abstraction as its result. For example, tt belows takes t to return another λ-abstraction $\lambda f.\, t$ which ignores its argument; ff below ignores its argument t to return a λ-abstraction $\lambda f.\, f$:

$$
\begin{aligned}
\mathsf{tt} &\;=\; \lambda t.\, \lambda f.\, t &\;=\;& \lambda t.\, (\lambda f.\, t)\\
\mathsf{ff} &\;=\; \lambda t.\, \lambda f.\, f &\;=\;& \lambda t.\, (\lambda f.\, f)
\end{aligned}
$$

Similarly a λ-abstraction may expect another λ-abstraction as its argument. For example, the λ-abstraction below expects another λ-abstraction s which is later applied to z:

$$\mathsf{one} \quad = \quad \lambda s.\, \lambda z.\, s\, z \quad = \quad \lambda s.\, (\lambda z.\, (s\, z))$$

3.2 Operational semantics of the λ-calculus

The semantics of a programming language answers the question of "what is the *meaning* of a given program?" This is an important question in the design of programming languages because lack of formal semantics implies potential ambiguities in interpreting programs. Put another way, lack of formal semantics makes it impossible to determine the meaning of certain programs. Surprisingly not every programming language has its semantics. For example (and perhaps to your surprise), the C language has no formal semantics — the same C program may exhibit different behavior depending on the state of the machine on which the program is executed.

There are three approaches to formulating the semantics of programming languages: *denotational semantics*, *axiomatic semantics*, and *operational semantics*. Throughout this course, we will use exclusively the operational semantics approach for its close connection with judgments and inference rules. The operational semantics approach is also attractive because it directly reflects the implementation of programming languages (*e.g.*, interpreters or compilers).

In general, the operational semantics of a programming language specifies how to transform a program into a *value* via a sequence of "operations." In the case of the λ-calculus, values consist of λ-abstractions and "operations" are called *reductions*. Thus the operational semantics of the λ-calculus specifies how to reduce an expression e into a value v where v is defined as follows:

$$\mathsf{value} \qquad v \quad ::= \quad \lambda x.\, e$$

Then we take v as the meaning of e. Since a λ-abstraction denotes a mathematical function, it follows that *every* expression in the λ-calculus denotes a mathematical function.

With this idea in mind, let us formally define reductions of expressions. We introduce a *reduction judgment* of the form $e \mapsto e'$:[1]

$$e \mapsto e' \qquad \Leftrightarrow \qquad e \text{ reduces to } e'$$

We write $\mapsto^*$ for the reflexive and transitive closure of $\mapsto$. That is, $e \mapsto^* e'$ holds if $e \mapsto e_1 \mapsto \cdots \mapsto e_n = e'$ where $n \geq 0$. (See Exercise 3.11 for a formal definition of $\mapsto^*$.) We say that e *evaluates* to v if $e \mapsto^* v$ holds.

Before we provide inference rules to complete the definition of the judgment $e \mapsto e'$, let us see what kind of expression can be reduced to another expression. Clearly variables and λ-abstractions cannot be further reduced:

$$x \not\mapsto \cdot$$
$$\lambda x.\, e \not\mapsto \cdot$$

($e \not\mapsto \cdot$ means that e does not reduce to another expression.) Then when can we reduce an application $e_1\, e_2$? If we think of it as an internal representation of an SML function application, we can reduce it only if e_1 represents an SML function. Thus the only candidate for reduction is an application of the form $(\lambda x.\, e_1')\, e_2$.

If we think of $\lambda x.\, e_1'$ as a mathematical function whose formal argument is x and whose body is e_1', the most natural way to reduce $(\lambda x.\, e_1')\, e_2$ is by substituting e_2 for every occurrence of x in e_1', or equivalently, by replacing every occurrence of x in e_1' by e_2. (For now, we do not consider the issue of whether e_2 is a value or not.) To this end, we introduce a *substitution* $[e'/x]e$:

> $[e'/x]e$ *is defined as an expression obtained by substituting e' for every occurrence of x in e.*

$[e'/x]e$ may also be read as "applying a substitution $[e'/x]$ to e." Then the following reduction is justified

$$(\lambda x.\, e)\, e' \mapsto [e'/x]e$$

where the expression being reduced, namely $(\lambda x.\, e)\, e'$, is called a *redex* (*red*ucible *ex*pression). For historical reasons, the above reduction is called a *β-reduction*.

Simple as it may seem, the precise definition of $[e'/x]e$ is remarkably subtle (see Section 3.3). For now, we just avoid complex examples whose reduction would require the precise definition of substitution. Here are a few examples of β-reductions; the redex for each step is underlined:

$$
\begin{aligned}
\underline{(\lambda x.\, x)\, (\lambda y.\, y)} &\mapsto \lambda y.\, y \\
(\lambda t.\, \lambda f.\, t)\, \underline{(\lambda x.\, x)\, (\lambda y.\, y)} &\mapsto \underline{(\lambda f.\, \lambda x.\, x)\, (\lambda y.\, y)} &\mapsto \lambda x.\, x \\
\underline{(\lambda t.\, \lambda f.\, f)\, (\lambda x.\, x)}\, (\lambda y.\, y) &\mapsto \underline{(\lambda f.\, f)\, (\lambda y.\, y)} &\mapsto \lambda y.\, y \\
\underline{(\lambda s.\, \lambda z.\, s\, z)\, (\lambda x.\, x)}\, (\lambda y.\, y) &\mapsto \underline{(\lambda z.\, (\lambda x.\, x)\, z)\, (\lambda y.\, y)} &\mapsto \underline{(\lambda x.\, x)\, (\lambda y.\, y)} \mapsto \lambda y.\, y
\end{aligned}
$$

The β-reduction is the basic principle for reducing expressions, but it does not yield unique inference rules for the judgment $e \mapsto e'$. That is, there can be more than one way to apply the β-reduction to an expression, or equivalently, an expression may contain multiple redexes in it. For example, $(\lambda x.\, x)\, ((\lambda y.\, y)\, (\lambda z.\, z))$ contains two redexes in it:

$$
\begin{aligned}
\underline{(\lambda x.\, x)\, ((\lambda y.\, y)\, (\lambda z.\, z))} &\mapsto \underline{(\lambda y.\, y)\, (\lambda z.\, z)} &\mapsto \lambda z.\, z \\
(\lambda x.\, x)\, \underline{((\lambda y.\, y)\, (\lambda z.\, z))} &\mapsto \underline{(\lambda x.\, x)\, (\lambda z.\, z)} &\mapsto \lambda z.\, z
\end{aligned}
$$

In the first case, the expression being reduced has the form $(\lambda x.\, e)\, e'$ and we immediately apply the β-reduction to the whole expression to obtain $[e'/x]e$. In the second case, we apply the β-reduction to e' which happens to be a redex; if e' was not a redex (*e.g.*, $e' = \lambda t.\, t$), the second case would be impossible. Here is another example of an expression containing two redexes:

$$
\begin{aligned}
\underline{(\lambda s.\, \lambda z.\, s\, z)\, ((\lambda x.\, x)\, (\lambda y.\, y))} &\mapsto \lambda z.\, \underline{((\lambda x.\, x)\, (\lambda y.\, y))}\, z &\mapsto \lambda z.\, \underline{(\lambda y.\, y)\, z} \mapsto \lambda z.\, z \\
(\lambda s.\, \lambda z.\, s\, z)\, \underline{((\lambda x.\, x)\, (\lambda y.\, y))} &\mapsto \underline{(\lambda s.\, \lambda z.\, s\, z)\, (\lambda y.\, y)} &\mapsto \lambda z.\, \underline{(\lambda y.\, y)\, z} \mapsto \lambda z.\, z
\end{aligned}
$$

---

[1] After all, the notion of judgment that we learned in Chapter 2 is not really useless!

In the course of reducing an expression to a value, therefore, we may be able to apply the β-reduction in many different ways. As we do not want to apply the β-reduction in an arbitrary way, we need a certain *reduction strategy* so as to apply the β-reduction in a systematic way.

In this course, we consider two reduction strategies: *call-by-name* and *call-by-value*. The call-by-name strategy always reduces the leftmost and outermost redex. To be specific, given an expression $e_1 \, e_2$, it checks if e_1 is a λ-abstraction $\lambda x. e_1'$. If so, it applies the β-reduction to the whole expression to obtain $[e_2/x]e_1'$. Otherwise it attempts to reduce e_1 using the same reduction strategy without considering e_2; when e_1 later reduces to a value (which must be a λ-abstraction), it applies the β-reduction to the whole expression. Consequently the second subexpression in an application (*e.g.*, e_2 in $e_1 \, e_2$) is never reduced. The call-by-value strategy is similar to the call-by-name strategy, but it reduces the second subexpression in an application to a value v after reducing the first subexpression. Hence the call-by-value strategy applies the β-reduction to an application of the form $(\lambda x. e) \, v$ only. Note that neither strategy reduces expressions inside a λ-abstraction, which implies that values are not further reduced.

As an example, let us consider an expression $(\mathsf{id}_1 \, \mathsf{id}_2) \, (\mathsf{id}_3 \, (\lambda z. \mathsf{id}_4 \, z))$ which reduces in different ways under the two reduction strategies; id_i is an abbreviation of an identity function $\lambda x_i. x_i$:

call-by-name	call-by-value
$\underline{(\mathsf{id}_1 \, \mathsf{id}_2)} \, (\mathsf{id}_3 \, (\lambda z. \mathsf{id}_4 \, z))$	$\underline{(\mathsf{id}_1 \, \mathsf{id}_2)} \, (\mathsf{id}_3 \, (\lambda z. \mathsf{id}_4 \, z))$
$\mapsto \underline{\mathsf{id}_2 \, (\mathsf{id}_3 \, (\lambda z. \mathsf{id}_4 \, z))}$	$\mapsto \mathsf{id}_2 \, \underline{(\mathsf{id}_3 \, (\lambda z. \mathsf{id}_4 \, z))}$
$\mapsto \underline{\mathsf{id}_3 \, (\lambda z. \mathsf{id}_4 \, z)}$	$\mapsto \underline{\mathsf{id}_2 \, (\lambda z. \mathsf{id}_4 \, z)}$
$\mapsto \lambda z. \mathsf{id}_4 \, z$	$\mapsto (\lambda z. \mathsf{id}_4 \, z)$

The reduction diverges in the second step: the call-by-name strategy applies the β-reduction to the whole expression because it does not need to inspect the second subexpression $\mathsf{id}_3 \, (\lambda z. \mathsf{id}_4 \, z)$ whereas the call-by-value strategy chooses to reduce the second subexpression which is not a value yet.

Now we are ready to provide inference rules for the judgment $e \mapsto e'$, which we refer to as *reduction rules*. The call-by-name strategy uses two reduction rules (*Lam* for *Lam*bda and *App* for *App*lication):

$$\frac{e_1 \mapsto e_1'}{e_1 \, e_2 \mapsto e_1' \, e_2} \; Lam \qquad \frac{}{(\lambda x. e) \, e' \mapsto [e'/x]e} \; App$$

The call-by-value strategy uses an additional rule to reduce second subexpression in applications; we reuse the reduction rule names from the call-by-name strategy (*Arg* for *Arg*ument):

$$\frac{e_1 \mapsto e_1'}{e_1 \, e_2 \mapsto e_1' \, e_2} \; Lam \qquad \frac{e_2 \mapsto e_2'}{(\lambda x. e) \, e_2 \mapsto (\lambda x. e) \, e_2'} \; Arg \qquad \frac{}{(\lambda x. e) \, v \mapsto [v/x]e} \; App$$

A drawback of the call-by-name strategy is that the same expression may be evaluated multiple times. For example, $(\lambda x. x \, x) \, ((\lambda y. y) \, (\lambda z. z))$ evaluates $(\lambda y. y) \, (\lambda z. z)$ to $\lambda z. z$ eventually twice:

$$\underline{(\lambda x. x \, x) \, ((\lambda y. y) \, (\lambda z. z))}$$
$$\mapsto \underline{((\lambda y. y) \, (\lambda z. z)) \, ((\lambda y. y) \, (\lambda z. z))}$$
$$\mapsto \underline{(\lambda z. z) \, ((\lambda y. y) \, (\lambda z. z))}$$
$$\mapsto \underline{(\lambda y. y) \, (\lambda z. z)}$$
$$\mapsto \lambda z. z$$

In the case of the call-by-value strategy, $(\lambda y. y) \, (\lambda z. z)$ is evaluated only once:

$$(\lambda x. x \, x) \, \underline{((\lambda y. y) \, (\lambda z. z))}$$
$$\mapsto \underline{(\lambda x. x \, x) \, (\lambda z. z)}$$
$$\mapsto \underline{(\lambda z. z) \, (\lambda z. z)}$$
$$\mapsto \lambda z. z$$

On the other hand, the call-by-name strategy never evaluates expressions that do not contribute to evaluations. For example,

$$(\lambda t.\, \lambda f.\, f)\, ((\lambda y.\, y)\, (\lambda z.\, z))\, ((\lambda y'.\, y')\, (\lambda z'.\, z'))$$

does not evaluate $(\lambda y.\, y)\, (\lambda z.\, z)$ at all because it is not used in the evaluation:

$$
\begin{aligned}
&\underline{(\lambda t.\, \lambda f.\, f)\, ((\lambda y.\, y)\, (\lambda z.\, z))}\, ((\lambda y'.\, y')\, (\lambda z'.\, z')) \\
\mapsto\ &\underline{(\lambda f.\, f)\, ((\lambda y'.\, y')\, (\lambda z'.\, z'))} \\
\mapsto\ &\cdots
\end{aligned}
$$

The call-by-value strategy evaluates $(\lambda y.\, y)\, (\lambda z.\, z)$, but the result $\lambda z.\, z$ is ignored in the next reduction:

$$
\begin{aligned}
&(\lambda t.\, \lambda f.\, f)\, (\underline{(\lambda y.\, y)\, (\lambda z.\, z)})\, ((\lambda y'.\, y')\, (\lambda z'.\, z')) \\
\mapsto\ &\underline{(\lambda t.\, \lambda f.\, f)\, (\lambda z.\, z)}\, ((\lambda y'.\, y'))\, (\lambda z'.\, z') \\
\mapsto\ &\underline{(\lambda f.\, f)\, ((\lambda y'.\, y'))}\, (\lambda z'.\, z') \\
\mapsto\ &\cdots
\end{aligned}
$$

The call-by-name strategy is adopted by the functional language Haskell. Haskell is called a *lazy* or *non-strict* functional language because it evaluates arguments to functions only if necessary (*i.e.,* "lazily"). The actual implementation of Haskell uses another reduction strategy called *call-by-need*, which is semantically equivalent to the call-by-name strategy but never evaluates the same expression more than once. The call-by-value strategy is adopted by SML which is called an *eager* or *strict* functional language because it always evaluates arguments to functions regardless of whether they are actually used in function bodies or not (*i.e.,* "eagerly").

We say that an expression is in *normal form* if no reduction rule is applicable. Clearly every value (which is a λ-abstraction in the case of the λ-calculus) is in normal form. There are, however, expressions in normal form that are not values. For example, $x\, \lambda y.\, y$ is in normal form because x cannot be further reduced, but it is not a value either. We say that such expression is *stuck* or its reduction gets *stuck*. A stuck expression may be thought of as an ill-formed program, and ideally should not arise during an evaluation. Chapter 4 presents an extension of the λ-calculus which statically (*i.e.,* at compile time) guarantees that a program satisfying a certain criterion never gets stuck.

3.3 Substitution

This section presents a definition of substitution $[e'/x]e$ to complete the operational semantics of the λ-calculus. While an informal interpretation of $[e'/x]e$ is obvious, its formal definition is a lot trickier than it appears.

First we need the notion of *free* variable which is the opposite of the notion of bound variable and plays a key role in the definition of substitution. A free variable is a variable that is not bound in any enclosing λ-abstraction. For example, y in $\lambda x.\, y$ is a free variable because no λ-abstraction of the form $\lambda y.\, e$ encloses its occurrence. To formalize the notion of free variable, we introduce a mapping $FV(e)$ to mean the set of free variables in e:

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.\, e) &= FV(e) - \{x\} \\
FV(e_1\, e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

Since a variable is either free or bound, a variable x in e such that $x \notin FV(e)$ must be bound in some λ-abstraction. We say that an expression e is *closed* if it contains no free variables, *i.e.,* $FV(e) = \varnothing$. Here are a few examples:

$$
\begin{aligned}
FV(\lambda x.\, x) &= \{\} \\
FV(x\, y) &= \{x, y\} \\
FV(\lambda x.\, x\, y) &= \{y\} \\
FV(\lambda y.\, \lambda x.\, x\, y) &= \{\} \\
FV((\lambda x.\, x\, y)\, (\lambda x.\, x\, z)) &= \{y, z\}
\end{aligned}
$$

A substitution $[e/x]e'$ is defined inductively with the following cases:

$$
\begin{aligned}
[e/x]x &= e \\
[e/x]y &= y && \text{if } x \neq y \\
[e/x](e_1\ e_2) &= [e/x]e_1\ [e/x]e_2
\end{aligned}
$$

In order to give the definition of the remaining case $[e/x]\lambda y.\ e'$, we need to understand two properties of variables. The first property is that the name of a bound variable does not matter, which also conforms to our intuition. For example, an identity function $\lambda x.\ x$ inside an expression e may be rewritten as $\lambda y.\ y$ for an arbitrary variable y without changing the intended meaning of e, since both $\lambda x.\ x$ and $\lambda y.\ y$ denote an identity function. Another example is to rewrite $\lambda x.\ \lambda y.\ x\ y$ as $\lambda y.\ \lambda x.\ y\ x$ where both expressions denote the same function that applies the first argument to the second argument.

Formally we use a judgment $e \equiv_\alpha e'$ to mean that e can be rewritten as e' by renaming bound variables in e and vice versa. Here are examples of $e \equiv_\alpha e'$:

$$
\begin{aligned}
\lambda x.\ x &\equiv_\alpha \lambda y.\ y \\
\lambda x.\ \lambda y.\ x\ y &\equiv_\alpha \lambda z.\ \lambda y.\ z\ y \\
\lambda x.\ \lambda y.\ x\ y &\equiv_\alpha \lambda x.\ \lambda z.\ x\ z \\
\lambda x.\ \lambda y.\ x\ y &\equiv_\alpha \lambda y.\ \lambda x.\ y\ x
\end{aligned}
$$

By a historical accident, $\equiv_\alpha$ is called the *α-equivalence relation*, or we say that an *α-conversion* of e into e' rewrites e as e' by renaming bound variables in e. It turns out that a definition of $e \equiv_\alpha e'$ is also tricky to develop, which is given at the end of the present section.

The first property justifies the following case of substitution:

$$
[e'/x]\lambda x.\ e = \lambda x.\ e
$$

Intuitively, if we rewrite $\lambda x.\ e$ as another λ-abstraction of the form $\lambda y.\ e''$ where y is a fresh variable such that $x \neq y$, the substitution $[e'/x]$ is effectively ignored because x is found nowhere in $\lambda y.\ e''$. Here is a simple example with $e = x$:

$$
\begin{aligned}
[e'/x]\lambda x.\ x &\equiv_\alpha [e'/x]\lambda y.\ y \\
&= \lambda y.\ y \\
&\equiv_\alpha \lambda x.\ x
\end{aligned}
$$

A generalization of the case is that $[e'/x]$ has no effect on e if x is not a free variable in e:

$$
[e'/x]e = e \qquad \text{if } x \notin FV(e)
$$

That is, we want to apply $[e'/x]$ to e only if x is a free variable in e.

The second property is that a free variable x in an expression e never turns into a bound variable; when explicitly replaced by another expression e', as in $[e'/x]e$, it simply disappears. To better understand the second property, let us consider a naive definition of $[e'/x]\lambda y.\ e$ where y may or may not be a free variable in e':

$$
[e'/x]\lambda y.\ e = \lambda y.\ [e'/x]e \qquad \text{if } x \neq y
$$

Now, if y is a free variable in e', it automatically becomes a bound variable in $\lambda y.\ [e'/x]e$, which is not acceptable. Here is an example showing such an anomaly:

$$
(\lambda x.\ \lambda y.\ x)\ y \mapsto [y/x]\lambda y.\ x = \lambda y.\ [y/x]x = \lambda y.\ y
$$

Before the substitution, $\lambda y.\ x$ is a λ-abstraction that ignores its argument and returns x, but after the substitution, it turns into an identity function! What happens in the example is that a free variable y to be substituted for x is supposed to remain free after the substitution, but is accidentally *captured* by the λ-abstraction $\lambda y.\ x$ and becomes a bound variable. Such a phenomenon is called a *variable capture* which destroys the intuition that a free variable remains free unless it is replaced by another expression. This observation is generalized in the following definition of $[e'/x]\lambda y.\ e$ which is called a *capture-avoiding substitution*:

$$
[e'/x]\lambda y.\ e = \lambda y.\ [e'/x]e \qquad \text{if } x \neq y, y \notin FV(e')
$$

If a variable capture occurs because $y \in FV(e')$, we rename y to another variable that is not free in e. For example, $(\lambda x. \lambda y. x\, y)\, y$ can be safely reduced after renaming the bound variable y to a fresh variable z:

$$(\lambda x. \lambda y. x\, y)\, y \mapsto [y/x]\lambda y. x\, y \equiv_\alpha [y/x]\lambda z. x\, z = \lambda z. y\, z$$

In the literature, the unqualified term "substitution" universally means a capture-avoiding substitution which renames bound variables as necessary.

Now we give a definition of the judgment $e \equiv_\alpha e'$. We need the notion of *variable swapping* $[x \leftrightarrow y]e$ which is obtained by replacing *all* occurrences of x in e by y and *all* occurrences of y in e by x. We emphasize that "all" occurrences include even those next to λ in λ-abstractions, which makes it straightforward to implement $[x \leftrightarrow y]e$. Here is an example:

$$[x \leftrightarrow y]\lambda x. \lambda y. x\, y \;\; = \;\; \lambda y. [x \leftrightarrow y]\lambda y. x\, y \;\; = \;\; \lambda y. \lambda x. [x \leftrightarrow y](x\, y) \;\; = \;\; \lambda y. \lambda x. y\, x$$

The definition of $e \equiv_\alpha e'$ is given inductively by the following inference rules:

$$\frac{}{x \equiv_\alpha x} \; Var_\alpha \qquad \frac{e_1 \equiv_\alpha e_1' \quad e_2 \equiv_\alpha e_2'}{e_1\, e_2 \equiv_\alpha e_1'\, e_2'} \; App_\alpha$$

$$\frac{e \equiv_\alpha e'}{\lambda x. e \equiv_\alpha \lambda x. e'} \; Lam_\alpha \qquad \frac{x \neq y \quad y \notin FV(e) \quad [x \leftrightarrow y]e \equiv_\alpha e'}{\lambda x. e \equiv_\alpha \lambda y. e'} \; Lam_\alpha'$$

The rule Lam_α says that to compare $\lambda x. e$ and $\lambda x. e'$ which bind the same variable, we compare their bodies e and e'. To compare two λ-abstractions binding different variables, we use the rule Lam_α'.

To see why the rule Lam_α' works, we need to understand the implication of the premise $y \notin FV(e)$. Since $y \notin FV(e)$ implies $y \notin FV(\lambda x. e)$ and we have $x \notin FV(\lambda x. e)$, an outside observer would notice no difference even if the two variables x and y were literally swapped in $\lambda x. e$. In other words, $\lambda x. e$ and $[x \leftrightarrow y]\lambda x. e$ are effectively the same from the point of view of an outside observer. Since $[x \leftrightarrow y]\lambda x. e = \lambda y. [x \leftrightarrow y]e$, we compare $[x \leftrightarrow y]e$ with e', which is precisely the third premise in the rule Lam_α'. As an example, here is a proof of $\lambda x. \lambda y. x\, y \equiv_\alpha \lambda y. \lambda x. y\, x$:

$$\frac{x \neq y \quad y \notin FV(\lambda y. x\, y) \quad \dfrac{\dfrac{\dfrac{}{y \equiv_\alpha y} Var_\alpha \quad \dfrac{}{x \equiv_\alpha x} Var_\alpha}{y\, x \equiv_\alpha y\, x} App_\alpha}{\lambda x. y\, x \equiv_\alpha \lambda x. y\, x} Lam_\alpha}{\lambda x. \lambda y. x\, y \equiv_\alpha \lambda y. \lambda x. y\, x} \; Lam_\alpha'$$

Exercise 3.1. Can we prove $\lambda x. e \equiv_\alpha \lambda y. e'$ when $x \neq y$ and $y \in FV(e)$?

Exercise 3.2. Suppose $x \notin FV(e)$ and $y \notin FV(e)$. Prove $e \equiv_\alpha [x \leftrightarrow y]e$.

Finally we give a complete definition of substitution:

$$
\begin{array}{rcll}
[e/x]x & = & e & \\
[e/x]y & = & y & \text{if } x \neq y \\
[e/x](e_1\, e_2) & = & [e/x]e_1\, [e/x]e_2 & \\
[e'/x]\lambda x. e & = & \lambda x. e & \\
[e'/x]\lambda y. e & = & \lambda y. [e'/x]e & \text{if } x \neq y, y \notin FV(e') \\
[e'/x]\lambda y. e & = & \lambda z. [e'/x][y \leftrightarrow z]e & \text{if } x \neq y, y \in FV(e') \\
& & & \quad \text{where } z \neq y, z \notin FV(e), z \neq x, z \notin FV(e')
\end{array}
$$

The last equation implies that if y is a free variable in e', we choose another variable z satisfying the *where* clause and rewrite $\lambda y. e$ as $\lambda z. [y \leftrightarrow z]e$ by α-conversion:

$$\frac{y \neq z \quad z \notin FV(e) \quad [y \leftrightarrow z]e \equiv_\alpha [y \leftrightarrow z]e}{\lambda y. e \equiv_\alpha \lambda z. [y \leftrightarrow z]e} \; Lam_\alpha'$$

Then $z \notin FV(e')$ allows us to rewrite $[e'/x]\lambda z.\,[y \leftrightarrow z]e$ as $\lambda z.\,[e'/x][y \leftrightarrow z]e$. In a typical implementation, we obtain such a variable z just by generating a fresh variable. In such a case, replacing z by y never occurs and thus the last equation can be written as follows:

$$[e'/x]\lambda y.\,e \quad = \quad \lambda z.\,[e'/x][z/y]e$$

The new equation is less efficient, however. Consider $e = x\,y\,(\lambda y.\,x\,y)$ for example. $[y \leftrightarrow z]e$ gives $x\,z\,(\lambda z.\,x\,z)$ and $[e'/x][y \leftrightarrow z]e$ encounters no variable capture:

$$[e'/x][y \leftrightarrow z]e \;=\; [e'/x](x\,z\,(\lambda z.\,x\,z)) \;=\; e'\,z\,(\lambda z.\,e'\,z)$$

In contrast, $[z/y]e$ gives $x\,z\,(\lambda y.\,x\,y)$ and $[e'/x][z/y]e$ again encounters a variable capture in $[e'/x]\lambda y.\,x\,y$:

$$[e'/x][z/y]e \;=\; [e'/x](x\,z\,(\lambda y.\,x\,y)) \;=\; e'\,z\,[e'/x](\lambda y.\,x\,y)$$

So we have to generate another fresh variable!

Exercise 3.3. What is the result of α-converting each expression in the left where a fresh variable to be generated in the conversion is provided in the right? Which expression is impossible to α-convert?

$$
\begin{array}{rcl}
\lambda x.\,\lambda x'.\,x\,x' & \equiv_\alpha & \lambda x'.\,\underline{\hspace{4cm}} \\
\lambda x.\,\lambda x'.\,x\,x'\,x'' & \equiv_\alpha & \lambda x'.\,\underline{\hspace{4cm}} \\
\lambda x.\,\lambda x'.\,x\,x'\,x'' & \equiv_\alpha & \lambda x''.\,\underline{\hspace{4cm}}
\end{array}
$$

3.4 Programming in the λ-calculus

In order to develop the λ-calculus to a full-fledged functional language, we need to show how to encode common datatypes such as boolean values, integers, and lists in the λ-calculus. Since all values in the λ-calculus are λ-abstractions, all such datatypes are also encoded with λ-abstractions. Once we show how to encode specific datatypes, we may use them as if they were built-in datatypes.

3.4.1 Church booleans

The inherent capability of a boolean value is to choose one of two different options. For example, a boolean truth chooses the first of two different options, as in an SML expression `if true then` e_1 `else` e_2. Thus boolean values in the λ-calculus, called Church booleans, are written as follows:

$$
\begin{array}{rcl}
\mathsf{tt} & = & \lambda t.\,\lambda f.\,t \\
\mathsf{ff} & = & \lambda t.\,\lambda f.\,f
\end{array}
$$

Then a conditional construct `if` e `then` e_1 `else` e_2 is defined as follows:

$$\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \quad = \quad e\,e_1\,e_2$$

Here are examples of reducing conditional constructs under the call-by-name strategy:

$$
\begin{array}{l}
\mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 = \mathsf{tt}\,e_1\,e_2 = \underline{(\lambda t.\,\lambda f.\,t)\,e_1}\,e_2 \mapsto \underline{(\lambda f.\,e_1)\,e_2} \mapsto e_1 \\
\mathsf{if}\ \mathsf{ff}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 = \mathsf{ff}\,e_1\,e_2 = \underline{(\lambda t.\,\lambda f.\,f)\,e_1}\,e_2 \mapsto \underline{(\lambda f.\,f)\,e_2} \mapsto e_2
\end{array}
$$

Logical operators on boolean values are defined as follows:

$$
\begin{array}{rcl}
\mathsf{and} & = & \lambda x.\,\lambda y.\,x\,y\,\mathsf{ff} \\
\mathsf{or} & = & \lambda x.\,\lambda y.\,x\,\mathsf{tt}\,y \\
\mathsf{not} & = & \lambda x.\,x\,\mathsf{ff}\,\mathsf{tt}
\end{array}
$$

As an example, here are sequences of reductions of $\mathsf{and}\ e_1\ e_2$ when $e_1 \mapsto^* \mathsf{tt}$ and $e_1 \mapsto^* \mathsf{ff}$, respectively, under the call-by-name strategy:

$$
\begin{array}{rl \qquad rl}
 & \mathsf{and}\ e_1\ e_2 & & \mathsf{and}\ e_1\ e_2 \\
\mapsto^* & e_1\,e_2\,\mathsf{ff} & \mapsto^* & e_1\,e_2\,\mathsf{ff} \\
\mapsto^* & \mathsf{tt}\,e_2\,\mathsf{ff} & \mapsto^* & \mathsf{ff}\,e_2\,\mathsf{ff} \\
\mapsto^* & e_2 & \mapsto^* & \mathsf{ff}
\end{array}
$$

The left sequence shows that when $e_1 \mapsto^* \mathsf{tt}$ holds, and $e_1\,e_2$ denotes the same truth value as e_2. The right sequence shows that when $e_1 \mapsto^* \mathsf{ff}$ holds, and $e_1\,e_2$ evaluates to ff regardless of e_2.

Exercise 3.4. Consider the conditional construct if e then e_1 else e_2 defined as $e\ e_1\ e_2$ under the call-by-value strategy. How is it different from the conditional construct in SML?

Exercise 3.5. Define the logical operator xor. An easy way to define it is to use a conditional construct and the logical operator not.

3.4.2 Pairs

The inherent capability of a pair is to carry two unrelated values and to retrieve either value when requested. Thus, in order to represent a pair of e_1 and e_2, we build a λ-abstraction which returns e_1 and e_2 when applied to tt and ff, respectively. Projection operators treat a pair as a λ-abstraction and applies it to either tt or ff.

$$
\begin{aligned}
\text{pair} &= \lambda x.\,\lambda y.\,\lambda b.\,b\ x\ y \\
\text{fst} &= \lambda p.\,p\ \text{tt} \\
\text{snd} &= \lambda p.\,p\ \text{ff}
\end{aligned}
$$

As an example, let us reduce fst (pair $e_1\ e_2$) under the call-by-name strategy. Note that pair $e_1\ e_2$ evaluates to $\lambda b.\,b\ e_1\ e_2$ which expects a boolean value for b in order to select either e_1 or e_2. If tt is substituted for b, then $b\ e_1\ e_2$ reduces to e_1.

$$
\begin{aligned}
\text{fst (pair } e_1\ e_2) &\mapsto & (\text{pair } e_1\ e_2)\ \text{tt} \\
&\mapsto^* & (\lambda b.\,b\ e_1\ e_2)\ \text{tt} \\
&\mapsto & \text{tt}\ e_1\ e_2 \\
&\mapsto^* & e_1
\end{aligned}
$$

3.4.3 Church numerals

The inherent capability of a natural number n is to repeat a given process n times. In the case of the λ-calculus, n is encoded as a λ-abstraction $\hat{n}$, called a Church numeral, that takes a function f and returns $f^n = f \circ f \cdots \circ f$ (n times). Note that f^0 is an identity function $\lambda x.\,x$ because f is applied 0 times to its argument x, and that $\hat{1}$ itself is an identity function $\lambda f.\,f$.

$$
\begin{aligned}
\hat{0} &= \lambda f.\,f^0 &=& \lambda f.\,\lambda x.\,x \\
\hat{1} &= \lambda f.\,f^1 &=& \lambda f.\,\lambda x.\,f\ x \\
\hat{2} &= \lambda f.\,f^2 &=& \lambda f.\,\lambda x.\,f\ (f\ x) \\
\hat{3} &= \lambda f.\,f^3 &=& \lambda f.\,\lambda x.\,f\ (f\ (f\ x)) \\
&\cdots \\
\hat{n} &= \lambda f.\,f^n &=& \lambda f.\,\lambda x.\,f\ (f\ (f\ \cdots (f\ x)\cdots))
\end{aligned}
$$

If we read f as S and x as O, $\hat{n}\ f\ x$ returns the representation of the natural number n shown in Chapter 2.

Now let us define arithmetic operations on natural numbers. The addition operation add $\hat{m}\ \hat{n}$ returns $\widehat{m+n}$ which is a λ-abstraction taking a function f and returning f^{m+n}. Since f^{m+n} may be written as $\lambda x.\,f^{m+n}\ x$, we develop add as follows; in order to differentiate natural numbers (*e.g.*, n) from their encoded form (*e.g.*, $\hat{n}$), we use $\hat{m}$ and $\hat{n}$ as variables:

$$
\begin{aligned}
\text{add} &= \lambda \hat{m}.\,\lambda \hat{n}.\,\lambda f.\,f^{m+n} \\
&= \lambda \hat{m}.\,\lambda \hat{n}.\,\lambda f.\,\lambda x.\,f^{m+n}\ x \\
&= \lambda \hat{m}.\,\lambda \hat{n}.\,\lambda f.\,\lambda x.\,f^m\ (f^n\ x) \\
&= \lambda \hat{m}.\,\lambda \hat{n}.\,\lambda f.\,\lambda x.\,\hat{m}\ f\ (\hat{n}\ f\ x)
\end{aligned}
$$

Note that f^m is obtained as $\hat{m}\ f$ (and similarly for f^n).

Exercise 3.6. Define the multiplication operation mult $\hat{m}\ \hat{n}$ which returns $\widehat{m*n}$.

The multiplication operation can be defined in two ways. An easy way is to exploit the equation $m*n = m+m+\cdots+m$ (n times). That is, $m*n$ is obtained by adding m to zero exactly n times. Since add $\hat{m}$ is conceptually a function adding m to its argument, we apply add $\hat{m}$ to $\hat{0}$ exactly n times to obtain $\widehat{m*n}$, or equivalently apply $(\text{add } \hat{m})^n$ to $\hat{0}$:

$$
\begin{aligned}
\text{mult} &= \lambda \hat{m}.\,\lambda \hat{n}.\,(\text{add } \hat{m})^n\ \hat{0} \\
&= \lambda \hat{m}.\,\lambda \hat{n}.\,\hat{n}\ (\text{add } \hat{m})\ \hat{0}
\end{aligned}
$$

An alternative way (which may in fact be easier to figure out than the first solution) is to exploit the equation $f^{m*n} = (f^m)^n = (\hat{m}\ f)^n = \hat{n}\ (\hat{m}\ f)$:

$$\mathsf{mult} \quad = \quad \lambda\hat{m}.\,\lambda\hat{n}.\,\lambda f.\,\hat{n}\ (\hat{m}\ f)$$

The subtraction operation is more difficult to define than the previous two operations. Suppose that we have a predecessor function pred computing the predecessor of a given natural number: pred $\hat{n}$ returns $\widehat{n-1}$ if $n > 0$ and $\hat{0}$ otherwise. To define the subtraction operation sub $\hat{m}\ \hat{n}$ which returns $\widehat{m-n}$ if $m > n$ and $\hat{0}$ otherwise, we apply pred to $\hat{m}$ exactly n times:

$$\begin{aligned}
\mathsf{sub} \quad &= \quad \lambda\hat{m}.\,\lambda\hat{n}.\,\mathsf{pred}^n\ \hat{m} \\
&= \quad \lambda\hat{m}.\,\lambda\hat{n}.\,(\hat{n}\ \mathsf{pred})\ \hat{m}
\end{aligned}$$

Exercise 3.7. Define the predecessor function pred. Use an idea similar to the one used in a tail-recursive implementation of the Fibonacci function.

The predecessor function pred uses an auxiliary function next which takes pair $\hat{k}\ \hat{m}$, ignores $\hat{k}$, and returns pair $\hat{m}\ \widehat{m+1}$:

$$\mathsf{next} \quad = \quad \lambda p.\,\mathsf{pair}\ (\mathsf{snd}\ p)\ (\mathsf{add}\ (\mathsf{snd}\ p)\ \hat{1})$$

It can be shown that by applying next to pair $\hat{0}\ \hat{0}$ exactly n times, we obtain pair $\widehat{n-1}\ \hat{n}$ if $n > 0$ (under a certain reduction strategy):

$$\begin{aligned}
\mathsf{next}^0\ (\mathsf{pair}\ \hat{0}\ \hat{0}) \quad &\mapsto^* \quad \mathsf{pair}\ \hat{0}\ \hat{0} \\
\mathsf{next}^1\ (\mathsf{pair}\ \hat{0}\ \hat{0}) \quad &\mapsto^* \quad \mathsf{pair}\ \hat{0}\ \hat{1} \\
\mathsf{next}^2\ (\mathsf{pair}\ \hat{0}\ \hat{0}) \quad &\mapsto^* \quad \mathsf{pair}\ \hat{1}\ \hat{2} \\
&\vdots \\
\mathsf{next}^n\ (\mathsf{pair}\ \hat{0}\ \hat{0}) \quad &\mapsto^* \quad \mathsf{pair}\ \widehat{n-1}\ \hat{n}
\end{aligned}$$

Since the predecessor of 0 is 0 anyway, the first component of $\mathsf{next}^n\ (\mathsf{pair}\ \hat{0}\ \hat{0})$ encodes the predecessor of n. Thus pred is defined as follows:

$$\begin{aligned}
\mathsf{pred} \quad &= \quad \lambda\hat{n}.\,\mathsf{fst}\ (\mathsf{next}^n\ (\mathsf{pair}\ \hat{0}\ \hat{0})) \\
&= \quad \lambda\hat{n}.\,\mathsf{fst}\ (\hat{n}\ \mathsf{next}\ (\mathsf{pair}\ \hat{0}\ \hat{0}))
\end{aligned}$$

Exercise 3.8. Define a function isZero $= \lambda\hat{n}.\cdots$ which tests if a given Church numeral is $\hat{0}$. Use it to define another function eq $= \lambda\hat{m}.\,\lambda\hat{n}.\cdots$ which tests if two given Church numerals are equal.

3.5 Fixed point combinator

Since the λ-calculus is equally powerful as Turing machines, every Turing machine can be simulated by a certain expression in the λ-calculus. In particular, there are expressions in the λ-calculus that correspond to Turing machines that do not terminate and Turing machines that compute *recursive functions.*

It is relatively easy to find an expression whose reduction does not terminate. Suppose that we wish to find an expression omega such that omega $\mapsto$ omega. Since it reduces to the same expression, its reduction never terminates. We rewrite omega as $(\lambda x.\,e)\ e'$ so that the β-reduction can be applied to the whole expression omega. Then we have

$$\mathsf{omega} = (\lambda x.\,e)\ e' \mapsto [e'/x]e = \mathsf{omega}.$$

Now omega $= [e'/x]e = (\lambda x.\,e)\ e'$ suggests $e = e''\ x$ for some expression e'' such that $[e'/x]e'' = \lambda x.\,e$ (and $[e'/x]x = e'$):

$$\begin{aligned}
\mathsf{omega} \quad &= \quad [e'/x]e \\
&= \quad [e'/x](e''\ x) && \text{from } e = e''\ x \\
&= \quad [e'/x]e''\ [e'/x]x \\
&= \quad [e'/x]e''\ e' \\
&= \quad (\lambda x.\,e)\ e' && \text{from } [e'/x]e'' = \lambda x.\,e
\end{aligned}$$

From $e = e'' \, x$ and $[e'/x]e'' = \lambda x. \, e$, we obtain $[e'/x]e'' = \lambda x. \, e'' \, x$. By letting $e'' = x$ in $[e'/x]e'' = \lambda x. \, e'' \, x$, we obtain $e' = \lambda x. \, x \, x$. Then omega can be defined as follows:

$$
\begin{aligned}
\text{omega} \quad &= \quad (\lambda x. \, e) \, e' \\
&= \quad (\lambda x. \, e'' \, x) \, e' \qquad &&\text{from } e = e'' \, x \\
&= \quad (\lambda x. \, x \, x) \, e' \qquad &&\text{from } e'' = x \\
&= \quad (\lambda x. \, x \, x) \, (\lambda x. \, x \, x) \qquad &&\text{from } e' = \lambda x. \, x \, x
\end{aligned}
$$

Now it can be shown that the reduction of omega defined as above never terminates.

Then how do we write recursive functions in the λ-calculus? We begin by assuming a *recursive function construct* fun $f \, x. \, e$ which defines a recursive function f whose argument is x and whose body is e. Note that the body e may contain references to f. Our goal is to show that fun $f \, x. \, e$ is *syntactic sugar* (which dissolves in the λ-calculus) in the sense that it can be rewritten as an existing expression in the λ-calculus and thus its addition does not increase the expressive power of the λ-calculus.

As a working example, we use a factorial function fac:

$$
\text{fac} \quad = \quad \text{fun } f \, n. \, \text{if eq } n \, \hat{0} \text{ then } \hat{1} \text{ else mult } n \, (f \, (\text{pred } n))
$$

Semantically f in the body refers to the very function fac being defined. First we mechanically derive a λ-abstraction $\text{FAC} = \lambda f. \, \lambda n. \, e$ from fac $= $ fun $f \, n. \, e$:

$$
\text{FAC} \quad = \quad \lambda f. \, \lambda n. \, \text{if eq } n \, \hat{0} \text{ then } \hat{1} \text{ else mult } n \, (f \, (\text{pred } n))
$$

Note that FAC has totally different characteristics than fac: while fac takes a natural number n to return another natural number, FAC takes a function f to return another function. (If fac and FAC were allowed to have types, fac would have type nat$\rightarrow$nat whereas FAC would have type (nat$\rightarrow$nat)$\rightarrow$(nat$\rightarrow$nat).)

The key idea behind constructing FAC is that given a *partial* implementation f of the factorial function, FAC f returns an *improved* implementation of the factorial function. Suppose that f correctly computes the factorial of any natural number up to n. Then FAC f correctly computes the factorial of any natural number up to $n + 1$, which is an improvement over f. Note also that FAC f correctly computes the factorial of 0 regardless of f. In particular, even when given a least informative function $f = \lambda n. \, \text{omega}$ (which does nothing because it never returns), FAC f correctly computes the factorial of 0. Thus we can imagine an infinite chain of functions $\{\text{fac}_0, \text{fac}_1, \cdots, \text{fac}_i, \cdots\}$ which begins with $\text{fac}_0 = \text{FAC } \lambda n. \, \text{omega}$ and repeatedly applies the equation $\text{fac}_{i+1} = \text{FAC } \text{fac}_i$:

$$
\begin{aligned}
\text{fac}_0 \quad &= \quad && \text{FAC } \lambda n. \, \text{omega} \\
\text{fac}_1 \quad &= \quad \text{FAC } \text{fac}_0 \quad &&= \quad \text{FAC}^2 \, \lambda n. \, \text{omega} \\
\text{fac}_2 \quad &= \quad \text{FAC } \text{fac}_1 \quad &&= \quad \text{FAC}^3 \, \lambda n. \, \text{omega} \\
&\quad \vdots && \\
\text{fac}_i \quad &= \quad \text{FAC } \text{fac}_{i-1} \quad &&= \quad \text{FAC}^{i+1} \, \lambda n. \, \text{omega} \\
&\quad \vdots &&
\end{aligned}
$$

Note that fac_i correctly computes the factorial of any natural number up to i. Then, if ω denotes an infinite natural number (greater than any natural number), we may take fac_ω as a correct implementation of the factorial function fac, *i.e.*, fac $= \text{fac}_\omega$.

Another important observation is that given a correct implementation fac of the factorial function, FAC fac returns another correct implementation of the factorial function, That is, if fac is a correct implementation of the factorial function,

$$
\lambda n. \, \text{if eq } n \, \hat{0} \text{ then } \hat{1} \text{ else mult } n \, (\text{fac } (\text{pred } n))
$$

is also a correct implementation of the factorial function. Since the two functions are essentially identical in that both return the same result for any argument, we may let fac $= $ FAC fac. If we substitute fac_ω for fac in the equation, we obtain $\text{fac}_\omega = \text{fac}_{\omega+1}$ which also makes sense because $\omega \leq \omega + 1$ by the definition of $+$ and $\omega + 1 \leq \omega$ by the definition of ω (which is greater than any natural number including $\omega + 1$).

Now it seems that FAC contains all necessary information to derive fac $= \text{fac}_\omega = $ FAC fac, but exactly how? It turns out that fac is obtained by applying the *fixed point combinator* fix to FAC, *i.e.*, fac $= $ fix FAC, where fix is defined as follows:

$$
\text{fix} \quad = \quad \lambda F. \, (\lambda f. \, F \, (\lambda x. \, f \, f \, x)) \, (\lambda f. \, F \, (\lambda x. \, f \, f \, x))
$$

Here we assume the call-by-value strategy; for the call-by-name strategy, we simplify $\lambda x.\, f\, f\, x$ into $f\, f$ and use the following fixed point combinator fix_{CBN}:

$$\text{fix}_{\text{CBN}} \quad = \quad \lambda F.\,(\lambda f.\, F\,(f\, f))\,(\lambda f.\, F\,(f\, f))$$

To understand how the fixed point combinator fix works, we need to learn the concept of *fixed point*. A fixed point of a function f is a value v such that $v = f(v)$. For example, the fixed point of a function $f(x) = 2 - x$ is 1 because $1 = f(1)$. As its name suggests, fix takes a function F (which itself transforms a function f into another function f') and returns its fixed point. That is, fix F is a fixed point of F:

$$\text{fix } F \quad = \quad F\,(\text{fix } F)$$

Informally the left expression transforms into the right expression via the following steps; we use a symbol $\approx$ to emphasize "informally" because the transformation is not completely justified by the β-reduction alone:

$$
\begin{array}{lll}
\text{fix } F & \mapsto & g\, g & \qquad \text{where } g = \lambda f.\, F\,(\lambda x.\, f\, f\, x) \\
& = & (\lambda f.\, F\,(\lambda x.\, f\, f\, x))\, g & \\
& \mapsto & F\,(\lambda x.\, g\, g\, x) & \\
& \approx & F\,(g\, g) & \qquad \text{because } \lambda x.\, g\, g\, x \approx g\, g \\
& \approx & F\,(\text{fix } F) & \qquad \text{because fix } F \mapsto g\, g
\end{array}
$$

Now we can explain why fix FAC gives an implementation of the factorial function. By the nature of the fixed point combinator fix, we have

$$\text{fix FAC} \quad = \quad \text{FAC (fix FAC)}.$$

That is, fix FAC returns a function f satisfying $f = \text{FAC}f$, which is precisely the property that fac needs to satisfy! Therefore we take fix FAC as an equivalent of fac.[2]

An alternative way to explain the behavior of fix FAC is as follows. Suppose that we wish to compute fac n for an arbitrary natural number n. Since fix FAC is a fixed point of FAC, we have the following equation:

$$
\begin{array}{lllll}
\text{fix FAC} & = & & & \text{FAC (fix FAC)} \\
& = & \text{FAC (FAC (fix FAC))} & = & \text{FAC}^2\,(\text{fix FAC}) \\
& = & \text{FAC}^2\,(\text{FAC (fix FAC)}) & = & \text{FAC}^3\,(\text{fix FAC}) \\
& \vdots & & & \\
& = & \text{FAC}^n\,(\text{FAC (fix FAC)}) & = & \text{FAC}^{n+1}\,(\text{fix FAC})
\end{array}
$$

The key observation is that $\text{FAC}^{n+1}\,(\text{fix FAC})$ correctly computes the factorial of any natural number up to n *regardless of what* fix FAC *does* (see Page 45). Since we have fix FAC $= \text{FAC}^{n+1}\,(\text{fix FAC})$, it follows that fix FAC correctly computes the factorial of an *arbitrary* natural number. That is, fix FAC does precisely what fac does.

In summary, in order to encode a recursive function fun $f\, x.\, e$ in the λ-calculus, we first derive a λ-abstraction $F = \lambda f.\, \lambda x.\, e$. Then fix F auto*magically* returns a function that exhibits the same behavior as fun $f\, x.\, e$ does.

Exercise 3.9. Under the call-by-value strategy, fac, or equivalently fix FAC, never terminates when applied to any natural number! Why? (Hint: Exercise 3.4)

3.6 Deriving the fixed point combinator

This section explains how to derive the fixed point combinator. As its formal derivation is extremely intricate, we will illustrate the key idea with an example. Students may choose to skip this section if they wish.

---

[2] The fixed point combinator fix actually yields what is called the *least* fixed point. That is, a function F may have many fixed points and fix returns the least one in the sense that the least one is the most informative one. The least fixed point is what we usually expect.

Let us try to write a factorial function fac without using the fixed point combinator. Consider the following function $\text{fac}_{\text{wrong}}$:

$$\text{fac}_{\text{wrong}} \quad = \quad \lambda n.\, \text{if eq } n \,\hat{0} \text{ then } \hat{1} \text{ else mult } n \,(f \,(\text{pred } n))$$

$\text{fac}_{\text{wrong}}$ is simply wrong because its body contains a reference to an unbound variable f. If, however, f points to a correct implementation fac of the factorial function, $\text{fac}_{\text{wrong}}$ would also be a correct implementation. Since there is no way to use a free variable f in reducing an expression, we have to introduce it in a λ-abstraction anyway:

$$\text{FAC} \quad = \quad \lambda f.\, \lambda n.\, \text{if eq } n \,\hat{0} \text{ then } \hat{1} \text{ else mult } n \,(f \,(\text{pred } n))$$

FAC is definitely an improvement over $\text{fac}_{\text{wrong}}$, but it is not a function taking a natural number; rather it takes a function f to return another function which refines f. More importantly, there seems to be no way to make a recursive call with FAC because FAC calls only its argument f in its body and never makes a recursive call to itself.

Then how do we make a recursive call with FAC? The problem at hand is that the body of FAC, which needs to call fac, calls only its argument f. Our instinct, however, says that FAC contains all necessary information to derive fac (*i.e.*, FAC $\approx$ fac) because its body resembles a typical implementation of the factorial function. Thus we are led to try substituting FAC itself for f. That is, we make a call to FAC using FAC itself as an argument — what a crazy idea it is!:

$$\text{FAC FAC} \quad = \quad \lambda n.\, \text{if eq } n \,\hat{0} \text{ then } \hat{1} \text{ else mult } n \,(\text{FAC }(\text{pred } n))$$

Unfortunately FAC FAC returns a function which does not make sense: in its body, a call to FAC is made with an argument pred n, but FAC expects not a natural number but a function. It is, however, easy to fix the problem: if FAC FAC returns a correct implementation of the factorial function, we only need to replace FAC in the body by FAC FAC. That is, what we want in the end is the following equation

$$\text{FAC FAC} \quad = \quad \lambda n.\, \text{if eq } n \,\hat{0} \text{ then } \hat{1} \text{ else mult } n \,(\text{FAC FAC }(\text{pred } n))$$

where FAC FAC serves as a correct implementation of the factorial function.

Let us change the definition of FAC so that it satisfies the above equation. All we need to do is to replace a reference to f in its body by an application $f\,f$. Thus we obtain a new function Fac defined as follows:

$$\text{Fac} \quad = \quad \lambda f.\, \lambda n.\, \text{if eq } n \,\hat{0} \text{ then } \hat{1} \text{ else mult } n \,(f\,f \,(\text{pred } n))$$

It is easy to see that Fac satisfies the following equation:

$$\text{Fac Fac} \quad = \quad \lambda n.\, \text{if eq } n \,\hat{0} \text{ then } \hat{1} \text{ else mult } n \,(\text{Fac Fac }(\text{pred } n))$$

Since Fac Fac returns a correct implementation of the factorial function, we define fac as follows:

$$\text{fac} \quad = \quad \text{Fac Fac}$$

Now let us derive the fixed point combinator fix by rewriting fac in terms of fix (and FAC as it turns out). Consider the body of Fac:

$$\text{Fac} \quad = \quad \lambda f.\, \lambda n.\, \underline{\text{if eq } n \,\hat{0} \text{ then } \hat{1} \text{ else mult } n \,(f\,f \,(\text{pred } n))}$$

The underlined expression is almost the body of a typical factorial function except for the application $f\,f$. The following definition of Fac abstracts from the application $f\,f$ by replacing it by a reference to a single function g:

$$\begin{aligned}
\text{Fac} \quad &= \quad \lambda f.\,(\lambda g.\, \lambda n.\, \text{if eq } n \,\hat{0} \text{ then } \hat{1} \text{ else mult } n \,(g \,(\text{pred } n)))\,(f\,f) \\
&= \quad \lambda f.\,\underline{\text{FAC}}\,(f\,f)
\end{aligned}$$

Then fac is rewritten as follows:

$$\begin{aligned}
\text{fac} \quad &= \quad \text{Fac Fac} \\
&= \quad (\lambda f.\,\text{FAC }(f\,f))\,(\lambda f.\,\text{FAC }(f\,f)) \\
&= \quad \lambda F.\,((\lambda f.\,F \,(f\,f))\,(\lambda f.\,F \,(f\,f)))\,\text{FAC} \\
&= \quad \text{fix}_{\text{CBN}}\,\text{FAC}
\end{aligned}$$

In the case of the call-by-value strategy, fix_{CBN} FAC always diverges. A quick fix is to rewrite $f\ f$ as $\lambda x.\, f\ f\ x$ and we obtain fix:

$$
\begin{aligned}
\text{fac} \;&=\; \text{Fac Fac} \\
&=\; \lambda F.\, ((\lambda f.\, F\ (f\ f))\ (\lambda f.\, F\ (f\ f)))\ \text{FAC} \\
&=\; \lambda F.\, ((\lambda f.\, F\ (\lambda x.\, f\ f\ x))\ (\lambda f.\, F\ (\lambda x.\, f\ f\ x)))\ \text{FAC} \\
&=\; \text{fix FAC}
\end{aligned}
$$

This is how to derive the fixed point combinator fix!

3.7 De Bruijn indexes

As λ-abstractions are intended to denote mathematical functions with formal arguments, variable names may seem to be an integral part of the syntax for the λ-calculus. For example, it seems inevitable to introduce a formal argument, say x, when defining an identity function. On the other hand, a specific choice of a variable name does not affect the meaning of a λ-abstraction. For example, $\lambda x.\, x$ and $\lambda y.\, y$ both denote the same identity function even though they bind different variable names as formal arguments. In general, α-conversion enables us to rewrite any λ-abstraction into another λ-abstraction with a different name for the bound variable. This observation suggests that there may be a way to represent variables in the λ-calculus without specific names. An example of such a nameless representation of variables is *de Bruijn indexes*.

The basic idea behind de Bruijn indexes is to represent each variable by an integer value, called a de Bruijn index, instead of a name. (De Bruijn indexes can be negative, but we consider non-negative indexes only.) Roughly speaking, a de Bruijn index counts the number of λ-*binders*, such as λx, λy, and λz, lying between a given variable and its corresponding (unique) λ-binder. For example, x in the body of $\lambda x.\, x$ is assigned a de Bruijn index 0 because there is no intervening λ-binder between x and λx. In contrast, x in the body of $\lambda x.\, \lambda y.\, x\ y$ is assigned a de Bruin index 1 because there lies an intervening λ-binder λy between x and λx. Thus a de Bruijn index for a variable specifies the relative position of its corresponding λ-binder. This, in turn, implies that the same variable can be assigned different de Bruijn indexes depending on its position. For example, in $\lambda x.\, x\ (\lambda y.\, x\ y)$, the first occurrence of x is assigned 0 whereas the second occurrence is assigned 1 because of the λ-binder λy.

Since all variables are now represented by integer values, there is no need to explicitly introduce variables in λ-abstractions. In fact, it is impossible because the same variable can be assigned different de Bruijn indexes. Thus, expressions with de Bruijn indexes, or *de Bruijn expressions*, are inductively defined as follows:

$$
\begin{aligned}
\text{de Bruijn expression} \quad & M \;::=\; n \mid \lambda.\, M \mid M\ M \\
\text{de Bruijn index} \quad & n \;::=\; 0 \mid 1 \mid 2 \mid \cdots
\end{aligned}
$$

For de Bruijn expressions, we use metavariables M and N; for de Bruijn indexes, we use metavariables n, m, and i.

We write $e \equiv_{\text{dB}} M$ to mean that an ordinary expression e is converted to a de Bruijn expression M. Sometimes it suffices to literally count the number of λ-binders lying between each variable and its corresponding λ-binder, as in all the examples given above:

$$
\begin{aligned}
\lambda x.\, x \quad &\equiv_{\text{dB}} \quad \lambda.\, 0 \\
\lambda x.\, \lambda y.\, x\ y \quad &\equiv_{\text{dB}} \quad \lambda.\, \lambda.\, 1\ 0 \\
\lambda x.\, x\ (\lambda y.\, x\ y) \quad &\equiv_{\text{dB}} \quad \lambda.\, 0\ (\lambda.\, 1\ 0)
\end{aligned}
$$

In general, however, converting an ordinary expression e into a de Bruijn expression requires us to interpret e as a tree-like structure rather than a linear structure. As an example, consider $\lambda x.\, (x\ (\lambda y.\, x\ y))\ (\lambda z.\, x\ z)$. Literally counting the number of λ-binders results in a de Bruijn expression $\lambda.\, (0\ (\lambda.\, 1\ 0))\ (\lambda.\, 2\ 0)$, in which the last occurrence of x is assigned a (wrong) de Bruijn index 2 because of λy and λz. Intuitively, however, the last occurrence of x must be assigned a de Bruijn index 1 because its corresponding λ-binder can be located irrespective of the λ-binder λy. Thus, a proper way to convert an expression e to a de Bruijn expression is to count the number of λ-binders found along the way from each variable to its corresponding λ-binder in the tree-like representation of e. For example, we have $\lambda x.\, (x\ (\lambda y.\, x\ y))\ (\lambda z.\, x\ z) \equiv_{\text{dB}} \lambda.\, (0\ (\lambda.\, 1\ 0))\ (\lambda.\, 1\ 0)$ as illustrated below:

$\lambda x.$ — $@$ — $@$, $\lambda z.$ — x, $\lambda y.$, $@$, x, z — $@$ — x, y $\equiv_{\mathsf{dB}}$ $\lambda.$ — $@$ — $@$, $\lambda.$ — 0, $\lambda.$, $@$, 1, 0 — $@$ — 1, 0
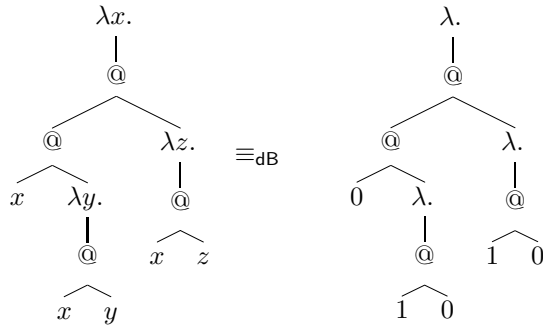
3.7.1 Substitution

In order to exploit de Bruijn indexes in implementing the operational semantics of the λ-calculus, we need a definition of substitution for de Bruijn expressions, from which a definition of β-reduction can be derived. We wish to define $\sigma_0(M, N)$ such that the following relationship holds:

$$
\begin{array}{ccc}
(\lambda x.\, e)\, e' & \mapsto & [e'/x]e \\
\equiv_{\mathsf{dB}} & & \equiv_{\mathsf{dB}} \\
(\lambda.\, M)\, N & \mapsto & \sigma_0(M, N)
\end{array}
$$

That is, applying $\lambda.\, M$ to N, or substituting N for the variable bound in $\lambda.\, M$, results in $\sigma_0(M, N)$. (The meaning of the subscript 0 in $\sigma_0(M, N)$ is explained later.)

Instead of beginning with a complete definition of $\sigma_0(M, N)$, let us refine it through a series of examples. Consider the following example in which the redex is underlined:

$$
\begin{array}{ccc}
\lambda x.\, \lambda y.\, \underline{(\lambda z.\, x\ y\ z)\ (\lambda w.\, w)} & \mapsto & \lambda x.\, \lambda y.\, x\ y\ (\lambda w.\, w) \\
\equiv_{\mathsf{dB}} & & \equiv_{\mathsf{dB}} \\
\lambda.\, \lambda.\, \underline{(\lambda.\, 2\ 1\ 0)\ (\lambda.\, 0)} & \mapsto & \lambda.\, \lambda.\, 1\ 0\ (\lambda.\, 0)
\end{array}
$$

We observe that 0, which corresponds to z bound in the λ-abstraction $\lambda z.\, x\ y\ z$, is replaced by the argument $\lambda.\, 0$. The other indexes 1 and 2 are decremented by one because the λ-binder λz disappears. These two observations lead to the following partial definition of $\sigma_0(M, N)$:

$$
\begin{array}{rcll}
\sigma_0(M_1\ M_2, N) & = & \sigma_0(M_1, N)\, \sigma_0(M_2, N) & \\
\sigma_0(0, N) & = & N & \\
\sigma_0(m, N) & = & m - 1 & \text{if } m > 0
\end{array}
$$

To see how the remaining case $\sigma_0(\lambda.\, M, N)$ is defined, consider another example in which the redex is underlined:

$$
\begin{array}{ccc}
\lambda x.\, \lambda y.\, \underline{(\lambda z.\, (\lambda u.\, x\ y\ z\ u))\ (\lambda w.\, w)} & \mapsto & \lambda x.\, \lambda y.\, (\lambda u.\, x\ y\ (\lambda w.\, w)\ u) \\
\equiv_{\mathsf{dB}} & & \equiv_{\mathsf{dB}} \\
\lambda.\, \lambda.\, \underline{(\lambda.\, (\lambda.\, 3\ 2\ 1\ 0))\ (\lambda.\, 0)} & \mapsto & \lambda.\, \lambda.\, (\lambda.\, 2\ 1\ (\lambda.\, 0)\ 0)
\end{array}
$$

We observe that unlike in the first example, 0 remains intact because it corresponds to u bound in $\lambda u.\, x\ y\ z\ u$, while 1 corresponds to z and is thus replaced by $\lambda.\, 0$. The reason why 1 is now replaced by $\lambda.\, 0$ is that in general, a de Bruijn index m outside $\lambda.\, M$ points to the same variable as $m + 1$ inside $\lambda.\, M$, *i.e.*, within M. This observation leads to an equation $\sigma_0(\lambda.\, M, N) = \lambda.\, \sigma_1(M, N)$ where $\sigma_1(M, N)$ is defined as follows:

$$
\begin{array}{rcll}
\sigma_1(M_1\ M_2, N) & = & \sigma_1(M_1, N)\, \sigma_1(M_2, N) & \\
\sigma_1(0, N) & = & 0 & \\
\sigma_1(1, N) & = & N & \\
\sigma_1(m, N) & = & m - 1 & \text{if } m > 1
\end{array}
$$

In the two examples above, we see that the subscript n in $\sigma_n(M, N)$ serves as a "boundary" index: m remains intact if $m < n$, m is replaced by N if $m = n$, and m is decremented by one if $m > n$.

Alternatively n in $\sigma_n(M, N)$ may be read as the number of λ-binders enclosing M as illustrated below:

$$\sigma_0(\underbrace{\lambda.\,\lambda.\,\cdots\lambda.}_{n}\,M, N) \;=\; \underbrace{\lambda.\,\lambda.\,\cdots\lambda.}_{n}\,\sigma_n(M, N)$$

The following definition of $\sigma_n(M, N)$ uses n as a boundary index and also generalizes the relationship between $\sigma_0(\lambda.\,M, N)$ and $\lambda.\,\sigma_1(M, N)$:

$$
\begin{aligned}
\sigma_n(M_1\ M_2, N) &= \sigma_n(M_1, N)\,\sigma_n(M_2, N) \\
\sigma_n(\lambda.\,M, N) &= \lambda.\,\sigma_{n+1}(M, N) \\
\sigma_n(m, N) &= m && \text{if } m < n \\
\sigma_n(n, N) &= N \\
\sigma_n(m, N) &= m - 1 && \text{if } m > n
\end{aligned}
$$

The following example combines the two examples given above:

$$
\begin{aligned}
\lambda x.\,\lambda y.\,\underline{(\lambda z.\,(\lambda u.\,x\ y\ z\ u)\ (x\ y\ z))\ (\lambda w.\,w)} &\;\mapsto\; \lambda x.\,\lambda y.\,(\lambda u.\,x\ y\ (\lambda w.\,w)\ u)\ (x\ y\ (\lambda w.\,w)) \\
\equiv_{\mathsf{dB}} \qquad\qquad &\qquad\qquad\qquad \equiv_{\mathsf{dB}} \\
\lambda.\,\lambda.\,\underline{(\lambda.\,(\lambda.\,3\ 2\ 1\ 0)\ (2\ 1\ 0))\ (\lambda.\,0)} &\;\mapsto\; \lambda.\,\lambda.\,(\lambda.\,2\ 1\ (\lambda.\,0)\ 0)\ (1\ 0\ (\lambda.\,0))
\end{aligned}
$$

The use of de Bruijn indexes obviates the need for α-conversion because variable names never clash. Put simply, there is no need to rename bound variables to avoid variable captures because variables have no names anyway.

3.7.2 Shifting

Although the previous definition of $\sigma_n(M, N)$ is guaranteed to work if N is closed, it may not work if N represents an expression with free variables. To be specific, the equation $\sigma_n(n, N) = N$ ceases to hold if $n > 0$ and N represents an expression with free variables. Consider the following example in which the redex is underlined:

$$
\begin{aligned}
\lambda x.\,\lambda y.\,\underline{(\lambda z.\,(\lambda u.\,z)\ z)\ (\lambda w.\,x\ y\ w)} &\;\mapsto\; \lambda x.\,\lambda y.\,(\lambda u.\,\lambda w.\,x\ y\ w)\ (\lambda w.\,x\ y\ w) \\
\equiv_{\mathsf{dB}} \qquad\qquad &\qquad\qquad \equiv_{\mathsf{dB}} \\
\lambda.\,\lambda.\,\underline{(\lambda.\,(\lambda.\,1)\ 0)\ (\lambda.\,2\ 1\ 0)} &\;\mapsto\; \lambda.\,\lambda.\,(\lambda.\,\lambda.\,3\ 2\ 0)\ (\lambda.\,2\ 1\ 0)
\end{aligned}
$$

The previous definition of $\sigma_n(M, N)$ yields a wrong result because $\sigma_1(1, \lambda.\,2\ 1\ 0)$ yields $\lambda.\,2\ 1\ 0$ instead of $\lambda.\,3\ 2\ 0$:

$$
\begin{aligned}
\underline{(\lambda.\,(\lambda.\,1)\ 0)\ (\lambda.\,2\ 1\ 0)} &\;\mapsto\; \sigma_0((\lambda.\,1)\ 0, \lambda.\,2\ 1\ 0) \\
&= \sigma_0(\lambda.\,1, \lambda.\,2\ 1\ 0)\,\sigma_0(0, \lambda.\,2\ 1\ 0) \\
&= (\lambda.\,\sigma_1(1, \lambda.\,2\ 1\ 0))\,\sigma_0(0, \lambda.\,2\ 1\ 0) \\
&= (\lambda.\,\lambda.\,2\ 1\ 0)\,(\lambda.\,2\ 1\ 0) \\
&\neq (\lambda.\,\lambda.\,3\ 2\ 0)\,(\lambda.\,2\ 1\ 0)
\end{aligned}
$$

To see why $\sigma_n(n, N) = N$ fails to hold in general, recall that the subscript n in $\sigma_n(n, N)$ denotes the number of λ-binders enclosing the de Bruijn index n:

$$\sigma_0(\underbrace{\lambda.\,\lambda.\,\cdots\lambda.}_{n}\,n, N) \;=\; \underbrace{\lambda.\,\lambda.\,\cdots\lambda.}_{n}\,\sigma_n(n, N)$$

Therefore all de Bruijn indexes in N corresponding to free variables must be shifted by n after the substitution so that they correctly skip those n λ-binders enclosing the de Bruijn index n. For example, we have:

$$\sigma_0(\underbrace{\lambda.\,\lambda.\,\cdots\lambda.}_{n}\,n, m) \;=\; \underbrace{\lambda.\,\lambda.\,\cdots\lambda.}_{n}\,\sigma_n(n, m) \;=\; \underbrace{\lambda.\,\lambda.\,\cdots\lambda.}_{n}\,m + n$$

(Here $m + n$ is a single de Bruijn index adding m and n, not a composite de Bruijn index expression consisting of m, n, and $+$.)

Let us write $\tau^n(N)$ for shifting by n all de Bruijn indexes in N corresponding to free variables. Now we use $\sigma_n(n, N) = \tau^n(N)$ instead of $\sigma_n(n, N) = N$. A partial definition of $\tau^n(N)$ is given as follows:

$$
\begin{aligned}
\tau^n(N_1\ N_2) &= \tau^n(N_1)\,\tau^n(N_2) \\
\tau^n(m) &= m + n
\end{aligned}
$$

The remaining case $\tau^n(\lambda.\,N)$, however, cannot be defined inductively in terms of $\tau^n(N)$, for example, like $\tau^n(\lambda.\,N) = \lambda.\,\tau^n(N)$. The reason is that within N, not every de Bruijn index corresponds to a free variable: 0 finds its corresponding λ-binder in $\lambda.\,N$ and thus must remain intact.

This observation suggests that we need to maintain another "boundary" index (similar to the boundary index n in $\sigma_n(M, N)$) in order to decide whether a given de Bruijn index corresponds to a free variable or not. For example, if the boundary index for $\lambda.\,N$ starts at 0, it increments to 1 within N. Thus we are led to use a general form $\tau_i^n(N)$ for shifting by n all de Bruijn indexes in N corresponding to free variables where a de Bruijn index m in N such that $m < i$ does not count as a free variable. Formally $\tau_i^n(N)$ is defined as follows:

$$
\begin{aligned}
\tau_i^n(N_1\ N_2) &= \tau_i^n(N_1)\,\tau_i^n(N_2) \\
\tau_i^n(\lambda.\,N) &= \lambda.\,\tau_{i+1}^n(N) \\
\tau_i^n(m) &= m + n && \text{if } m \geq i \\
\tau_i^n(m) &= m && \text{if } m < i
\end{aligned}
$$

Accordingly the complete definition of $\sigma_n(M, N)$ is given as follows:

$$
\begin{aligned}
\sigma_n(M_1\ M_2, N) &= \sigma_n(M_1, N)\,\sigma_n(M_2, N) \\
\sigma_n(\lambda.\,M, N) &= \lambda.\,\sigma_{n+1}(M, N) \\
\sigma_n(m, N) &= m && \text{if } m < n \\
\sigma_n(n, N) &= \tau_0^n(N) \\
\sigma_n(m, N) &= m - 1 && \text{if } m > n
\end{aligned}
$$

Now $(\lambda.\,(\lambda.\,1)\,0)\,(\lambda.\,2\,1\,0)$ from the earlier example reduces correctly:

$$
\begin{aligned}
\underline{(\lambda.\,(\lambda.\,1)\,0)\,(\lambda.\,2\,1\,0)} &\mapsto \sigma_0((\lambda.\,1)\,0, \lambda.\,2\,1\,0) \\
&= \sigma_0(\lambda.\,1, \lambda.\,2\,1\,0)\,\sigma_0(0, \lambda.\,2\,1\,0) \\
&= (\lambda.\,\sigma_1(1, \lambda.\,2\,1\,0))\,\sigma_0(0, \lambda.\,2\,1\,0) \\
&= (\lambda.\,\tau_0^1(\lambda.\,2\,1\,0))\,\tau_0^0(\lambda.\,2\,1\,0) \\
&= (\lambda.\,\lambda.\,\tau_1^1(2\,1\,0))\,\lambda.\,\tau_1^0(2\,1\,0) \\
&= (\lambda.\,\lambda.\,(2+1)\,(1+1)\,0)\,(\lambda.\,(2+0)\,(1+0)\,0) \\
&= (\lambda.\,\lambda.\,3\,2\,0)\,(\lambda.\,2\,1\,0)
\end{aligned}
$$

When converting an ordinary expression e with free variables $x_0, x_1, \cdots, x_n$ into a de Bruijn expression, we may convert $\lambda x_0.\,\lambda x_1.\,\cdots \lambda x_n.\,e$ instead, which effectively assigns de Bruijn indexes $0, 1, \cdots, n$ to $x_0, x_1, \cdots, x_n$, respectively. Then we can think of reducing e as reducing $\lambda x_0.\,\lambda x_1.\,\cdots \lambda x_n.\,e$ where the n λ-binders are all ignored. In this way, we can exploit de Bruijn indexes in reducing expressions with free variables (or global variables).

3.8 Exercises

Exercise 3.10. We wish to develop a weird reduction strategy for the λ-calculus:

- Given an application $e_1\,e_2$, we first reduce e_2.

- After reducing e_2 to a value, we reduce e_1.

- When e_1 reduces to a λ-abstraction, we apply the β-reduction.

Give the rules for the reduction judgment $e \mapsto e'$ under the weird reduction strategy.

Exercise 3.11. In a reduction sequence judgment $e \mapsto^* e'$, we use $\mapsto^*$ for the reflexive and transitive closure of $\mapsto$. That is, $e \mapsto^* e'$ holds if $e \mapsto e_1 \mapsto \cdots \mapsto e_n = e'$ where $n \geq 0$. Formally we use the following inductive definition:

$$\frac{}{e \mapsto^* e} \; Refl \qquad \frac{e \mapsto e'' \quad e'' \mapsto^* e'}{e \mapsto^* e'} \; Trans$$

We would expect that $e \mapsto^* e'$ and $e' \mapsto^* e''$ together imply $e \mapsto^* e''$ because we obtain a proof of $e \mapsto^* e''$ simply by concatenating $e \mapsto e_1 \mapsto \cdots \mapsto e_n = e'$ and $e' \mapsto e'_1 \mapsto \cdots \mapsto e'_m = e''$:

$$e \mapsto e_1 \mapsto \cdots \mapsto e_n = e' \mapsto e'_1 \mapsto \cdots \mapsto e'_m = e''$$

Give a proof of this transitivity property of $\mapsto^*$: if $e \mapsto^* e'$ and $e' \mapsto^* e''$, then $e \mapsto^* e''$. To which judgment of $e \mapsto^* e'$ and $e' \mapsto^* e''$ do we have to apply rule induction?

Exercise 3.12. Define a function double $= \lambda \hat{n}. \cdots$ for doubling a given natural number encoded as a Church numeral. Specifically double $\hat{n}$ returns $\widehat{2 * n}$.

Exercise 3.13. Define an operation halve for halving a given natural number. Specifically halve $\hat{n}$ returns $\widehat{n/2}$:

- halve $\widehat{2 * k}$ returns $\hat{k}$.
- halve $(\widehat{2 * k + 1})$ returns $\hat{k}$.

You may use pair, fst, and snd without expanding them into their definitions. You may also use zero for a natural number zero and succ for finding the successor to a given natural number:

$$\begin{aligned} \text{zero} &= \hat{0} = \lambda f. \lambda x. x \\ \text{succ} &= \lambda \hat{n}. \lambda f. \lambda x. \hat{n} \; f \; (f \; x) \end{aligned}$$

Chapter 4

Simply typed λ-calculus

This chapter presents the *simply typed λ-calculus*, an extension of the λ-calculus with *types*. Since the λ-calculus in the previous chapter does not use types, we refer to it as the *untyped* λ-calculus so that we can differentiate it from the simply typed λ-calculus.

Unlike the untyped λ-calculus in which base types (such as boolean and integers) are simulated with λ-abstractions, the simply typed λ-calculus assumes a fixed set of base types with primitive constructs. For example, we may choose to include a base type bool with boolean constants true and false and a conditional construct if e then e_1 else e_2. Thus the simply typed λ-calculus may be thought of as not just a core calculus for investigating the expressive power but indeed a subset of a functional language. Then any expression in the simply typed λ-calculus can be literally translated in a functional language such as SML.

As with the untyped λ-calculus, we first formulate the abstract syntax and operational semantics of the simply typed λ-calculus. The difference in the operational semantics is nominal because types play no role in reducing expressions. A major change arises from the introduction of a *type system*, a collection of judgments and inference rules for assigning types to expressions. The type assigned to an expression determines the form of the value to which it evaluates. For example, an expression of type bool may evaluate to either true or false, but nothing else.

The focus of the present chapter is on *type safety*, the most basic property of a type system that an expression with a valid type, or a *well-typed* expression, cannot go wrong at runtime. Since an expression is assigned a type at compile time and type safety ensures that a well-typed expression is well-behaved at runtime, we do not need the trial and error method (of running a program to locate the source of bugs in it) in order to detect fatal bugs such as adding memory addresses, subtracting an integer from a string, using an integer as a destination address in a function invocation, and so forth. Since it is often these simple (and stupid) bugs that cause considerable delay in software development, type safety offers a huge advantage over those programming languages without type systems or with type systems that fail to support type safety. Type safety is also the reason behind the phenomenon that programs that successfully compile run correctly in many cases.

Every extension to the simply typed λ-calculus discussed in this course will preserve type safety. We definitely do not want to squander our time developing a programming language as uncivilized as C!

4.1 Abstract syntax

The abstract syntax for the simply typed λ-calculus is given as follows:

$$
\begin{array}{llll}
\text{type} & A & ::= & P \mid A \to A \\
\text{base type} & P & ::= & \text{bool} \\
\text{expression} & e & ::= & x \mid \lambda x\,{:}\,A.\,e \mid e\,e \mid \\
& & & \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \\
\text{value} & v & ::= & \lambda x\,{:}\,A.\,e \mid \text{true} \mid \text{false}
\end{array}
$$

A type is either a *base type P* or a *function type* $A \rightarrow A'$. A base type is a type whose primitive constructs are given as part of the definition. Here we use a boolean type bool as a base type with which three primitive constructs are associated: boolean constants true and false and a conditional construct if e then e_1 else e_2. A function type $A \rightarrow A'$ describes those functions taking an argument of type A and returning a result of type A'. We use metavariables A, B, C for types.

It is important that the simply typed λ-calculus does not stipulate specific base types. In other words, the simply typed λ-calculus is just a framework for functional languages whose type system is extensible with additional base types. For example, the definition above considers bool as the only base type, but it should also be clear how to extend the definition with another base type (*e.g.*, an integer type int with integer constants and arithmetic operators). On the other hand, the simply typed λ-calculus must have at least one base type. Otherwise the set P of base types is empty, which in turn makes the set A of types empty. Then we would never be able to create an expression with a valid type!

As in the untyped λ-calculus, expressions include variables, λ-abstractions or functions, and applications. A λ-abstraction $\lambda x \colon A.\,e$ now explicitly specifies the type A of its formal argument x. If $\lambda x \colon A.\,e$ is applied to an expression of a different type A' (*i.e.*, $A \neq A'$), the application does not typecheck and thus has no type, as will be seen in Section 4.3. We say that variable x is bound to type A in a λ-abstraction $\lambda x \colon A.\,e$, or that a λ-abstraction $\lambda x \colon A.\,e$ binds variable x to type A.

4.2 Operational semantics

The development of the operational semantics of the simply typed λ-calculus is analogous to the case for the untyped λ-calculus: we define a mapping $FV(e)$ to calculate the set of free variables in e, a capture-avoiding substitution $[e'/x]e$, and a reduction judgment $e \mapsto e'$ with reduction rules. Since the simply typed λ-calculus is no different from the untyped λ-calculus except for its use of a type system, its operational semantics reverts to the operational semantics of the untyped λ-calculus if we ignore types in expressions.

A mapping $FV(e)$ is defined as follows:

$$
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(\lambda x \colon A.\,e) & = & FV(e) - \{x\} \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(\text{true}) & = & \varnothing \\
FV(\text{false}) & = & \varnothing \\
FV(\text{if } e \text{ then } e_1 \text{ else } e_2) & = & FV(e) \cup FV(e_1) \cup FV(e_2)
\end{array}
$$

As in the untyped λ-calculus, we say that an expression is closed if it contains no free variables.

A capture-avoiding substitution $[e'/x]e$ is defined as follows:

$$
\begin{array}{rcll}
[e'/x]x & = & e' & \\
[e'/x]y & = & y & \text{if } x \neq y \\
[e'/x]\lambda x \colon A.\,e & = & \lambda x \colon A.\,e & \\
[e'/x]\lambda y \colon A.\,e & = & \lambda y \colon A.\,[e'/x]e & \text{if } x \neq y, y \notin FV(e') \\
[e'/x](e_1\ e_2) & = & [e'/x]e_1\ [e'/x]e_2 & \\
[e'/x]\text{true} & = & \text{true} & \\
[e'/x]\text{false} & = & \text{false} & \\
[e'/x]\text{if } e \text{ then } e_1 \text{ else } e_2 & = & \text{if } [e'/x]e \text{ then } [e'/x]e_1 \text{ else } [e'/x]e_2 &
\end{array}
$$

When a variable capture occurs in $[e'/x]\lambda y \colon A.\,e$, we rename the bound variable y using the α-equivalence relation $\equiv_\alpha$. We omit the definition of $\equiv_\alpha$ because it requires no further consideration than the definition given in Chapter 3.

As with the untyped λ-calculus, different reduction strategies yield different reduction rules for the reduction judgment $e \mapsto e'$. We choose the call-by-value strategy which lends itself well to extending the simply typed λ-calculus with *computational effects* such as mutable references, exceptions, and continuations (to be discussed in subsequent chapters). Thus we use the following reduction rules:

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ Lam \qquad \frac{e_2 \mapsto e_2'}{(\lambda x\!:\!A.\ e)\ e_2 \mapsto (\lambda x\!:\!A.\ e)\ e_2'}\ Arg \qquad \frac{}{(\lambda x\!:\!A.\ e)\ v \mapsto [v/x]e}\ App$$

$$\frac{e \mapsto e'}{\text{if } e \text{ then } e_1 \text{ else } e_2 \mapsto \text{if } e' \text{ then } e_1 \text{ else } e_2}\ If$$

$$\frac{}{\text{if true then } e_1 \text{ else } e_2 \mapsto e_1}\ If_{\text{true}} \qquad \frac{}{\text{if false then } e_1 \text{ else } e_2 \mapsto e_2}\ If_{\text{false}}$$

The rules Lam, Arg, and App are exactly the same as in the untyped λ-calculus except that we use a λ-abstraction of the form $\lambda x\!:\!A.\ e$. (To implement the call-by-name strategy, we remove the rule Arg and rewrite the rule App as $\dfrac{}{(\lambda x\!:\!A.\ e)\ e' \mapsto [e'/x]e}\ App$.) The three rules If, If_{true}, and If_{false} combined together specify how to reduce a conditional construct if e then e_1 else e_2:

- We reduce e to either true or false.

- If e reduces to true, we choose the then branch and begin to reduce e_1.

- If e reduces to false, we choose the else branch and begin to reduce e_2.

As before, we write $\mapsto^*$ for the reflexive and transitive closure of $\mapsto$. We say that e evaluates to v if $e \mapsto^* v$ holds.

4.3 Type system

The goal of this section is to develop a system of inference rules for assigning types to expressions in the simply typed λ-calculus. We use a judgment called a *typing judgment*, and refer to inference rules deducing a typing judgment as *typing rules*. The resultant system is called the *type system* of the simply typed λ-calculus.

To figure out the right form for the typing judgment, let us consider an identity function $\text{id} = \lambda x\!:\!A.\ x$. Intuitively id has a function type $A \to A$ because it takes an argument of type A and returns a result of the same type. Then how do we determine, or "infer," the type of id? Since id is a λ-abstraction with an argument of type A, all we need is the type of its body. It is easy to see, however, that its body cannot be considered in isolation: without any assumption on the type of its argument x, we cannot infer the type of its body x.

The example of id suggests that it is inevitable to use assumptions on types of variables in typing judgments. Thus we are led to introduce a *typing context* to denote an unordered set of assumptions on types of variables; we use a *type binding* $x : A$ to mean that variable x assumes type A:

$$\text{typing context} \qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A$$

- $\cdot$ denotes an empty typing context and is our notation for an empty set $\varnothing$.

- $\Gamma, x : A$ augments Γ with a type binding $x : A$ and is our notation for $\Gamma \cup \{x : A\}$. We abbreviate $\cdot, x : A$ as $x : A$ to denote a singleton typing context $\{x : A\}$.

- We use the notation for typing contexts in a flexible way. For example, $\Gamma, x : A, \Gamma'$ denotes $\Gamma \cup \{x : A\} \cup \Gamma'$, and Γ, Γ' denotes $\Gamma \cup \Gamma'$.

For the sake of simplicity, we assume that variables in a typing context are all distinct. That is, $\Gamma, x : A$ is not defined if Γ contains another type binding of the form $x : A'$, or simply if $x : A' \in \Gamma$.

The type system uses the following form of typing judgment:

$$\Gamma \vdash e : A \qquad \Leftrightarrow \qquad \textit{expression } e \textit{ has type } A \textit{ under typing context } \Gamma$$

$\Gamma \vdash e : A$ means that if we use each type binding $x : A$ in Γ as an assumption, we can show that expression e has type A. An easy way to understand the role of Γ is by thinking of it as a set of type bindings for free variables in e, although Γ may also contain type bindings for those variables not found in e. For example, a closed expression e of type A needs a typing judgment $\cdot \vdash e : A$ with an empty typing context (because it contains no free variables), whereas an expression e' with a free variable x needs a typing judgment $\Gamma \vdash e' : A'$ where Γ contains at least a type binding $x : B$ for some type B.[1]

With the above interpretation of typing judgments, we can now explain the typing rules for the simply typed λ-calculus:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ Var} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x{:}A.\, e : A \to B} \to\text{I} \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\, e' : B} \to\text{E}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{ True} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{ False} \qquad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : A} \text{ If}$$

- The rule Var means that a type binding in a typing context is an assumption. Alternatively we may rewrite the rule as follows:

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \text{ Var}$$

- The rule $\to$I says that if e has type B under the assumption that x has type A, then $\lambda x{:}A.\, e$ has type $A \to B$. If we read the rule $\to$I from the premise to the conclusion (*i.e.*, top-down), we "introduce" a function type $A \to B$ from the judgment in the premise, which is the reason why it is called the "$\to$ Introduction rule." Note that if Γ already contains a type binding for variable x (*i.e.*, $x : A' \in \Gamma$), we rename x to a fresh variable by α-conversion. Hence we may assume without loss of generality that variable clashes never occur in the rule $\to$I.

- The rule $\to$E says that if e has type $A \to B$ and e' has type A, then $e\, e'$ has type B. If we read the rule $\to$E from the premise to the conclusion, we "eliminate" a function type $A \to B$ to produce an expression of a smaller type B, which is the reason why it is called the "$\to$ Elimination rule."

- The rules True and False assign base type bool to boolean constants true and false. Note that typing context Γ is not used because there is no free variable in true and false.

- The rule If says that if e has type bool and both e_1 and e_2 have the same type A, then if e then e_1 else e_2 has type A.

A derivation tree for a typing judgment is called a *typing derivation*. Here are a few examples of valid typing derivations. The first example infers the type of an identify function (where we use no premise in the rule Var):

$$\frac{\dfrac{}{\Gamma, x : A \vdash x : A} \text{ Var}}{\Gamma \vdash \lambda x{:}A.\, x : A \to A} \to\text{I}$$

Since $\lambda x{:}A.\, x$ is closed, we may use an empty typing context $\cdot$ for Γ:

$$\frac{\dfrac{}{x : A \vdash x : A} \text{ Var}}{\cdot \vdash \lambda x{:}A.\, x : A \to A} \to\text{I}$$

---

[1] A typing judgment $\Gamma \vdash e : A$ is an example of a *hypothetical judgment* which deduces a "judgment" $e : A$ using each "judgment" $x_i : A_i$ in Γ as a hypothesis. From this point of view, the turnstile symbol $\vdash$ is just a syntactic device which plays no semantic role at all. Although the notion of hypothetical judgment is of great significance in the study of logic, I do not find it particularly useful in helping students to understand the type system of the simply typed λ-calculus.

September 30, 2011

In the second example below, we abbreviate $\Gamma, x : \mathsf{bool}, y_1 : A, y_2 : A$ as Γ'. Note also that $\mathsf{bool} \to A \to A \to A$ is equivalent to $\mathsf{bool} \to (A \to (A \to A))$ because $\to$ is right-associative:

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        \cfrac{x : \mathsf{bool} \in \Gamma'}{\Gamma' \vdash x : \mathsf{bool}} \ \mathsf{Var} \quad
        \cfrac{y_1 : A \in \Gamma'}{\Gamma' \vdash y_1 : A} \ \mathsf{Var} \quad
        \cfrac{y_2 : A \in \Gamma'}{\Gamma' \vdash y_2 : A} \ \mathsf{Var}
      }{\Gamma, x : \mathsf{bool}, y_1 : A, y_2 : A \vdash \mathsf{if}\ x\ \mathsf{then}\ y_1\ \mathsf{else}\ y_2 : A} \ \mathsf{If}
    }{\Gamma, x : \mathsf{bool}, y_1 : A \vdash \lambda y_2 {:} A.\ \mathsf{if}\ x\ \mathsf{then}\ y_1\ \mathsf{else}\ y_2 : A \to A} \ {\to}\mathsf{I}
  }{\Gamma, x : \mathsf{bool} \vdash \lambda y_1 {:} A.\ \lambda y_2 {:} A.\ \mathsf{if}\ x\ \mathsf{then}\ y_1\ \mathsf{else}\ y_2 : A \to A \to A} \ {\to}\mathsf{I}
}{\Gamma \vdash \lambda x {:} \mathsf{bool}.\ \lambda y_1 {:} A.\ \lambda y_2 {:} A.\ \mathsf{if}\ x\ \mathsf{then}\ y_1\ \mathsf{else}\ y_2 : \mathsf{bool} \to A \to A \to A} \ {\to}\mathsf{I}
$$

The third example infers the type of a function composing two functions f and g where we abbreviate $\Gamma, f : A \to B, g : B \to C, x : A$ as Γ':

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        \cfrac{g : B \to C \in \Gamma'}{\Gamma' \vdash g : B \to C} \ \mathsf{Var} \quad
        \cfrac{
          \cfrac{f : A \to B \in \Gamma'}{\Gamma' \vdash f : A \to B} \ \mathsf{Var} \quad
          \cfrac{x : A \in \Gamma'}{\Gamma' \vdash x : A} \ \mathsf{Var}
        }{\Gamma' \vdash f\ x : B} \ {\to}\mathsf{E}
      }{\Gamma, f : A \to B, g : B \to C, x : A \vdash g\ (f\ x) : C} \ {\to}\mathsf{E}
    }{\Gamma, f : A \to B, g : B \to C \vdash \lambda x {:} A.\ g\ (f\ x) : A \to C} \ {\to}\mathsf{I}
  }{\Gamma, f : A \to B \vdash \lambda g {:} B {\to} C.\ \lambda x {:} A.\ g\ (f\ x) : (B \to C) \to (A \to C)} \ {\to}\mathsf{I}
}{\Gamma \vdash \lambda f {:} A {\to} B.\ \lambda g {:} B {\to} C.\ \lambda x {:} A.\ g\ (f\ x) : (A \to B) \to (B \to C) \to (A \to C)} \ {\to}\mathsf{I}
$$

We close this section by proving two properties of typing judgments: *permutation* and *weakening*. The permutation property reflects the assumption that a typing context Γ is an unordered set, which means that two typing contexts are identified up to permutation. For example, $\Gamma, x : A, y : B$ is identified with $\Gamma, y : B, x : A$, with $x : A, \Gamma, y : B$, with $x : A, y : B, \Gamma$, and so on. The weakening property says that if we can prove that expression e has type A under typing context Γ, we can also prove it under another typing context Γ' augmenting Γ with a new type binding $x : A$ (because we can just ignore the new type binding). These properties are called *structural properties* of typing judgments because they deal with the structure of typing judgments rather than their derivations.[2]

Proposition 4.1 (Permutation). *If $\Gamma \vdash e : A$ and Γ' is a permutation of Γ, then $\Gamma' \vdash e : A$.*

Proof. By rule induction on the judgment $\Gamma \vdash e : A$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

Proposition 4.2 (Weakening). *If $\Gamma \vdash e : C$, then $\Gamma, x : A \vdash e : C$.*

Proof. By rule induction on the judgment $\Gamma \vdash e : C$. We show three cases. The remaining cases are similar to the case for the rule $\to\mathsf{E}$.

Case $\cfrac{y : C \in \Gamma}{\Gamma \vdash y : C} \ \mathsf{Var}$ where $e = y$:

$y : C \in \Gamma, x : A$	from $y : C \in \Gamma$
$\Gamma, x : A \vdash y : C$	by the rule Var

Case $\cfrac{\Gamma, y : C_1 \vdash e' : C_2}{\Gamma \vdash \lambda y {:} C_1.\ e' : C_1 \to C_2} \ {\to}\mathsf{I}$ where $e = \lambda y {:} C_1.\ e'$ and $C = C_1 \to C_2$:

This is the case where $\Gamma \vdash e : C$ is proven by applying the rule $\to\mathsf{I}$. In other words, the last inference rule applied in the proof of $\Gamma \vdash e : C$ is the rule $\to\mathsf{I}$. Then e must have the form $\lambda y {:} C_1.\ e'$ for some type $C = C_1 \to C_2$; otherwise the rule $\to\mathsf{I}$ cannot be applied. Then the premise is uniquely determined as $\Gamma, y : C_1 \vdash e' : C_2$.

$\Gamma, y : C_1, x : A \vdash e' : C_2$	by induction hypothesis
$\Gamma, x : A, y : C_1 \vdash e' : C_2$	by Proposition 4.1
$\Gamma, x : A \vdash \lambda y {:} C_1.\ e' : C_1 \to C_2$	by the rule $\to\mathsf{I}$

---

[2] There is no strict rule on whether a property should be called a theorem or a proposition, although a general rule of thumb is that a property relatively easy to prove is called a proposition. A property that is of great significance is usually called a theorem. (For example, it is Fermat's last theorem rather than Fermat's last proposition.) A lemma is a property proven for facilitating proofs of other theorems, propositions, or lemmas.

Case $\dfrac{\Gamma \vdash e_1 : B \rightarrow C \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1\,e_2 : C}$ →E where $e = e_1\,e_2$:

This is the case where $\Gamma \vdash e : C$ is proven by applying the rule →E. Then e must have the form $e_1\,e_2$ and the two premises are uniquely determined for some type B; otherwise the rule →E cannot be applied.

$\Gamma, x : A \vdash e_1 : B \rightarrow C$ by induction hypothesis on $\Gamma \vdash e_1 : B \rightarrow C$

$\Gamma, x : A \vdash e_2 : B$ by induction hypothesis on $\Gamma \vdash e_2 : B$

$\Gamma, x : A \vdash e_1\,e_2 : C$ by the rule →E

$\square$

As typing contexts are always assumed to be unordered sets, we implicitly use the permutation property in proofs. For example, we may deduce $\Gamma, x : A \vdash \lambda y{:}C_1.\,e' : C_1 \rightarrow C_2$ directly from $\Gamma, y : C_1, x : A \vdash e' : C_2$ without an intermediate step of permutating $\Gamma, y : C_1, x : A$ into $\Gamma, x : A, y : C_1$.

4.4 Type safety

In order to determine properties of expressions, we have developed two systems for the simply typed λ-calculus: operational semantics and type system. The operational semantics enables us to find out dynamic properties, namely values, associated with expressions. Values are dynamic properties in the sense that they can be determined only at runtime in general. For this reason, an operational semantics is also called a *dynamic semantics*. In contrast, the type system enables us to find out static properties, namely types, of expressions. Types are static properties in the sense that they are determined at compile time and remain "static" at runtime. For this reason, a type system is also called a *static semantics*.

We have developed the type system independently of the operational semantics. Therefore there remains a possibility that it does not respect the operational semantics, whether intentionally or unintentionally. For example, it may assign different types to two expressions e and e' such that $e \mapsto e'$, which is unnatural because we do not anticipate a change in type when an expression reduces to another expression. Or it may assign a valid type to a nonsensical expression, which is also unnatural because we expect every expression of a valid type to be a valid program. Type safety, the most basic property of a type system, connects the type system with the operational semantics by ensuring that it lives in harmony with the operational semantics. It is often rephrased as *"well-typed expressions cannot go wrong."*

Type safety consists of two theorems: *progress* and *type preservation*. The progress theorem states that a (closed) well-typed expression is not stuck: either it is a value or it reduces to another expression:

Theorem 4.3 (Progress). *If $\cdot \vdash e : A$ for some type A, then either e is a value or there exists e' such that $e \mapsto e'$.*

The type preservation theorem states that when a well-typed expression reduces, the resultant expression is also well-typed and has the same type; type preservation is also called *subject reduction*:

Theorem 4.4 (Type preservation). *If $\Gamma \vdash e : A$ and $e \mapsto e'$, then $\Gamma \vdash e' : A$.*

Note that the progress theorem assumes an empty typing context (hence a *closed* well-typed expression e) whereas the type preservation theorem does not. It actually makes sense if we consider whether a reduction judgment $e \mapsto e'$ is part of the conclusion or is given as an assumption. In the case of the progress theorem, we are interested in whether e reduces to another expression or not, provided that it is well-typed. Therefore we use an empty typing context to disallow free variables in e which may make its reduction impossible. If we allowed any typing context Γ, the progress theorem would be downright false, as evidenced by a simple counterexample $e = x$ which is not a value and is irreducible. In the case of the type preservation theorem, we begin with an assumption $e \mapsto e'$. Then there is no reason not to allow free variables in e because we already know that it reduces to another expression e'. Thus we use a metavariable Γ (ranging over all typing contexts) instead of an empty typing context.

Combined together, the two theorems guarantee that a (closed) well-typed expression never reduces to a stuck expression: either it is a value or it reduces to another well-typed expressions. Consider a well-typed expression e such that $\cdot \vdash e : A$ for some type A. If e is already a value, there is no need to reduce it (and hence it is not stuck). If not, the progress theorem ensures that there exists an expression e' such that $e \mapsto e'$, which is also a well-typed expression of the same type A by the type preservation theorem.

Below we prove the two theorems using rule induction. It turns out that a direct proof attempt by rule induction fails, and thus we need a couple of lemmas. These lemmas (*canonical forms* and *substitution*) are so prevalent in programming language theory that their names are worth memorizing.

4.4.1 Proof of progress

The proof of Theorem 4.3 is relatively straightforward: the theorem is written in the form "*If J holds, then $P(J)$ holds,*" and we apply rule induction to the judgment J, which is a typing judgment $\cdot \vdash e : A$. So we begin with an assumption $\cdot \vdash e : A$. If e happens to be a value, the $P(J)$ part holds trivially because the judgment "*e is a value*" holds. Thus we make a stronger assumption $\cdot \vdash e : A$ with e not being a value. Then we analyze the structure of the proof of $\cdot \vdash e : A$, which gives three cases to consider:

$$\frac{x : A \in \cdot}{\cdot \vdash x : A} \; \text{Var} \qquad \frac{\cdot \vdash e_1 : A \to B \quad \cdot \vdash e_2 : A}{\cdot \vdash e_1 \, e_2 : B} \; \to\!\text{E} \qquad \frac{\cdot \vdash e_b : \text{bool} \quad \cdot \vdash e_1 : A \quad \cdot \vdash e_2 : A}{\cdot \vdash \text{if } e_b \text{ then } e_1 \text{ else } e_2 : A} \; \text{If}$$

The case Var is impossible because $x : A$ cannot be a member of an empty typing context $\cdot$. That is, the premise $x : A \in \cdot$ is never satisfied. So we are left with the two cases $\to\!\text{E}$ and If. Let us analyze the case $\to\!\text{E}$ in depth. By the principle of rule induction, the induction hypothesis on the first premise $\cdot \vdash e_1 : A \to B$ opens two possibilities:

1. e_1 is a value.

2. e_1 is not a value and reduces to another expression e_1', *i.e.*, $e_1 \mapsto e_1'$.

If the second possibility is the case, we have found an expression to which $e_1 \, e_2$ reduces, namely $e_1' \, e_2$:

$$\frac{e_1 \mapsto e_1'}{e_1 \, e_2 \mapsto e_1' \, e_2} \; Lam$$

Now what if the first possibility is the case? Since e_1 has type $A \to B$, it is likely to be a λ-abstraction, in which case the induction hypothesis on the second premise $\cdot \vdash e_2 : A$ opens another two possibilities and we use either the rule Arg or the rule App to show the progress property. Unfortunately we do not have a formal proof that e_1 is indeed a λ-abstraction; we know only that e_1 has type $A \to B$ under an empty typing context. Our instinct, however, says that e_1 must be a λ-abstraction because it has type $A \to B$. The following lemma formalizes our instinct on the correct, or "canonical," form of a well-typed value:

Lemma 4.5 (Canonical forms).
  If v is a value of type bool, *then v is either* true *or* false.
  If v is a value of type $A \to B$, then v is a λ-abstraction $\lambda x : A.\, e$.

Proof. By case analysis of v. (Not every proof uses rule induction!)
  Suppose that v is a value of type bool. The only typing rules that assign a boolean type to a given value are True and False. Therefore v is a boolean constant true or false. Note that the rules Var, $\to\!\text{E}$, and If may assign a boolean type, but never to a value.
  Suppose that v is a value of type $A \to B$. The only typing rule that assigns a function type to a given value is $\to\!\text{I}$. Therefore v must be a λ-abstraction of the form $\lambda x : A.\, e$ (which binds variable x to type A). Note that the rules Var, $\to\!\text{E}$, and If may assign a function type, but not to a value. □
  Now we are ready to prove the progress theorem:

Proof of Theorem 4.3. By rule induction on the judgment $\cdot \vdash e : A$.
  If e is already a value, we need no further consideration. Therefore we assume that e is not a value. Then there are three cases to consider.

Case $\dfrac{x : A \in \cdot}{\cdot \vdash x : A}$ Var where $e = x$:
impossible   from $x : A \notin \cdot$

Case $\dfrac{\cdot \vdash e_1 : B \to A \quad \cdot \vdash e_2 : B}{\cdot \vdash e_1 \, e_2 : A}$ $\to\!\text{E}$ where $e = e_1 \, e_2$:

e_1 is a value or there exists e_1' such that $e_1 \mapsto e_1'$ by induction hypothesis on $\cdot \vdash e_1 : B \to A$
e_2 is a value or there exists e_2' such that $e_2 \mapsto e_2'$ by induction hypothesis on $\cdot \vdash e_2 : B$

    Subcase: e_1 is a value and e_2 is a value
    $e_1 = \lambda x{:}B.\,e_1'$ by Lemma 4.5
    $e_2 = v_2$ because e_2 is a value
    $e_1\,e_2 \mapsto [v_2/x]e_1'$ by the rule App
    We let $e' = [v_2/x]e_1'$.

    Subcase: e_1 is a value and there exists e_2' such that $e_2 \mapsto e_2'$
    $e_1 = \lambda x{:}B.\,e_1'$ by Lemma 4.5
    $e_1\,e_2 \mapsto (\lambda x{:}B.\,e_1')\,e_2'$ by the rule Arg
    We let $e' = (\lambda x{:}B.\,e_1')\,e_2'$.

    Subcase: there exists e_1' such that $e_1 \mapsto e_1'$
    $e_1\,e_2 \mapsto e_1'\,e_2$ by the rule Lam
    We let $e' = e_1'\,e_2$.

Case $\dfrac{\cdot \vdash e_b : \mathsf{bool} \quad \cdot \vdash e_1 : A \quad \cdot \vdash e_2 : A}{\cdot \vdash \mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A}$ If where $e = \mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$:
e_b is either a value or there exists e_b' such that $e_b \mapsto e_b'$ by induction hypothesis on $\cdot \vdash e_b : \mathsf{bool}$

    Subcase: e_b is a value
    e_b is either true or false   by Lemma 4.5
    $\mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_1$ or $\mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_2$ by the rule If_{true} or If_{false}
    We let $e' = e_1$ or $e' = e_2$.

    Subcase: there exists e_b' such that $e_b \mapsto e_b'$
    $\mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto \mathsf{if}\ e_b'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ by the rule If
    We let $e' = \mathsf{if}\ e_b'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. $\square$

4.4.2   Proof of type preservation

The proof of Theorem 4.4 is not as straightforward as the proof of Theorem 4.3 because the *If* part in the theorem contains two judgments: $\Gamma \vdash e : A$ and $e \mapsto e'$. (We have seen a similar case in the proof of Lemma 2.7.) Therefore we need to decide to which judgment of $\Gamma \vdash e : A$ and $e \mapsto e'$ we apply rule induction. It turns out that the type preservation theorem is a special case in which we may apply rule induction to either judgment!

    Suppose that we choose to apply rule induction to $e \mapsto e'$. Since there are six reduction rules, we need to consider (at least) six cases. The question now is: which case do we do consider first?

    As a general rule of thumb, if you are proving a property that is expected to hold, the most difficult case should be the first to consider. The rationale is that eventually you have to consider the most difficult case anyway, and by considering it at an early stage of the proof, you may find a flaw in the system or identify auxiliary lemmas required for the proof. Even if you discover a flaw in the system from the analysis of the most difficult case, you at least avoid considering easy cases more than once. Conversely, if you are trying to locate flaws in the system by proving a property that is not expected to hold, the easiest case should be the first to consider. The rationale is that the cheapest way to locate a flaw is to consider the easiest case in which the flaw manifests itself (although it is not as convincing as the previous rationale). The most difficult case may not even shed any light on hidden flaws in the system, thereby wasting your efforts to analyze it.

    Since we wish to *prove* the type preservation theorem rather than *refute* it, we consider the most difficult case of $e \mapsto e'$ first. Intuitively the most difficult case is when $e \mapsto e'$ is proven by applying the rule App, since the substitution in it may transform an application e into a completely different form of expression, for example, a conditional construct. (The rules If_{true} and If_{false} are the easiest cases because they have no premise and e' is a subexpression of e.)

    So let us consider the most difficult case in which $(\lambda x{:}A.\,e)\,v \mapsto [v/x]e$ holds. Our goal is to use an

assumption $\Gamma \vdash (\lambda x{:}A.\, e)\, v : C$ to prove $\Gamma \vdash [v/x]e : C$. The typing judgment $\Gamma \vdash (\lambda x{:}A.\, e)\, v : C$ must have the following derivation tree:

$$\dfrac{\dfrac{\Gamma, x : A \vdash e : C}{\Gamma \vdash \lambda x{:}A.\, e : A \to C} \ {\to}\mathsf{I} \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda x{:}A.\, e)\, v : C} \ {\to}\mathsf{E}$$

Therefore our new goal is to use two assumptions $\Gamma, x : A \vdash e : C$ and $\Gamma \vdash v : A$ to prove $\Gamma \vdash [v/x]e : C$. The substitution lemma below generalizes the problem:

Lemma 4.6 (Substitution). *If $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash e' : C$, then $\Gamma \vdash [e/x]e' : C$.*

The substitution lemma is similar to the type preservation theorem in that the *If* part contains two judgments. Unlike the type preservation theorem, however, we need to take great care in applying rule induction because picking up a wrong judgment makes it impossible to complete the proof!

Exercise 4.7. To which judgment do you think we have to apply rule induction in the proof of Lemma 4.6? $\Gamma \vdash e : A$ or $\Gamma, x : A \vdash e' : C$? Why?

The key observation is that $[e/x]e'$ analyzes the structure of e', not e. That is, $[e/x]e'$ searches for every occurrence of variable x in e' only to replace it by e, and thus does not even need to know the structure of e. Thus the right judgment for applying rule induction is $\Gamma, x : A \vdash e' : C$.

Proof of Lemma 4.6. By rule induction on the judgment $\Gamma, x : A \vdash e' : C$. Recall that variables in a typing context are assumed to be all distinct. We show four cases. The first two deal with those cases where e' is a variable. The remaining cases are similar to the case for the rule ${\to}\mathsf{E}$.

Case $\dfrac{y : C \in \Gamma, x : A}{\Gamma, x : A \vdash y : C}$ Var where $e' = y$ and $y : C \in \Gamma$:

This is the case where e' is a variable y that is different from x. Since $y \neq x$, the premise $y : C \in \Gamma, x : A$ implies the side condition $y : C \in \Gamma$.

$\Gamma \vdash y : C$ from $y : C \in \Gamma$
$[e/x]y = y$ from $x \neq y$
$\Gamma \vdash [e/x]y : C$

Case $\dfrac{}{\Gamma, x : A \vdash x : A}$ Var where $e' = x$ and $C = A$:

This is the case where e' is the variable x.

$\Gamma \vdash e : A$ assumption
$\Gamma \vdash [e/x]x : A$ $[e/x]x = e$

Case $\dfrac{\Gamma, x : A, y : C_1 \vdash e'' : C_2}{\Gamma, x : A \vdash \lambda y{:}C_1.\, e'' : C_1 \to C_2}$ ${\to}\mathsf{I}$ where $e' = \lambda y{:}C_1.\, e''$ and $C = C_1 \to C_2$:

Here we may assume without loss of generality that y is a fresh variable such that $y \notin FV(e)$ and $y \neq x$. If $y \in FV(e)$ or $y = x$, we can always choose a different variable by applying an α-conversion to $\lambda y{:}C_1.\, e''$.

$\Gamma, y : C_1 \vdash [e/x]e'' : C_2$ by induction hypothesis
$\Gamma \vdash \lambda y{:}C_1.\, [e/x]e'' : C_1 \to C_2$ by the rule ${\to}\mathsf{I}$
$[e/x]\lambda y{:}C_1.\, e'' = \lambda y{:}C_1.\, [e/x]e''$ from $y \notin FV(e)$ and $x \neq y$
$\Gamma \vdash [e/x]\lambda y{:}C_1.\, e'' : C_1 \to C_2$

Case $\dfrac{\Gamma, x : A \vdash e_1 : B \to C \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma, x : A \vdash e_1\, e_2 : C}$ ${\to}\mathsf{E}$ where $e' = e_1\, e_2$:

$\Gamma \vdash [e/x]e_1 : B \to C$ by induction hypothesis on $\Gamma, x : A \vdash e_1 : B \to C$
$\Gamma \vdash [e/x]e_2 : B$ by induction hypothesis on $\Gamma, x : A \vdash e_2 : B$
$\Gamma \vdash [e/x]e_1\, [e/x]e_2 : C$ by the rule ${\to}\mathsf{E}$
$\Gamma \vdash [e/x](e_1\, e_2) : C$ from $[e/x](e_1\, e_2) = [e/x]e_1\, [e/x]e_2$
$\square$

At last, we are ready to prove the type preservation theorem. The proof proceeds by rule induction on the judgment $e \mapsto e'$. It exploits the fact that there is only one typing rule for each form of expression.

For example, the only way to prove $\Gamma \vdash e_1\, e_2 : A$ is to apply the rule →E. Thus the type system is said to be *syntax-directed* in that the *syntactic* form of expression e in a judgment $\Gamma \vdash e : A$ decides, or *directs*, the rule to be applied. Since the syntax-directedness of the type system decides a unique typing rule R for deducing $\Gamma \vdash e : A$, the premises of the rule R may be assumed to hold whenever $\Gamma \vdash e : A$ holds. For example, $\Gamma \vdash e_1\, e_2 : A$ can be proven only by applying the rule →E, from which we may conclude that the two premises $\Gamma \vdash e_1 : B \to A$ and $\Gamma \vdash e_2 : B$ hold for some type B. This is called the *inversion property* which inverts the typing rule so that its conclusion justifies the use of its premises. We state the inversion property as a separate lemma.

Lemma 4.8 (Inversion). *Suppose $\Gamma \vdash e : C$.*

If $e = x$, then $x : C \in \Gamma$.
If $e = \lambda x{:}A.\, e'$, then $C = A \to B$ and $\Gamma, x : A \vdash e' : B$ for some type B.
If $e = e_1\, e_2$, then $\Gamma \vdash e_1 : A \to C$ and $\Gamma \vdash e_2 : A$ for some type A.
If $e =$ true, then $C =$ bool.
If $e =$ false, then $C =$ bool.
If $e =$ if e_b then e_1 else e_2, then $\Gamma \vdash e_b :$ bool and $\Gamma \vdash e_1 : C$ and $\Gamma \vdash e_2 : C$.

Proof. By the syntax-directedness of the type system. A formal proof proceeds by rule induction on the judgment $\Gamma \vdash e : C$. $\qquad\square$

Proof of Theorem 4.4. By rule induction on the judgment $e \mapsto e'$.

Case $\dfrac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}$ *Lam*

$\Gamma \vdash e_1\, e_2 : A$ $\hfill$ assumption
$\Gamma \vdash e_1 : B \to A$ and $\Gamma \vdash e_2 : B$ for some type B $\hfill$ by Lemma 4.8
$\Gamma \vdash e_1' : B \to A$ $\hfill$ by induction hypothesis on $e_1 \mapsto e_1'$ with $\Gamma \vdash e_1 : B \to A$
$\Gamma \vdash e_1'\, e_2 : A$ $\hfill$ from $\dfrac{\Gamma \vdash e_1' : B \to A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1'\, e_2 : A}$ →E

Case $\dfrac{e_2 \mapsto e_2'}{(\lambda x{:}B.\, e_1')\, e_2 \mapsto (\lambda x{:}B.\, e_1')\, e_2'}$ *Arg*

$\Gamma \vdash (\lambda x{:}B.\, e_1')\, e_2 : A$ $\hfill$ assumption
$\Gamma \vdash \lambda x{:}B.\, e_1' : B \to A$ and $\Gamma \vdash e_2 : B$ $\hfill$ by Lemma 4.8
$\Gamma \vdash e_2' : B$ $\hfill$ by induction hypothesis on $e_2 \mapsto e_2'$ with $\Gamma \vdash e_2 : B$
$\Gamma \vdash (\lambda x{:}B.\, e_1')\, e_2' : A$ $\hfill$ from $\dfrac{\Gamma \vdash \lambda x{:}B.\, e_1' : B \to A \quad \Gamma \vdash e_2' : B}{\Gamma \vdash (\lambda x{:}B.\, e_1')\, e_2' : A}$ →E

Case $\dfrac{}{(\lambda x{:}B.\, e_1')\, v \mapsto [v/x]e_1'}$ *App*

$\Gamma \vdash (\lambda x{:}B.\, e_1')\, v : A$ $\hfill$ assumption
$\Gamma \vdash \lambda x{:}B.\, e_1' : B \to A$ and $\Gamma \vdash v : B$ $\hfill$ by Lemma 4.8
$\Gamma, x : B \vdash e_1' : A$ $\hfill$ by Lemma 4.8 on $\Gamma \vdash \lambda x{:}B.\, e_1' : B \to A$
$\Gamma \vdash [v/x]e_1' : A$ $\hfill$ by applying Lemma 4.6 to $\Gamma \vdash v : B$ and $\Gamma, x : B \vdash e_1' : A$

Case $\dfrac{e_b \mapsto e_b'}{\text{if } e_b \text{ then } e_1 \text{ else } e_2 \mapsto \text{if } e_b' \text{ then } e_1 \text{ else } e_2}$ *If*

$\Gamma \vdash$ if e_b then e_1 else $e_2 : A$ $\hfill$ assumption
$\Gamma \vdash e_b :$ bool and $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ $\hfill$ by Lemma 4.8
$\Gamma \vdash e_b' :$ bool $\hfill$ by induction hypothesis on $e_b \mapsto e_b'$ with $\Gamma \vdash e_b :$ bool
$\Gamma \vdash$ if e_b' then e_1 else $e_2 : A$ $\hfill$ from $\dfrac{\Gamma \vdash e_b' : \text{bool} \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \text{if } e_b' \text{ then } e_1 \text{ else } e_2 : A}$ *If*

Case $\dfrac{}{\text{if true then } e_1 \text{ else } e_2 \mapsto e_1}$ *If*$_{\text{true}}$

$\Gamma \vdash$ if true then e_1 else $e_2 : A$ $\hfill$ assumption
$\Gamma \vdash$ true $:$ bool and $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ $\hfill$ by Lemma 4.8
$\Gamma \vdash e_1 : A$

Case $\dfrac{}{\text{if false then } e_1 \text{ else } e_2 \mapsto e_2}$ $\textit{If}_{\text{false}}$

(Similar to the case for the rule $\textit{If}_{\text{true}}$) $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad$ $\square$

4.5 Exercises

Exercise 4.9. Prove Theorem 4.4 by rule induction on the judgment $\Gamma \vdash e : A$.

Exercise 4.10. For the simply typed λ-calculus considered in this chapter, prove the following structural property. The property is called *contraction* because it enables us to contract $x : A, x : A$ in a typing context to $x : A$.

$$\text{If } \Gamma, x : A, x : A \vdash e : C, \text{ then } \Gamma, x : A \vdash e : C.$$

In your proof, you may assume that a typing context Γ is an unordered set. That is, you may identify typing contexts up to permutation. For example, $\Gamma, x : A, y : B$ is identified with $\Gamma, y : B, x : A$. As is already implied by the theorem, however, you may not assume that variables in a typing context are all distinct. A typing context may even contain multiple bindings with different types for the same variable. For example, $\Gamma = \Gamma', x : A, x : B$ is a valid typing context even if $A \neq B$. (In this case, x can have type A or type B, and thus typechecking is ambiguous. Still the type system is sound.)

Chapter 5

Extensions to the simply typed λ-calculus

This chapter presents three extensions to the simply typed λ-calculus: *product types*, *sum types*, and the *fixed point construct*. Product types account for pairs, tuples, records, and units in SML. Sum types are sometimes (no pun intended!) called *disjoint unions* and can be thought of as special cases of datatypes in SML. Like the fixed point *combinator* for the untyped λ-calculus, the fixed point *construct* enables us to encode recursive functions in the simply typed λ-calculus. Unlike the fixed point combinator, however, it is *not* syntactic sugar: it cannot be written as another expression and its addition strictly increases the expressive power of the simply typed λ-calculus.

5.1 Product types

The idea behind product types is that a value of a product type $A_1 \times A_2$ contains a value of type A_1 *and also* a value of type A_2. In order to create an expression of type $A_1 \times A_2$, therefore, we need two expressions: one of type A_1 and another of type A_2; we use a *pair* (e_1, e_2) to pair up two expressions e_1 and e_2. Conversely, given an expression of type $A_1 \times A_2$, we may need to extract its individual components. We use *projections* fst e and snd e to retrieve the first and the second component of e, respectively.

$$
\begin{array}{rrcl}
\text{type} & A & ::= & \cdots \mid A \times A \\
\text{expression} & e & ::= & \cdots \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e
\end{array}
$$

As with function types, a typing rule for product types is either an introduction rule or an elimination rule. Since there are two kinds of projections, we need two elimination rules ($\times \mathsf{E}_1$ and $\times \mathsf{E}_2$):

$$
\frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2} \times \mathsf{I} \qquad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \mathsf{fst}\ e : A_1} \times \mathsf{E}_1 \qquad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \mathsf{snd}\ e : A_2} \times \mathsf{E}_2
$$

As for reduction rules, there are two alternative strategies which differ in the definition of values of product types (just like there are two reduction strategies for function types). If we take an *eager* approach, we do not regard (e_1, e_2) as a value; only if both e_1 and e_2 are values do we regard it as a value, as stated in the following definition of values:

$$
\begin{array}{rrcl}
\text{value} & v & ::= & \cdots \mid (v, v)
\end{array}
$$

Here the ellipsis $\cdots$ denotes the previous definition of values which is irrelevant to the present discussion of product types. Then the eager reduction strategy is specified by the following reduction rules:

$$\frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)} \ Pair \qquad \frac{e_2 \mapsto e_2'}{(v_1, e_2) \mapsto (v_1, e_2')} \ Pair'$$

$$\frac{e \mapsto e'}{\mathsf{fst}\ e \mapsto \mathsf{fst}\ e'} \ Fst \qquad \frac{}{\mathsf{fst}\ (v_1, v_2) \mapsto v_1} \ Fst' \qquad \frac{e \mapsto e'}{\mathsf{snd}\ e \mapsto \mathsf{snd}\ e'} \ Snd \qquad \frac{}{\mathsf{snd}\ (v_1, v_2) \mapsto v_2} \ Snd'$$

Alternatively we may take a *lazy* approach which regards (e_1, e_2) as a value:

$$\text{value} \qquad v \quad ::= \quad \cdots \mid (e, e)$$

The lazy reduction strategy reduces (e_1, e_2) "lazily" in that it postpones the reduction of e_1 and e_2 until the result is explicitly requested. It is specified by the following reduction rules:

$$\frac{e \mapsto e'}{\mathsf{fst}\ e \mapsto \mathsf{fst}\ e'} \ Fst \qquad \frac{}{\mathsf{fst}\ (e_1, e_2) \mapsto e_1} \ Fst' \qquad \frac{e \mapsto e'}{\mathsf{snd}\ e \mapsto \mathsf{snd}\ e'} \ Snd \qquad \frac{}{\mathsf{snd}\ (e_1, e_2) \mapsto e_2} \ Snd'$$

Exercise 5.1. Why is it a bad idea to reduce fst (e_1, e_2) to e_1 (and similarly for snd (e_1, e_2)) under the eager reduction strategy?

In order to incorporate these reduction rules into the operational semantics, we extend the definition of $FV(e)$ and $[e'/x]e$ accordingly:

$$\begin{array}{rclcrcl}
FV((e_1, e_2)) & = & FV(e_1) \cup FV(e_2) & \qquad & [e'/x](e_1, e_2) & = & ([e'/x]e_1, [e'/x]e_2) \\
FV(\mathsf{fst}\ e) & = & FV(e) & & [e'/x]\mathsf{fst}\ e & = & \mathsf{fst}\ [e'/x]e \\
FV(\mathsf{snd}\ e) & = & FV(e) & & [e'/x]\mathsf{snd}\ e & = & \mathsf{snd}\ [e'/x]e
\end{array}$$

5.2 General product types and unit type

Product types are easily generalized to n-ary cases $A_1 \times A_2 \times \cdots \times A_n$. A *tuple* $(e_1, e_2, \cdots, e_n)$ has a general product type $A_1 \times A_2 \times \cdots \times A_n$ if e_i has type A_i for $1 \leq i \leq n$. A projection $\mathsf{proj}_i\ e$ now uses an index i to indicate which component to retrieve from e.

$$\begin{array}{rcll}
\text{type} & A & ::= & \cdots \mid A_1 \times A_2 \times \cdots \times A_n \\
\text{expression} & e & ::= & \cdots \mid (e_1, e_2, \cdots, e_n) \mid \mathsf{proj}_i\ e
\end{array}$$

$$\frac{\Gamma \vdash e_i : A_i \quad 1 \leq i \leq n}{\Gamma \vdash (e_1, e_2, \cdots, e_n) : A_1 \times A_2 \times \cdots \times A_n} \ \times\mathsf{I} \qquad \frac{\Gamma \vdash e : A_1 \times A_2 \times \cdots \times A_n \quad 1 \leq i \leq n}{\Gamma \vdash \mathsf{proj}_i\ e : A_i} \ \times\mathsf{E}_i$$

As in binary cases, eager and lazy reduction strategies are available for general product types. Below we give the specification of the eager reduction strategy; the lazy reduction strategy is left as an exercise.

$$\text{value} \qquad v \quad ::= \quad \cdots \mid (v_1, v_2, \cdots, v_n)$$

$$\frac{e_i \mapsto e_i'}{(v_1, v_2, \cdots, v_{i-1}, e_i, \cdots, e_n) \mapsto (v_1, v_2, \cdots, v_{i-1}, e_i', \cdots, e_n)} \ Pair$$

$$\frac{e \mapsto e'}{\mathsf{proj}_i\ e \mapsto \mathsf{proj}_i\ e'} \ Proj \qquad \frac{1 \leq i \leq n}{\mathsf{proj}_i\ (v_1, v_2, \cdots, v_n) \mapsto v_i} \ Proj'$$

Of particular importance is the special case $n = 0$ in a general product type $A_1 \times A_2 \times \cdots \times A_n$. To better understand the ramifications of setting n to 0, let us interpret the rules $\times\mathsf{I}$ and $\times\mathsf{E}_i$ as follows:

- The rule $\times$I says that in order to build a value of type $A_1 \times A_2 \times \cdots \times A_n$, we have to provide n different values of types A_1 through A_n in the premise.

- The rule $\times E_i$ says that since we have already provided n different values of types A_1 through A_n, we may retrieve any of these values individually in the conclusion.

Now let us see what happens when we set n to 0:

- In order to build a value of type $A_1 \times A_2 \times \cdots \times A_0$, we have to provide 0 different values. That is, we do not have to provide any value in the premise at all!

- Since we have provided 0 different values, we cannot retrieve any value in the conclusion at all. That is, the rule $\times E_i$ never applies if $n = 0$!

The type unit is a general product type $A_1 \times A_2 \times \cdots \times A_n$ with $n = 0$. It has an introduction rule with no premise (because we do not have to provide any value), but has no elimination rule (because there is no way to retrieve a value after providing no value). An expression () is called a *unit* and is the only value belonging to type unit. The typing rule Unit below is the introduction rule for unit:

$$
\begin{array}{rcll}
\text{type} & A & ::= & \cdots \mid \text{unit} \\
\text{expression} & e & ::= & \cdots \mid () \\
\text{value} & v & ::= & \cdots \mid ()
\end{array}
$$

$$
\frac{}{\Gamma \vdash () : \text{unit}} \ \text{Unit}
$$

The type unit is useful when we introduce computational effects such as input/output and mutable references. For example, a function returning a character typed by the user does not need an argument of particular meaning. Hence it may use unit as the type of its arguments.

5.3   Sum types

The idea behind sum types is that a value of a sum type $A_1 + A_2$ contains a value of type A_1 *or else* a value of type A_2, but not both. Therefore there are two ways to create an expression of type $A_1 + A_2$: using an expression e_1 of type A_1 and using an expression e_2 of type A_2. In the first case, we use a *left injection*, or *inleft* for short, $\text{inl}_{A_2} \ e_1$; in the second case, we use a *right injection*, or *inright* for short, $\text{inr}_{A_1} \ e_2$.

Then how do we extract back a value from an expression of type $A_1 + A_2$? In general, it is unknown which of the two types A_1 and A_2 has been used in creating a value of type $A_1 + A_2$. For example, in the body e of a λ-abstraction $\lambda x : A_1 + A_2.\ e$, nothing is known about variable x except that its value can be created from a value of either type A_1 or type A_2. In order to examine the value associated with an expression of type $A_1 + A_2$, therefore, we have to provide for two possibilities: when a left injection has been used and when a right injection has been used. We use a *case expression* case e of inl $x_1.\ e_1$ | inr $x_2.\ e_2$ to perform a case analysis on expression e which must have a sum type $A_1 + A_2$. Informally speaking, if e has been created with a value v_1 of type A_1, the case expression takes the first branch, reducing e_1 after binding x_1 to v_1; otherwise it takes the second branch, reducing e_2 in an analogous way.

$$
\begin{array}{rcll}
\text{type} & A & ::= & \cdots \mid A + A \\
\text{expression} & e & ::= & \cdots \mid \text{inl}_A \ e \mid \text{inr}_A \ e \mid \text{case } e \text{ of inl } x.\ e \mid \text{inr } x.\ e
\end{array}
$$

As is the case with function types and product types, a typing rule for sum types is either an introduction rule or an elimination rule. Since there are two ways to create an expression of type $A_1 + A_2$, there are two introduction rules ($+_L$ for $\text{inl}_A \ e$ and $+_R$ for $\text{inr}_A \ e$):

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \mathsf{inl}_{A_2}\, e : A_1 + A_2}\ +\!\mathsf{I_L} \qquad \frac{\Gamma \vdash e : A_2}{\Gamma \vdash \mathsf{inr}_{A_1}\, e : A_1 + A_2}\ +\!\mathsf{I_R}$$

$$\frac{\Gamma \vdash e : A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : C \quad \Gamma, x_2 : A_2 \vdash e_2 : C}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 : C}\ +\!\mathsf{E}$$

In the rule $+$E, expressions e_1 and e_2 must have the same type; otherwise we cannot statically determine the type of the whole case expression.

As with product types, reduction rules for sum types depend on the definition of values of sum types. An eager approach uses the following definition of values:

$$\mathsf{value} \qquad v \ ::= \ \cdots \mid \mathsf{inl}_A\, v \mid \mathsf{inr}_A\, v$$

Then the eager reduction strategy is specified by the following reduction rules:

$$\frac{e \mapsto e'}{\mathsf{inl}_A\, e \mapsto \mathsf{inl}_A\, e'}\ \mathit{Inl} \qquad \frac{e \mapsto e'}{\mathsf{inr}_A\, e \mapsto \mathsf{inr}_A\, e'}\ \mathit{Inr}$$

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto \mathsf{case}\ e'\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2}\ \mathit{Case}$$

$$\frac{}{\mathsf{case}\ \mathsf{inl}_A\, v\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [v/x_1]e_1}\ \mathit{Case'}$$

$$\frac{}{\mathsf{case}\ \mathsf{inr}_A\, v\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [v/x_2]e_2}\ \mathit{Case''}$$

A lazy approach regards $\mathsf{inl}_A\, e$ and $\mathsf{inr}_A\, e$ as values regardless of the form of expression e:

$$\mathsf{value} \qquad v \ ::= \ \cdots \mid \mathsf{inl}_A\, e \mid \mathsf{inr}_A\, e$$

Then the lazy reduction strategy is specified by the following reduction rules:

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto \mathsf{case}\ e'\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2}\ \mathit{Case}$$

$$\frac{}{\mathsf{case}\ \mathsf{inl}_A\, e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [e/x_1]e_1}\ \mathit{Case'}$$

$$\frac{}{\mathsf{case}\ \mathsf{inr}_A\, e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [e/x_2]e_2}\ \mathit{Case''}$$

In extending the definition of $FV(e)$ and $[e'/x]e$ for sum types, we have to be careful about case expressions. Intuitively x_1 and x_2 in $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2$ are bound variables, just like x in $\lambda x\!:\!A.\, e$ is a bound variable. Thus x_1 and x_2 are not free variables in $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2$, and may have to be renamed to avoid variable captures in a substitution $[e'/x]\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2$.

$$
\begin{aligned}
FV(\mathsf{inl}_A\, e) &= FV(e) \\
FV(\mathsf{inr}_A\, e) &= FV(e) \\
FV(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2) &= FV(e) \cup (FV(e_1) - \{x_1\}) \cup (FV(e_2) - \{x_2\})
\end{aligned}
$$

$$
\begin{aligned}
[e'/x]\mathsf{inl}_A\, e &= \mathsf{inl}_A\, [e'/x]e \\
[e'/x]\mathsf{inr}_A\, e &= \mathsf{inr}_A\, [e'/x]e \\
[e'/x]\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 &= \mathsf{case}\ [e'/x]e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, [e'/x]e_1 \mid \mathsf{inr}\ x_2.\, [e'/x]e_2 \\
&\quad \text{if } x \neq x_1, x_1 \notin FV(e'), x \neq x_2, x_2 \notin FV(e')
\end{aligned}
$$

As an example of using sum types, let us encode the type bool. The inherent capability of a boolean value is to choose one of two different options, as mentioned in Section 3.4.1. Hence a sum type

unit+unit is sufficient for encoding the type bool because the left unit corresponds to the first option and the right unit to the second option. Then true, false, and if e then e_1 else e_2 are encoded as follows, where x_1 and x_2 are dummy variables of no significance:

$$
\begin{aligned}
\text{true} &= \text{inl}_{\text{unit}}\,() \\
\text{false} &= \text{inr}_{\text{unit}}\,() \\
\text{if } e \text{ then } e_1 \text{ else } e_2 &= \text{case } e \text{ of inl } x_1.\, e_1 \mid \text{inr } x_2.\, e_2
\end{aligned}
$$

Sum types are easily generalized to n-ary cases $A_1+A_2+\cdots+A_n$. Here we discuss the special case $n = 0$.

Consider a general sum type $A = A_1+A_2+\cdots+A_n$. We have n different ways of creating a value of type A: by providing a value of type A_1, a value of type A_2, $\cdots$, and a value of type A_n. Now what happens if $n = 0$? We have 0 different ways of creating a value of type A, which is tantamount to saying that there is no way to create a value of type A. Therefore it is impossible to create a value of type A!

Next suppose that an expression e has type $A = A_1+A_2+\cdots+A_n$. In order to examine the value associated with e and obtain an expression of another type C, we have to consider n different possibilities. (See the rule +E for the case $n = 2$.) Now what happens if $n = 0$? We have to consider 0 different possibilities, which is tantamount to saying that we do not have to consider anything at all. Therefore we can obtain an expression of an arbitrary type C for free!

The type void is a general sum type $A_1+A_2+\cdots+A_n$ with $n = 0$. It has no introduction rule because a value of type void is impossible to create. Consequently there is no value belonging to type void. The typing rule Abort below is the elimination rule for void; $\text{abort}_A\, e$ is called an *abort* expression.

$$
\begin{aligned}
\text{type} \qquad A &::= \quad \cdots \mid \text{void} \\
\text{expression} \qquad e &::= \quad \cdots \mid \text{abort}_A\, e
\end{aligned}
$$

$$
\frac{\Gamma \vdash e : \text{void}}{\Gamma \vdash \text{abort}_C\, e : C} \;\; \text{Abort}
$$

There is no reduction rule for an abort expression $\text{abort}_A\, e$: if we keep reducing expression e, we will eventually obtain a value of type void, which must never happen because there is no value of type void. So we stop!

The rule Abort may be a bit disquieting because its premise appears to contradict the fact that there is no value of type void. That is, if there is no value of type void, how can we possibly create an expression of type void? The answer is that we can never create a value of type void, but we may still "assume" that there is a value of type void. For example, $\lambda x\!:\!\text{void}.\ \text{abort}_A\, x$ is a well-typed expression of type $\text{void} \to A$ in which we "assume" that variable x has type void. In essence, there is nothing wrong with making an assumption that something impossible has actually happened.

5.4   Fixed point construct

In the untyped λ-calculus, the fixed point combinator is syntactic sugar which is just a particular expression. We may hope, then, that encoding recursive functions in the simply typed λ-calculus boils down to finding a type for the fixed point combinator from the untyped λ-calculus. Unfortunately the fixed point combinator is untypable in the sense that we cannot assign a type to it by annotating all bound variables in it with suitable types. Thus the fixed point combinator cannot be an expression in the simply typed λ-calculus.

It is not difficult to see why the fixed point combinator is untypable. Consider the fixed point combinator for the call-by-value strategy in the untyped λ-calculus:

$$
\lambda F.\, (\lambda f.\, F\, (\lambda x.\, f\, f\, x))\, (\lambda f.\, F\, (\lambda x.\, f\, f\, x))
$$

Let us assign a type A to variable f:

$$
\lambda F.\, (\lambda f\!:\!A.\, F\, (\lambda x.\, f\, f\, x))\, (\lambda f\!:\!A.\, F\, (\lambda x.\, f\, f\, x))
$$

Since f in $f\ f\ x$ is applied to f itself which is an expression of type A, it must have a type $A\to B$ for some type B. Since f can have only a single unique type, A and $A\to B$ must be identical, which is impossible.

Thus we are led to introduce a fixed point construct $\text{fix}\ x:A.\ e$ as a primitive construct (as opposed to syntactic sugar) which cannot be rewritten as an existing expression in the simply typed λ-calculus:

$$\text{expression}\qquad e\ \ ::=\ \ \cdots\ |\ \text{fix}\ x:A.\ e$$

$\text{fix}\ x:A.\ e$ is intended to find a fixed point of a λ-abstraction $\lambda x:A.\ e$. The typing rule Fix states that a fixed point is defined on a function of type $A\to A$ only, in which case it has also type A:

$$\frac{\Gamma,x:A\vdash e:A}{\Gamma\vdash \text{fix}\ x:A.\ e:A}\ \ \text{Fix}$$

Since $\text{fix}\ x:A.\ e$ is intended as a fixed point of a λ-abstraction $\lambda x:A.\ e$, the definition of fixed point justifies the following (informal) equation:

$$\text{fix}\ x:A.\ e\ \ =\ \ (\lambda x:A.\ e)\ \text{fix}\ x:A.\ e$$

As $(\lambda x:A.\ e)\ \text{fix}\ x:A.\ e$ reduces to $[\text{fix}\ x:A.\ e/x]e$ by the β-reduction, we obtain the following reduction rule for the fixed point construct:

$$\frac{}{\text{fix}\ x:A.\ e\ \mapsto\ [\text{fix}\ x:A.\ e/x]e}\ \ \textit{Fix}$$

In extending the definition of $FV(e)$ and $[e'/x]e$, we take into account the fact that y in $\text{fix}\ y:A.\ e$ is a bound variable:

$$\begin{array}{rcll}FV(\text{fix}\ x:A.\ e) & = & FV(e)-\{x\} & \\ {[e'/x]}\text{fix}\ y:A.\ e & = & \text{fix}\ y:A.\ [e'/x]e & \text{if }x\neq y, y\notin FV(e')\end{array}$$

In the case of the call-by-name strategy, the rule $\textit{Fix}$ poses no particular problem. In the case of the call-by-value strategy, however, a reduction by the rule $\textit{Fix}$ may fall into an infinite loop because $[\text{fix}\ x:A.\ e/x]e$ needs to be further reduced *unless e is already a value*:

$$\text{fix}\ x:A.\ e\ \mapsto\ [\text{fix}\ x:A.\ e/x]e\ \mapsto\ \cdots$$

For this reason, a typical functional language based on the call-by-value strategy requires that e in $\text{fix}\ x:A.\ e$ be a λ-abstraction (among all those values including integers, booleans, λ-abstractions, and so on). Hence it allows the fixed point construct of the form $\text{fix}\ f:A\to B.\ \lambda x:A.\ e$ only, which implies that it uses the fixed point construct only to define recursive functions. For example, $\text{fix}\ f:A\to B.\ \lambda x:A.\ e$ may be thought of as a recursive function f of type $A\to B$ whose formal argument is x and whose body is e. Note that its reduction immediately returns a value:

$$\text{fix}\ f:A\to B.\ \lambda x:A.\ e\ \mapsto\ \lambda x:A.\ [\text{fix}\ f:A\to B.\ \lambda x:A.\ e/f]e$$

One important question remains unanswered: how do we encode mutually recursive functions? For example, how do we encode two mutually recursive functions f_1 of type $A_1\to B_1$ and f_2 of type $A_2\to B_2$? The trick is to find a fixed point of a product type $(A_1\to B_1)\times(A_2\to B_2)$:

$$\text{fix}\ f_{12}:(A_1\to B_1)\times(A_2\to B_2).\ (\lambda x_1:A_1.\ e_1,\lambda x_2:A_2.\ e_2)$$

In expressions e_1 and e_2, we use fst f_{12} and snd f_{12} to refer to f_1 and f_2, respectively. To be precise, therefore, e in $\text{fix}\ x:A.\ e$ can be not only a λ-abstraction but also a pair/tuple of λ-abstractions.

$$\begin{array}{llll}
\text{type} & A & ::= & \cdots \mid A \times A \mid \text{unit} \mid A{+}A \mid \text{void} \\
\text{expression} & e & ::= & \cdots \mid (e,e) \mid \text{fst } e \mid \text{snd } e \mid () \mid \text{inl}_A\, e \mid \text{inr}_A\, e \mid \text{case } e \text{ of inl } x.\,e \mid \text{inr } x.\,e \mid \text{fix } x{:}A.\,e \\
\text{value} & v & ::= & \cdots \mid (v,v) \mid () \mid \text{inl}_A\, v \mid \text{inr}_A\, v
\end{array}$$

$$\frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2} \times\mathsf{I} \quad
\frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \text{fst } e : A_1} \times\mathsf{E}_1 \quad
\frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \text{snd } e : A_2} \times\mathsf{E}_2 \quad
\frac{}{\Gamma \vdash () : \text{unit}} \ \mathsf{Unit}$$

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \text{inl}_{A_2}\, e : A_1{+}A_2} +\mathsf{I}_\mathsf{L} \quad
\frac{\Gamma \vdash e : A_2}{\Gamma \vdash \text{inr}_{A_1}\, e : A_1{+}A_2} +\mathsf{I}_\mathsf{R}$$

$$\frac{\Gamma \vdash e : A_1{+}A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : C \quad \Gamma, x_2 : A_2 \vdash e_2 : C}{\Gamma \vdash \text{case } e \text{ of inl } x_1.\,e_1 \mid \text{inr } x_2.\,e_2 : C} +\mathsf{E} \quad
\frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash \text{fix } x{:}A.\,e : A} \ \mathsf{Fix}$$

$$\frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)} \ Pair \quad
\frac{e_2 \mapsto e_2'}{(v_1, e_2) \mapsto (v_1, e_2')} \ Pair' \quad
\frac{e \mapsto e'}{\text{fst } e \mapsto \text{fst } e'} \ Fst \quad
\frac{}{\text{fst } (v_1, v_2) \mapsto v_1} \ Fst'$$

$$\frac{e \mapsto e'}{\text{snd } e \mapsto \text{snd } e'} \ Snd \quad
\frac{}{\text{snd } (v_1, v_2) \mapsto v_2} \ Snd' \quad
\frac{e \mapsto e'}{\text{inl}_A\, e \mapsto \text{inl}_A\, e'} \ Inl \quad
\frac{e \mapsto e'}{\text{inr}_A\, e \mapsto \text{inr}_A\, e'} \ Inr$$

$$\frac{e \mapsto e'}{\text{case } e \text{ of inl } x_1.\,e_1 \mid \text{inr } x_2.\,e_2 \mapsto \text{case } e' \text{ of inl } x_1.\,e_1 \mid \text{inr } x_2.\,e_2} \ Case$$

$$\frac{}{\text{case inl}_A\, v \text{ of inl } x_1.\,e_1 \mid \text{inr } x_2.\,e_2 \mapsto [v/x_1]e_1} \ Case'$$

$$\frac{}{\text{case inr}_A\, v \text{ of inl } x_1.\,e_1 \mid \text{inr } x_2.\,e_2 \mapsto [v/x_2]e_2} \ Case'' \quad
\frac{}{\text{fix } x{:}A.\,e \mapsto [\text{fix } x{:}A.\,e/x]e} \ Fix$$

Figure 5.1: Definition of the extended simply typed λ-calculus with the eager reduction strategy

5.5 Type inhabitation

We say that a type A is *inhabited* if there exists an expression of type A. For example, the function type $A \to A$ is inhabited for any type A because $\lambda x{:}A.\,x$ is an example of such an expression. Interestingly not every type is inhabited in the simply typed λ-calculus without the fixed point construct. For example, there is no expression of type $((A \to B) \to A) \to A$.[1] Consequently, in order to use an expression of type $((A \to B) \to A) \to A$, we have to introduce it as a primitive construct which then strictly increases the expressive power of the simply typed λ-calculus. (callcc in Chapter 10 can be thought of such a primitive construct.)

The presence of the fixed point construct, however, completely defeats the purpose of introducing the concept of type inhabitation, since every type is now inhabited: fix $x{:}A.\,x$ has type A! In this regard, the fixed point construct is not a welcome guest to type theory.

5.6 Type safety

This section proves type safety, *i.e.*, progress and type preservation, of the extended simply typed λ-calculus:

Theorem 5.2 (Progress). *If $\cdot \vdash e : A$ for some type A, then either e is a value or there exists e' such that $e \mapsto e'$.*

Theorem 5.3 (Type preservation). *If $\Gamma \vdash e : A$ and $e \mapsto e'$, then $\Gamma \vdash e' : A$.*

We assume the eager reduction strategy and do not consider general product types and general sum types. Figure 5.1 shows the typing rules and the reduction rules to be considered in the proof. Note that the extended simply typed λ-calculus does not include an abort expression $\text{abort}_A\, e$ which destroys the progress property. (Why?)

The proof of progress extends the proof of Theorem 4.3. First we extend the canonical forms lemma (Lemma 4.5).

---

[1] In logic, $((A \to B) \to A) \to A$ is called *Peirce's Law*. Note that it is not *Pierce's Law*!

Lemma 5.4 (Canonical forms).
  If v is a value of type $A_1 \times A_2$, then v is a pair (v_1, v_2) of values.
  If v is a value of type unit, *then v is* ().
  If v is a value of type $A_1 + A_2$, then v is either $\text{inl}_{A_2}\, v'$ *or* $\text{inr}_{A_1}\, v'$.
  There is no value of type void.

Proof. By case analysis of v.

Suppose that v is a value of type $A_1 \times A_2$. The only typing rule that assigns a product type $A_1 \times A_2$ to a value is the rule $\times$I. Therefore v must be a pair. Since v is a value, it must be a pair (v_1, v_2) of values. Note that other typing rules may assign a product type, but never to a value.

Suppose that v is a value of type unit. The only typing rule that assigns type unit to a value is the rule Unit. Therefore v must be ().

Suppose that v is a value of type $A_1 + A_2$. The only typing rules that assign a sum type $A_1 + A_2$ to a value are the rules $+_\text{L}$ and $+_\text{R}$. Therefore v must be either $\text{inl}_{A_2}\, e$ or $\text{inr}_{A_1}\, e$. Since v is a value, it must be either $\text{inl}_{A_2}\, v'$ or $\text{inr}_{A_1}\, v'$.

There is no value of type void because there is no typing rule assigning type void to a value. □

The proof of Theorem 5.2 extends the proof of Theorem 4.3.

Proof of Theorem 5.2. By rule induction on the judgment $\cdot \vdash e : A$. If e is already a value, we need no further consideration. Therefore we assume that e is not a value. Then there are eight cases to consider.

Case $\dfrac{\cdot \vdash e_1 : A_1 \quad \cdot \vdash e_2 : A_2}{\cdot \vdash (e_1, e_2) : A_1 \times A_2}\ \times$I where $e = (e_1, e_2)$ and $A = A_1 \times A_2$:

e_1 is a value or there exists e_1' such that $e_1 \mapsto e_1'$ $\qquad\qquad$ by induction hypothesis on $\cdot \vdash e_1 : A_1$
e_2 is a value or there exists e_2' such that $e_2 \mapsto e_2'$ $\qquad\qquad$ by induction hypothesis on $\cdot \vdash e_2 : A_2$
Both e_1 and e_2 cannot be values simultaneously because $e = (e_1, e_2)$ is assumed not to be a value.

Subcase: e_1 is a value and there exists e_2' such that $e_2 \mapsto e_2'$
$(e_1, e_2) \mapsto (e_1, e_2')$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ by the rule *Pair'*
We let $e' = (e_1, e_2')$.

Subcase: there exists e_1' such that $e_1 \mapsto e_1'$
$(e_1, e_2) \mapsto (e_1', e_2)$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ by the rule *Pair*
We let $e' = (e_1', e_2)$.

Case $\dfrac{\cdot \vdash e_0 : A_1 \times A_2}{\cdot \vdash \text{fst}\, e_0 : A_1}\ \times$E$_1$ where $e = \text{fst}\, e_0$ and $A = A_1$:

e_0 is a value or there exists e_0' such that $e_0 \mapsto e_0'$ $\qquad\qquad$ by induction hypothesis on $\cdot \vdash e_0 : A_1 \times A_2$
e_0 cannot be a value because $e = \text{fst}\, e_0$ is assumed not to be a value.
$\text{fst}\, e_0 \mapsto \text{fst}\, e_0'$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ by the rule *Fst*
We let $e' = \text{fst}\, e_0'$.

(The cases for the rules $\times$E$_2$, $+_\text{L}$, and $+_\text{R}$ are all similar.)

Case $\dfrac{\cdot \vdash e_s : A_1 + A_2 \quad x_1 : A_1 \vdash e_1 : A \quad x_2 : A_2 \vdash e_2 : A}{\cdot \vdash \text{case}\, e_s\, \text{of}\, \text{inl}\, x_1.\, e_1 \mid \text{inr}\, x_2.\, e_2 : A}\ +$E where $e = \text{case}\, e_s\, \text{of}\, \text{inl}\, x_1.\, e_1 \mid \text{inr}\, x_2.\, e_2$:

e_s is a value or there exists e_s' such that $e_s \mapsto e_s'$ $\qquad\qquad$ by induction hypothesis on $\cdot \vdash e_s : A_1 + A_2$

Subcase: e_s is a value
$e_s = \text{inl}_{A_2}\, v$ or $e_s = \text{inr}_{A_1}\, v$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ by Lemma 5.4
$e \mapsto [v/x_1]e_1$ or $e \mapsto [v/x_2]e_2$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ by the rule *Case'* or *Case''*
We let $e' = [v/x_1]e_1$ or $e' = [v/x_2]e_2$.

Subcase: there exists e_s' such that $e_s \mapsto e_s'$
$\text{case}\, e_s\, \text{of}\, \text{inl}\, x_1.\, e_1 \mid \text{inr}\, x_2.\, e_2 \mapsto \text{case}\, e_s'\, \text{of}\, \text{inl}\, x_1.\, e_1 \mid \text{inr}\, x_2.\, e_2$ $\qquad\qquad\qquad$ by the rule *Case*
We let $e' = \text{case}\, e_s'\, \text{of}\, \text{inl}\, x_1.\, e_1 \mid \text{inr}\, x_2.\, e_2$.

Case $\dfrac{x : A \vdash e_0 : A}{\cdot \vdash \mathsf{fix}\ x : A.\ e_0 : A}$ Fix   where $e = \mathsf{fix}\ x : A.\ e_0$:

$\mathsf{fix}\ x : A.\ e_0 \mapsto [\mathsf{fix}\ x : A.\ e_0 / x] e_0$ $\hfill$ by the rule *Fix*

We let $e' = [\mathsf{fix}\ x : A.\ e_0 / x] e_0$. $\hfill\Box$

The proof of type preservation extends the proof of Theorem 4.4. First we extend the substitution lemma (Lemma 4.6) and the inversion lemma (Lemma 4.8).

Lemma 5.5 (Substitution). *If $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash e' : C$, then $\Gamma \vdash [e/x]e' : C$.*

Proof of Lemma 5.5. By rule induction on the judgment $\Gamma, x : A \vdash e' : C$. The proof extends the proof of Lemma 4.6.

The case for the rule $\times$I is similar to the case for the rule $\to$E. The cases for the rules $\times$E$_1$, $\times$E$_2$, $+$I$_L$, and $+$I$_R$ are also similar to the case for the rule $\to$E except that e' contains only one smaller subexpression (*e.g.*, $e' = \mathsf{fst}\ e_0$). The case for the rule Fix is similar to the case for the rule $\to$I.

Case $\dfrac{\Gamma, x : A \vdash e_0 : A_1 + A_2 \quad \Gamma, x : A, x_1 : A_1 \vdash e_1 : C \quad \Gamma, x : A, x_2 : A_2 \vdash e_2 : C}{\Gamma, x : A \vdash \mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inl}\ x_1.\ e_1 \mid \mathsf{inr}\ x_2.\ e_2 : C}$ $+$E

$\hfill$ where $e' = \mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inl}\ x_1.\ e_1 \mid \mathsf{inr}\ x_2.\ e_2$:

Without loss of generality, we may assume $x_1 \neq x$, $x_1 \notin FV(e)$, $x_2 \neq x$, and $x_2 \notin FV(e)$ because we can apply α-conversions to x_1 and x_2 if necessary. This case is similar to the case for the rule $\to$I.

$\Gamma \vdash [e/x]e_0 : A_1 + A_2$ $\hfill$ by induction hypothesis on $\Gamma, x : A \vdash e_0 : A_1 + A_2$

$\Gamma, x_1 : A_1 \vdash [e/x]e_1 : C$ $\hfill$ by induction hypothesis on $\Gamma, x : A, x_1 : A_1 \vdash e_1 : C$

$\Gamma, x_2 : A_2 \vdash [e/x]e_2 : C$ $\hfill$ by induction hypothesis on $\Gamma, x : A, x_2 : A_2 \vdash e_2 : C$

$\Gamma \vdash \mathsf{case}\ [e/x]e_0\ \mathsf{of}\ \mathsf{inl}\ x_1.\ [e/x]e_1 \mid \mathsf{inr}\ x_2.\ [e/x]e_2 : C$ $\hfill$ by the rule $+$E

$[e/x]\mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inl}\ x_1.\ e_1 \mid \mathsf{inr}\ x_2.\ e_2 = \mathsf{case}\ [e/x]e_0\ \mathsf{of}\ \mathsf{inl}\ x_1.\ [e/x]e_1 \mid \mathsf{inr}\ x_2.\ [e/x]e_2$

$\hfill$ from $x_1 \neq x$, $x_1 \notin FV(e)$, $x_2 \neq x$, $x_2 \notin FV(e)$

$\Gamma \vdash [e/x]\mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inl}\ x_1.\ e_1 \mid \mathsf{inr}\ x_2.\ e_2 : C$

Case $\dfrac{}{\Gamma, x : A \vdash () : \mathsf{unit}}$ Unit   where $e' = ()$ and $C = \mathsf{unit}$:

$\Gamma \vdash () : \mathsf{unit}$ $\hfill$ by the rule Unit

$\Gamma \vdash [e/x]() : \mathsf{unit}$ $\hfill$ from $[e/x]() = ()$

$\hfill\Box$

Lemma 5.6 (Inversion). *Suppose $\Gamma \vdash e : C$.*

If $e = (e_1, e_2)$, then $C = A_1 \times A_2$ and $\Gamma \vdash e_1 : A_1$ and $\Gamma \vdash e_2 : A_2$ for some types A_1 and A_2.

If $e = \mathsf{fst}\ e'$, then $\Gamma \vdash e' : C \times A_2$ for some type A_2.

If $e = \mathsf{snd}\ e'$, then $\Gamma \vdash e' : A_1 \times C$ for some type A_1.

If $e = ()$, then $C = \mathsf{unit}$.

If $e = \mathsf{inl}_{A_2}\ e'$, then $C = A_1 + A_2$ and $\Gamma \vdash e' : A_1$ for some type A_1.

If $e = \mathsf{inr}_{A_1}\ e'$, then $C = A_1 + A_2$ and $\Gamma \vdash e' : A_2$ for some type A_2.

If $e = \mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inl}\ x_1.\ e_1 \mid \mathsf{inr}\ x_2.\ e_2$, then $\Gamma \vdash e_0 : A_1 + A_2$, $\Gamma, x_1 : A_1 \vdash e_1 : C$, and $\Gamma, x_2 : A_2 \vdash e_2 : C$ for some types A_1 and A_2.

If $e = \mathsf{fix}\ x : A.\ e'$, then $C = A$ and $\Gamma, x : A \vdash e' : A$.

Proof. By the syntax-directedness of the type system. $\hfill\Box$

The proof of Theorem 5.3 extends the proof of Theorem 4.4.

Proof of Theorem 5.3. By rule induction on the judgment $e \mapsto e'$. We consider two cases that use Lemma 5.5. All other cases use a simple pattern (as in the case for the rule *Lam*): apply Lemma 5.6 to $\Gamma \vdash e : A$, apply induction hypothesis, and apply a typing rule to deduce $\Gamma \vdash e' : A$.

Case $\dfrac{}{\mathsf{case}\ \mathsf{inl}_C\ v\ \mathsf{of}\ \mathsf{inl}\ x_1.\ e_1 \mid \mathsf{inr}\ x_2.\ e_2 \mapsto [v/x_1]e_1}$ *Case′*

$\Gamma \vdash \mathsf{case}\ \mathsf{inl}_C\ v\ \mathsf{of}\ \mathsf{inl}\ x_1.\ e_1 \mid \mathsf{inr}\ x_2.\ e_2 : A$ $\hfill$ assumption

$\Gamma \vdash \mathsf{inl}_C\ v : A_1 + A_2$ and $\Gamma, x_1 : A_1 \vdash e_1 : A$ and $\Gamma, x_2 : A_2 \vdash e_2 : A$ for some types A_1 and A_2

$\hfill$ by Lemma 5.6

$\Gamma \vdash v : A_1$ and $C = A_2$ $\hfill$ by Lemma 5.6 on $\Gamma \vdash \mathsf{inl}_C\ v : A_1 + A_2$

$\Gamma \vdash [v/x_1]e_1 : A$ $\hfill$ by applying Lemma 5.5 to $\Gamma \vdash v : A_1$ and $\Gamma, x_1 : A_1 \vdash e_1 : A$

(The case for the rule $Case''$ is similar.)

Case $\dfrac{}{\text{fix } x\,{:}\,C.\,e_0 \mapsto [\text{fix } x\,{:}\,C.\,e_0/x]e_0}$ Fix

$\Gamma \vdash \text{fix } x\,{:}\,C.\,e_0 : A$	assumption
$A = C$ and $\Gamma, x : C \vdash e_0 : C$	by Lemma 5.6
$\Gamma \vdash \text{fix } x\,{:}\,C.\,e_0 : C$	from $\Gamma \vdash \text{fix } x\,{:}\,C.\,e_0 : A$ and $A = C$
$\Gamma \vdash [\text{fix } x\,{:}\,C.\,e_0/x]e_0 : C$	by applying Lemma 5.5 to $\Gamma \vdash \text{fix } x\,{:}\,C.\,e_0 : C$ and $\Gamma, x : C \vdash e_0 : C$

$\square$

September 30, 2011

Chapter 6

Mutable References

In the (typed or untyped) λ-calculus, or in "pure" functional languages, a variable is immutable in that once bound to a value as the result of a substitution, its contents never change. While it may appear to be too restrictive or even strange from the perspective of imperative programming, immutability of variables allows us to consider λ-abstractions as equivalent to mathematical functions whose meaning does not change, and thus makes programs more readable than in other programming paradigms. For example, the following λ-abstraction denotes a mathematical function taking a boolean value x and returning a logical conjunction of x and y where the value of y is determined at the time of evaluating the λ-abstraction:

$$\lambda x : \mathsf{bool}. \text{ if } x \text{ then } y \text{ else false}$$

Then the meaning of the λ-abstraction does not change throughout the evaluation; hence we only have to look at the λ-abstraction itself to learn what it means.

This chapter extends the simply typed λ-calculus with *mutable references*, or *references* for short, in the presence of which λ-abstractions no longer denote mathematical functions. We will introduce three new constructs for manipulating references; references may be thought of as another name for pointers familiar from imperative programming, and all these constructs find their counterparts in imperative languages:

- ref e creates or *allocates* a reference pointing to the value to which e evaluates.

- $!e$ obtains a reference by evaluating e, and then *dereferences* it, *i.e.*, retrieves the contents of it.

- $e := e'$ obtains a reference and a value by evaluating e and e', respectively, and then updates the contents of the reference with the value. That is, it *assigns* a new value to a reference.

It is easy to see that if a λ-abstraction is "contaminated" with references, it no longer denotes a mathematical function. For example, the meaning of the following λ-abstraction depends on the contents of a reference in y, and we cannot decide its meaning once and for all:

$$\lambda x : \mathsf{bool}. \text{ if } x \text{ then } !y \text{ else false}$$

In other words, each time we invoke the above λ-abstraction, we have to look up an environment (called a *store* or a *heap*) to obtain the value of $!y$. It is certainly either true or false, but we can never decide its meaning by looking only at the λ-abstraction itself. Hence it does not denote a mathematical function.

In general, we refer to those constructs that destroy the connection between λ-abstractions and mathematical functions, or the "purity" of the λ-calculus, as *computational effects*. References are the most common form of computational effects; other kinds of computational effects include exceptions, continuations, and input/output. A functional language with such features as references is often called an "impure" functional language.[1]

Below we extend the type system and the operational semantics of the simply typed λ-calculus to incorporate the three constructs for references. The development will be incremental in that if we remove the new constructs, the "impure" definition reverts to the "pure" definition of the simply typed

---

[1] SML is an impure functional language. Haskell is a pure functional language, although it comes with a few impure language constructs.

λ-calculus. An important implication is that immutability of variables is *not* affected by the new constructs — it is the contents of a reference that change; the contents of a variable never change!

6.1   Abstract syntax and type system

We augment the simply typed λ-calculus with three constructs for references; for the sake of simplicity, we do not consider base types P:

$$
\begin{array}{llll}
\text{type} & A & ::= & P \mid A \rightarrow A \mid \text{unit} \mid \text{ref } A \\
\text{expression} & e & ::= & x \mid \lambda x \colon A.\, e \mid e\, e \mid () \mid \text{ref } e \mid !e \mid e := e \\
\text{value} & v & ::= & \lambda x \colon A.\, e \mid ()
\end{array}
$$

A *reference type* ref A is the type for references pointing to values of type A, or equivalently, the type for references whose contents are values of type A. All the constructs for references use reference types, and behave as follows:

- ref e evaluates e to obtain a value v and then allocates a reference initialized with v; hence, if e has type A, then ref e has type ref A. It uses the same keyword ref as in reference types to maintain consistency with the syntax of SML.

- $!e$ evaluates e to obtain a reference and then retrieves its contents; hence, if e has type ref A, then $!e$ has type A.

- $e := e'$ evaluates e to obtain a reference and then updates its contents with a value obtained by evaluating e'. Since the result of updating the contents of a reference is computationally meaningless, $e := e'$ is assigned a unit type unit.

Thus we obtain the following typing rules for the constructs for references:

$$
\frac{\Gamma \vdash e : A}{\Gamma \vdash \text{ref } e : \text{ref } A}\ \text{Ref} \qquad
\frac{\Gamma \vdash e : \text{ref } A}{\Gamma \vdash !e : A}\ \text{Deref} \qquad
\frac{\Gamma \vdash e : \text{ref } A \quad \Gamma \vdash e' : A}{\Gamma \vdash e := e' : \text{unit}}\ \text{Assign}
$$

For proving type safety, we need to elaborate the typing judgment $\Gamma \vdash e : A$ (see Section 6.3), but for typechecking a given expression, the above typing rules suffice.

Before we give the reduction rules for the new constructs, let us consider a couple of examples exploiting references. Both examples use syntactic sugar let $x = e$ in e' for $(\lambda x \colon A.\, e')\, e$ for some type A. That is, let $x = e$ in e' first evaluates e and store the result in x; then it evaluates e'. We may think of the rule Let below as the typing rule for let $x = e$ in e':

$$
\frac{\Gamma \vdash e : A \quad \Gamma, x : A \vdash e' : B}{\Gamma \vdash \text{let } x = e \text{ in } e' : B}\ \text{Let}
$$

Both examples also assume common constructs for types bool and int (*e.g.*, if e then e_1 else e_2 as a conditional construct, $+$ for addition, $-$ for subtraction, $=$ for equality, and so on). We use a wildcard pattern _ for a variable not used in the body of a λ-abstraction.

The first example exploits references to simulate arrays of integers. We choose a functional representation of arrays by defining type iarray for arrays of integers as follows:

$$
\text{iarray} = \text{ref } (\text{int} \rightarrow \text{int})
$$

That is, we represent an array of integers as a function taking an index (of type int) and returning a corresponding element of the array. We need the following constructs for arrays:

- new : unit $\rightarrow$ iarray for creating a new array.
  new () returns a new array of indefinite size; all elements are initialized as 0.

- access : iarray $\rightarrow$ int $\rightarrow$ int for accessing an array.
  access $a\ i$ returns the i-th element of array a.

- update : iarray $\to$ int $\to$ int $\to$ unit for updating an array.
  update a i n updates the i-the element of array a with integer n.

Exercise 6.1. Implement new, access, and update.

We implement new and access according to the definition of type iarray:

$$\begin{aligned} \text{new} &= \lambda\_\!:\text{unit. ref } \lambda i\!:\!\text{int. } 0 \\ \text{access} &= \lambda a\!:\!\text{iarray. } \lambda i\!:\!\text{int. } (!a)\ i \end{aligned}$$

To implement update, we have to first retrieve a function of type int $\to$ int from a given array and then build a new function of type int $\to$ int:

$$\begin{aligned} \text{update} &= \lambda a\!:\!\text{iarray. } \lambda i\!:\!\text{int. } \lambda n\!:\!\text{int.} \\ &\quad \text{let } old = !a \text{ in} \\ &\quad a := \lambda j\!:\!\text{int. if } i = j \text{ then } n \text{ else } old\ j \end{aligned}$$

The following implementation of update has a correct type, but is wrong: a in the body does not point to the old array that exists *before* the update, but ends up pointing to the same array that is created *after* the update:

$$\begin{aligned} \text{update} &= \lambda a\!:\!\text{iarray. } \lambda i\!:\!\text{int. } \lambda n\!:\!\text{int.} \\ &\quad a := \lambda j\!:\!\text{int. if } i = j \text{ then } n \text{ else } (!a)\ j \end{aligned}$$

The wrong implementation of update illustrates that a reference a can be assigned a value that dereferences the same reference a. We can exploit such "self-references" to implement recursive functions without using the fixed point construct. The second example implements the following recursive function (written in the syntax of SML) which takes an integer n and returns the sum of integers from 0 to n:

$$\text{fun } f\ n\!:\!\text{int. if } n = 0 \text{ then } 0 \text{ else } n + f\ (n-1)$$

To implement the above recursive function, we first allocate a reference f initialized with a dummy function of type int $\to$ int. Then we assign the reference f a function which dereferences the same reference f when its argument is not equal to 0, thereby effecting a recursive call:

$$\begin{aligned} &\text{let } f = \text{ref } \lambda n\!:\!\text{int. } 0 \text{ in} \\ &\text{let } \_ = f := \lambda n\!:\!\text{int. if } n = 0 \text{ then } 0 \text{ else } n + (!f)\ (n-1) \text{ in} \\ &!f \end{aligned}$$

6.2 Operational semantics

The operational semantics for references needs a reduction judgment that departs from the previous reduction judgment $e \mapsto e'$ for the simply typed λ-calculus. To see why, consider how to reduce ref v for a value v. The operational semantics needs to allocate a reference initialized with v, but allocates it where? In fact, the abstract syntax given in the previous section does not even include expressions for results of allocating references. That is, what is the syntax for the result of reducing ref v? Thus we are led to extend both the abstract syntax to include values for references and the reduction judgment to record the contents of all references allocated during an evaluation.

We use a *store* ψ to record the contents of all references. A store is a mapping from *locations* to values, where a location l is a value for a reference, or simply another name for a reference. (We do not call l a "pointer" to emphasize that arithmetic operations may not be applied on l.) As we will see in the reduction rules below, a fresh location l is created only by reducing ref v, in which case the store is extended so as to map l to v. We define a store as an unordered collection of bindings of the form $l \mapsto v$:

$$\begin{array}{llll} \text{expression} & e & ::= & \cdots \mid l & \text{location} \\ \text{value} & v & ::= & \cdots \mid l & \text{location} \\ \text{store} & \psi & ::= & \cdot \mid \psi, l \mapsto v \end{array}$$

We write $dom(\psi)$ for the domain of ψ, *i.e.*, the set of locations mapped to certain values under ψ. Formally we define $dom(\psi)$ as follows:

$$\begin{aligned} dom(\cdot) &= \varnothing \\ dom(\psi, l \mapsto v) &= dom(\psi) \cup \{l\} \end{aligned}$$

We write $[l \mapsto v]\psi$ for the store obtained by updating the contents of l in ψ with v. Note that in order for $[l \mapsto v]\psi$ to be defined, l must be in $dom(\psi)$:

$$[l \mapsto v](\psi', l \mapsto v') \quad = \quad \psi', l \mapsto v$$

We write $\psi(l)$ for the value to which l is mapped under ψ; in order for $\psi(l)$ to be defined, l must be in $dom(\psi)$:

$$(\psi', l \mapsto v)(l) \quad = \quad v$$

Since the reduction of an expression may need to access or update a store, we use the following reduction judgment which carries a store along with an expression being reduced:

$$e \mid \psi \mapsto e' \mid \psi' \qquad \Leftrightarrow \qquad e \text{ with store } \psi \text{ reduces to } e' \text{ with store } \psi'$$

In the judgment $e \mid \psi \mapsto e' \mid \psi'$, we definitely have $e \neq e'$, but ψ and ψ' may be the same if the reduction of e does not make a change to ψ. The reduction rules are given as follows:

$$\frac{e_1 \mid \psi \mapsto e_1' \mid \psi'}{e_1 \, e_2 \mid \psi \mapsto e_1' \, e_2 \mid \psi'} \; Lam \qquad \frac{e_2 \mid \psi \mapsto e_2' \mid \psi'}{(\lambda x \colon A.\, e) \, e_2 \mid \psi \mapsto (\lambda x \colon A.\, e) \, e_2' \mid \psi'} \; Arg \qquad \frac{}{(\lambda x \colon A.\, e) \, v \mid \psi \mapsto [v/x]e \mid \psi} \; App$$

$$\frac{e \mid \psi \mapsto e' \mid \psi'}{\mathsf{ref}\, e \mid \psi \mapsto \mathsf{ref}\, e' \mid \psi'} \; Ref \qquad \frac{l \notin dom(\psi)}{\mathsf{ref}\, v \mid \psi \mapsto l \mid \psi, l \mapsto v} \; Ref'$$

$$\frac{e \mid \psi \mapsto e' \mid \psi'}{!e \mid \psi \mapsto !e' \mid \psi'} \; Deref \qquad \frac{\psi(l) = v}{!l \mid \psi \mapsto v \mid \psi} \; Deref'$$

$$\frac{e \mid \psi \mapsto e'' \mid \psi'}{e := e' \mid \psi \mapsto e'' := e' \mid \psi'} \; Assign \qquad \frac{e \mid \psi \mapsto e' \mid \psi'}{l := e \mid \psi \mapsto l := e' \mid \psi'} \; Assign' \qquad \frac{}{l := v \mid \psi \mapsto () \mid [l \mapsto v]\psi} \; Assign''$$

Note that locations are runtime values. That is, they are not part of the syntax for the source language; they are created only at runtime by reducing $\mathsf{ref}\, e$.

Here is the reduction sequence of the expression in Section 6.1 that builds a recursive function adding integers from 0 to n. It starts with an empty store and creates a fresh location l to store the recursive function. Recall that let $x = e$ in e' is syntactic sugar for $(\lambda x \colon A.\, e') \, e$ for some type A.

$$\begin{array}{l}
\text{let } f = \mathsf{ref}\, \lambda n \colon \mathsf{int}.\, 0 \text{ in} \\
\text{let } \_ = f := \lambda n \colon \mathsf{int}.\, \text{if } n = 0 \text{ then } 0 \text{ else } n + (!f)\,(n-1) \text{ in} \quad \mid \cdot \\
!f
\end{array}$$

$$\mapsto \quad \begin{array}{l}
\text{let } f = l \text{ in} \\
\text{let } \_ = f := \lambda n \colon \mathsf{int}.\, \text{if } n = 0 \text{ then } 0 \text{ else } n + (!f)\,(n-1) \text{ in} \quad \mid l \mapsto \lambda n \colon \mathsf{int}.\, 0 \\
!f
\end{array}$$

$$\mapsto \quad \begin{array}{l}
\text{let } \_ = l := \lambda n \colon \mathsf{int}.\, \text{if } n = 0 \text{ then } 0 \text{ else } n + (!l)\,(n-1) \text{ in} \\
!l
\end{array} \quad \mid l \mapsto \lambda n \colon \mathsf{int}.\, 0$$

$$\mapsto \quad \begin{array}{l}
\text{let } \_ = () \text{ in} \\
!l
\end{array} \quad \mid l \mapsto \lambda n \colon \mathsf{int}.\, \text{if } n = 0 \text{ then } 0 \text{ else } n + (!l)\,(n-1)$$

$$\mapsto \quad !l \quad \mid l \mapsto \lambda n \colon \mathsf{int}.\, \text{if } n = 0 \text{ then } 0 \text{ else } n + (!l)\,(n-1)$$

$$\mapsto \quad \lambda n \colon \mathsf{int}.\, \text{if } n = 0 \text{ then } 0 \text{ else } n + (!l)\,(n-1) \quad \mid l \mapsto \lambda n \colon \mathsf{int}.\, \text{if } n = 0 \text{ then } 0 \text{ else } n + (!l)\,(n-1)$$

6.3 Type safety

This section proves type safety of the simply typed λ-calculus extended with references. First of all, we have to extend the type system for locations which are valid (runtime) expressions but have not been given a typing rule. The development of the typing rule for locations gives rise to a new form of typing judgment, since deciding the type of a location requires information on a store, but the previous typing judgment $\Gamma \vdash e : A$ does not include information on a store. Then a proof of type safety rewrites the typing rules Ref, Deref, Assign in terms of the new typing judgment, although these typing rules in their current form suffice for the purpose of typechecking expressions containing no locations.

Let us begin with the following (blatantly wrong) typing rule for locations:

$$\frac{}{\Gamma \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc_1}$$

The rule $\mathsf{Loc_1}$ does not make sense: we wish to assign l a reference type ref A only if it is mapped to a value of type A under a certain store, but the rule does not even inspect the value to which l is mapped. The following typing rule uses a new form of typing judgment involving a store ψ, and assigns l a reference type ref A only if it is mapped to a value of type A under ψ:

$$\frac{\psi(l) = v \quad \Gamma \vdash v : A}{\Gamma \mid \psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc_2}$$

The rule $\mathsf{Loc_2}$ is definitely an improvement over the rule $\mathsf{Loc_1}$, but it is still inadequate: it uses an ordinary typing judgment $\Gamma \vdash v : A$ on the assumption that value v does not contain locations, but in general, any value in a store may contain locations. For example, it is perfectly fine to store a pair of locations in a store. Thus we are led to typecheck v in the premise of the rule $\mathsf{Loc_2}$ using the same typing judgment as in the conclusion:

$$\frac{\psi(l) = v \quad \Gamma \mid \psi \vdash v : A}{\Gamma \mid \psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc_3}$$

Unfortunately the rule $\mathsf{Loc_3}$ has a problem, too: if a location l is mapped to a value containing l under ψ, the derivation of $\Gamma \mid \psi \vdash l : \mathsf{ref}\ A$ never terminates because of the infinite chain of typing judgments all of which address the same location l:

$$\frac{\psi(l) = \cdots\, l\, \cdots \quad \dfrac{\dfrac{\vdots}{\Gamma \mid \psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc_3} \\ \vdots \\ \Gamma \mid \psi \vdash \cdots\, l\, \cdots : A}{}}{\Gamma \mid \psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc_3}$$

For example, it is impossible to determine the type of a location l that is mapped to a λ-abstraction $\lambda n : \mathsf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ n + (!l)\ (n-1)$ containing l itself. (Section 6.2 gives an example of creating such a binding for l.)

We fix the rule $\mathsf{Loc_3}$ by analyzing a store only once, rather than each time a location is encountered during typechecking. To this end, we introduce a *store typing context* which records types of all values in a store:

$$\text{store typing context} \quad \Psi \quad ::= \quad \cdot \mid \Psi, l \mapsto A$$

The idea is that if a store maps location l_i to value v_i of type A_i, it is given a store typing context mapping l_i to A_i (for $i = 1, \cdots, n$). Given a store typing context Ψ corresponding to a store ψ, then, we use Ψ, instead of ψ, for typechecking expressions:

$$\Gamma \mid \Psi \vdash e : A \quad \Leftrightarrow \quad \textit{expression e has type A under typing context } \Gamma \textit{ and store typing context } \Psi$$

We write $dom(\Psi)$ for the domain of Ψ:

$$\begin{aligned} dom(\cdot) &= \varnothing \\ dom(\Psi, l \mapsto A) &= dom(\Psi) \cup \{l\} \end{aligned}$$

We write $\Psi(l)$ for the type to which l is mapped under Ψ; in order for $\Psi(l)$ to be defined, l must be in $dom(\Psi)$:

$$(\Psi', l \mapsto A)(l) \quad = \quad A$$

Now the typing rule for locations looks up a store typing context included in the typing judgment:

$$\frac{\Psi(l) = A}{\Gamma \mid \Psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc}$$

All other typing rules are obtained by replacing $\Gamma \vdash e : A$ by $\Gamma \mid \Psi \vdash e : A$ in the previous typing rules:

$$\frac{x : A \in \Gamma}{\Gamma \mid \Psi \vdash x : A}\ \mathsf{Var} \quad \frac{\Gamma, x : A \mid \Psi \vdash e : B}{\Gamma \mid \Psi \vdash \lambda x{:}A.\,e : A \to B}\ {\to}\mathsf{I} \quad \frac{\Gamma \mid \Psi \vdash e : A \to B \quad \Gamma \mid \Psi \vdash e' : A}{\Gamma \mid \Psi \vdash e\,e' : B}\ {\to}\mathsf{E}$$

$$\frac{}{\Gamma \mid \Psi \vdash () : \mathsf{unit}}\ \mathsf{Unit}$$

$$\frac{\Gamma \mid \Psi \vdash e : A}{\Gamma \mid \Psi \vdash \mathsf{ref}\ e : \mathsf{ref}\ A}\ \mathsf{Ref} \quad \frac{\Gamma \mid \Psi \vdash e : \mathsf{ref}\ A}{\Gamma \mid \Psi \vdash !e : A}\ \mathsf{Deref} \quad \frac{\Gamma \mid \Psi \vdash e : \mathsf{ref}\ A \quad \Gamma \mid \Psi \vdash e' : A}{\Gamma \mid \Psi \vdash e := e' : \mathsf{unit}}\ \mathsf{Assign}$$

The remaining question is: how do we decide a store typing context Ψ corresponding to a store ψ? We use a new judgment

$$\psi :: \Psi$$

to mean that Ψ corresponds to ψ, or simply, ψ is *well-typed* with Ψ. Then the goal is to give an inference rule for the judgment $\psi :: \Psi$.

Suppose that ψ maps l_i to v_i for $i = 1, \cdots, n$. Loosely speaking, Ψ maps l_i to A_i if v_i has type A_i. Then how do we verify that v_i has type A_i? Since we are in the process of deciding Ψ (which is unknown yet), $\cdot \mid \cdot \vdash v_i : A_i$ may appear to be the right judgment. The judgment is, however, inadequate because v_i itself may contain locations pointing to other values in the same store ψ. Therefore we have to typecheck *all locations simultaneously using the same typing context currently being decided*:

$$\frac{dom(\Psi) = dom(\psi) \quad \cdot \mid \Psi \vdash \psi(l) : \Psi(l)\ \text{ for every } l \in dom(\psi)}{\psi :: \Psi}\ \mathsf{Store}$$

Note that the use of an empty typing context in the premise implies that every value in a well-typed store is closed.

Type safety is is stated as follows:

Theorem 6.2 (Progress). *Suppose that expression e satisfies $\cdot \mid \Psi \vdash e : A$ for some store typing context Ψ and type A. Then either:*

(1) e is a value, or

(2) for any store ψ such that $\psi :: \Psi$, there exist some expression e' and store ψ' such that $e \mid \psi \mapsto e' \mid \psi'$.

Theorem 6.3 (Type preservation).

Suppose $\left\{ \begin{array}{l} \Gamma \mid \Psi \vdash e : A \\ \psi :: \Psi \\ e \mid \psi \mapsto e' \mid \psi' \end{array} \right.$ *. Then there exists a store typing context Ψ' such that* $\left\{ \begin{array}{l} \Gamma \mid \Psi' \vdash e' : A \\ \Psi \subset \Psi' \\ \psi' :: \Psi' \end{array} \right.$ *.*

In Theorem 6.3, ψ' may extend ψ by the rule Ref', in which case Ψ' also extends Ψ, *i.e.*, $\Psi' \supset \Psi$ and $\Psi' \neq \Psi$. Note also that ψ' is not always a superset of ψ (*i.e.*, $\psi' \not\supset \psi$) because the rule $Assign''$ updates ψ without extending it. Even in this case, however, $\Psi' \supset \Psi$ still holds because Ψ' and Ψ are the same.

We use type safety to show that a well-typed expression cannot go wrong. Suppose that we are reducing a closed well-typed expression e with a well-typed store ψ. That is, $\psi :: \Psi$ and $\cdot \mid \Psi \vdash e : A$ hold for some store typing context Ψ and type A. If e is already a value, the reduction has been finished. Otherwise Theorem 6.2 guarantees that there exist some expression e' and store ψ' such that $e \mid \psi \mapsto e' \mid \psi'$. By Theorem 6.3, then, there exists a store typing context Ψ' such that $\cdot \mid \Psi' \vdash e' : A$ and $\psi' :: \Psi'$. That is,

e' is a close well-typed expression and ψ' is a well-typed store (with Ψ'). Therefore the reduction of e with ψ cannot go wrong!

Chapter 7

Typechecking

So far, our interpretation of the typing judgment $\Gamma \vdash e : A$ has been *declarative* in the sense that given a triple of Γ, e, and A, the judgment answers either "yes" (meaning that e has type A under Γ) or "no" (meaning that e does not have type A under Γ). While the declarative interpretation is enough for proving type safety of the simply typed λ-calculus, it does not lend itself well to an implementation of the type system, which takes a pair of Γ and e and decides a type for e under Γ, if one exists. That is, an implementation of the type system requires not a declarative interpretation but an *algorithmic* interpretation of the typing judgment $\Gamma \vdash e : A$ such that given Γ and e as input, the interpretation produces A as output.

This chapter discusses two implementations of the type system. The first employs an algorithmic interpretation of the typing judgment, and is purely synthetic in that given Γ and e, it synthesizes a type A such that $\Gamma \vdash e : A$. The second mixes an algorithmic interpretation with a declarative interpretation, and achieves what is called *bidirectional typechecking*. It is both synthetic and analytic in that depending on the form of a given expression e, it requires either only Γ to synthesize a type A such that $\Gamma \vdash e : A$, or both Γ and A to confirm that $\Gamma \vdash e : A$ holds.

7.1 Purely synthetic typechecking

Let us consider a direct implementation of the type system, or equivalently the judgment $\Gamma \vdash e : A$. We introduce a function typing with the following invariant:

$$
\begin{array}{llll}
\text{typing } (\Gamma, e, A) & = & \text{okay} & \text{if } \Gamma \vdash e : A \text{ holds.} \\
\text{typing } (\Gamma, e, A) & = & \text{fail} & \text{if } \Gamma \vdash e : A \text{ does not hold.}
\end{array}
$$

Since Γ, e, and A are all given as input, we only have to translate each typing rule in the direction from the conclusion to the premise(s) (*i.e.*, bottom-up), as illustrated in the pseudocode below:

$$
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ Var} \quad \Leftrightarrow \quad
\begin{array}{l}
\text{typing } (\Gamma, x, A) = \\
\quad \text{if } x : A \in \Gamma \text{ then okay else fail}
\end{array}
$$

$$
\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x{:}A.\, e : A \to B} \to\!\text{I} \quad \Leftrightarrow \quad
\begin{array}{l}
\text{typing } (\Gamma, \lambda x{:}A.\, e, A \to B) = \\
\quad \text{typing } (\Gamma', e, B) \text{ where } \Gamma' = \Gamma, x : A
\end{array}
$$

It is not obvious, however, how to translate the rule $\to\!\text{E}$ because both premises require a type A which does not appear in the conclusion:

$$
\frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\, e' : B} \to\!\text{E} \quad \Leftrightarrow \quad
\begin{array}{l}
\text{typing } (\Gamma, e\, e', B) = \\
\quad \text{if typing } (\Gamma, e, A \to B) = \text{okay} \\
\quad\quad \text{andalso typing } (\Gamma, e', A) = \text{okay} \\
\quad \text{then okay else fail} \\
\quad \text{where } A = ?
\end{array}
$$

Therefore, in order to return okay, typing $(\Gamma, e\, e', B)$ must "guess" a type A such that both typing $(\Gamma, e, A \to B)$ and typing (Γ, e', A) return okay. The problem of guessing such a type A from e and e' involves the problem of deciding the type of a given expression (*e.g.*, deciding type A of expression e'). Thus we need to

be able to decide the type of a given expression anyway, and are led to interpret the typing judgment $\Gamma \vdash e : A$ algorithmically so that given Γ and e as input, an algorithmic interpretation of the judgment produces A as output.

We introduce a new judgment $\Gamma \vdash e \triangleright A$, called an *algorithmic typing judgment*, to express the algorithmic interpretation of the typing judgment $\Gamma \vdash e : A$:

$$\Gamma \vdash e \triangleright A \quad \Leftrightarrow \quad \text{under typing context } \Gamma, \text{ the type of expression } e \text{ is inferred as } A$$

That is, an algorithmic typing judgment $\Gamma \vdash e \triangleright A$ synthesizes type A (output) for expression e (input) under typing context Γ (input). Algorithmic typing rules (*i.e.*, inference rules for algorithmic typing judgments) are as given follows:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \triangleright A} \; \text{Var}_a \quad \frac{\Gamma, x : A \vdash e \triangleright B}{\Gamma \vdash \lambda x : A.\, e \triangleright A \to B} \; \to\!\text{I}_a \quad \frac{\Gamma \vdash e \triangleright A \to B \quad \Gamma \vdash e' \triangleright C \quad A = C}{\Gamma \vdash e\, e' \triangleright B} \; \to\!\text{E}_a$$

$$\frac{}{\Gamma \vdash \text{true} \triangleright \text{bool}} \; \text{True}_a \quad \frac{}{\Gamma \vdash \text{false} \triangleright \text{bool}} \; \text{False}_a \quad \frac{\Gamma \vdash e \triangleright \text{bool} \quad \Gamma \vdash e_1 \triangleright A_1 \quad \Gamma \vdash e_2 \triangleright A_2 \quad A_1 = A_2}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \triangleright A_1} \; \text{If}_a$$

Note that in the rule $\to\!\text{E}_a$, we may not write the second premise as $\Gamma \vdash e' \triangleright A$ (and remove the third premise) because type C to be inferred from Γ and e' is unknown in general and must be explicitly compared with type A as is done in the third premise. (Similarly for types A_1 and A_2 in the rule If_a.) A typechecking algorithm based on the algorithmic typing judgment $\Gamma \vdash e \triangleright A$ is said to be purely synthetic.

The equivalence between two judgments $\Gamma \vdash e \triangleright A$ and $\Gamma \vdash e : A$ is stated in Theorem 7.3, whose proof uses Lemmas 7.1 and 7.2. Lemma 7.1 proves soundness of $\Gamma \vdash e \triangleright A$ in the sense that if an algorithmic typing judgment infers type A for expression e under typing context Γ, then A is indeed the type for e under Γ. In other words, if an algorithmic typing judgment gives an answer, it always gives a correct answer and is thus "sound." Lemma 7.2 proves completeness of $\Gamma \vdash e \triangleright A$ in the sense that for any well-typed expression e under typing context Γ, there exists an algorithmic typing judgment inferring its type. In other words, an algorithmic typing judgment covers all possible cases of well-typed expressions and is thus "complete."

Lemma 7.1 (soundness). *If $\Gamma \vdash e \triangleright A$, then $\Gamma \vdash e : A$.*

Proof. By rule induction on the judgment $\Gamma \vdash e \triangleright A$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Lemma 7.2 (completeness). *If $\Gamma \vdash e : A$, then $\Gamma \vdash e \triangleright A$.*

Proof. By rule induction on the judgment $\Gamma \vdash e : A$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Theorem 7.3. $\Gamma \vdash e : A$ *if and only if* $\Gamma \vdash e \triangleright A$.

Proof. Follows from Lemmas 7.1 and 7.2. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

7.2 Bidirectional typechecking

In the simply typed λ-calculus, every variable in a λ-abstraction is annotated with its type (*e.g.*, $\lambda x : A.\, e$). While it is always good to know the type of a variable for the purpose of typechecking, a typechecking algorithm may not need the type annotation of every variable which sometimes reduces code readability. As an example, consider the following expression which has type bool:

$$(\lambda f : \text{bool} \to \text{bool}.\, f \text{ true})\, \lambda x : \text{bool}.\, x$$

The type of the first subexpression $\lambda f : \text{bool} \to \text{bool}.\, f \text{ true}$ is $(\text{bool} \to \text{bool}) \to \text{bool}$, so the whole expression typechecks only if the second subexpression $\lambda x : \text{bool}.\, x$ has type $\text{bool} \to \text{bool}$ (according to the rule $\to\!\text{E}$). Then the type annotation for variable x becomes redundant because it must have type bool anyway if $\lambda x : \text{bool}.\, x$ is to have type $\text{bool} \to \text{bool}$. This example illustrates that not every variable in a well-typed expression needs to be annotated with its type.

A bidirectional typechecking algorithm takes a different approach by allowing λ-abstractions with no type annotations (*i.e.*, $\lambda x.\,e$ as in the untyped λ-calculus), but also requiring certain expressions to be explicitly annotated with their types. Thus bidirectional typechecking assumes a modified definition of abstract syntax:

$$\text{expression} \quad e \quad ::= \quad x \mid \lambda x.\,e \mid e\,e \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid (e : A)$$

A λ-abstraction $\lambda x.\,e$ does not annotate its formal argument with a type. (It is okay to permit $\lambda x\!:\!A.\,e$ in addition to $\lambda x.\,e$, but it does not expand the set of well-typed expressions under bidirectional typechecking.) $(e : A)$ explicitly annotates expression e with type A, and plays the role of variable x bound in a λ-abstraction $\lambda x\!:\!A.\,e$. Specifically it is $(e : A)$ that feeds type information into a bidirectional typechecking algorithm whereas it is $\lambda x\!:\!A.\,e$ that feeds type information into an ordinary typechecking algorithm.

A bidirectional typechecking algorithm proceeds by alternating between an *analysis* phase, in which it "analyzes" a given expression to verify that it indeed has a given type, and a *synthesis* phase, in which it "synthesizes" the type of a given expression. We use two new judgments for the two phases of bidirectional typechecking:

- $\Gamma \vdash e \Uparrow A$ means that we are checking expression e against type A under typing context Γ. That is, Γ, e, and A are all given and we are checking if $\Gamma \vdash e : A$ holds. $\Gamma \vdash e \Uparrow A$ corresponds to a declarative interpretation of the typing judgment $\Gamma \vdash e : A$.

- $\Gamma \vdash e \Downarrow A$ means that we have synthesized type A from expression e under typing context Γ. That is, only Γ and e are given and we have synthesized type A such that $\Gamma \vdash e : A$ holds. $\Gamma \vdash e \Downarrow A$ corresponds to an algorithmic interpretation of the typing judgment $\Gamma \vdash e : A$, and is stronger (*i.e.*, more difficult to prove) than $\Gamma \vdash e \Uparrow A$.

Now we have to decide which of $\Gamma \vdash e \Uparrow A$ and $\Gamma \vdash e \Downarrow A$ is applicable to a given expression e. Let us consider a λ-abstraction $\lambda x.\,e$ first:

$$\frac{\cdots}{\Gamma \vdash \lambda x.\,e \Downarrow A \to B} \,{\to}\mathsf{I_b} \qquad or \qquad \frac{\cdots}{\Gamma \vdash \lambda x.\,e \Uparrow A \to B} \,{\to}\mathsf{I_b}$$

Intuitively we cannot hope to synthesize type $A \to B$ from $\lambda x.\,e$ because the type of x is unknown in general. For example, e may not use x at all, in which case it is literally impossible to infer the type of x! Therefore we have to check $\lambda x.\,e$ against a type $A \to B$ to be given in advance:

$$\frac{\Gamma, x : A \vdash e \Uparrow B}{\Gamma \vdash \lambda x.\,e \Uparrow A \to B} \,{\to}\mathsf{I_b}$$

Next let us consider an application $e\,e'$:

$$\frac{\cdots}{\Gamma \vdash e\,e' \Downarrow B} \,{\to}\mathsf{E_b} \qquad or \qquad \frac{\cdots}{\Gamma \vdash e\,e' \Uparrow B} \,{\to}\mathsf{E_b}$$

Intuitively it is pointless to check $e\,e'$ against type B, since we have to synthesize type $A \to B$ for e anyway. With type $A \to B$ for e, then, we automatically synthesize type B for $e\,e'$ as well, and the problem of checking $e\,e'$ against type B becomes obsolete because it is easier than the problem of synthesizing type B for $e\,e'$. Therefore we synthesize type B from $e\,e'$ by first synthesizing type $A \to B$ from e and then verifying that e' has type A:

$$\frac{\Gamma \vdash e \Downarrow A \to B \quad \Gamma \vdash e' \Uparrow A}{\Gamma \vdash e\,e' \Downarrow B} \,{\to}\mathsf{E_b}$$

For a variable, we can always synthesize its type by looking up a typing context:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Downarrow A} \,\mathsf{Var_b}$$

Then how can we relate the two judgments $\Gamma \vdash e \Uparrow A$ and $\Gamma \vdash e \Downarrow A$? Since $\Gamma \vdash e \Downarrow A$ is stronger than $\Gamma \vdash e \Uparrow A$, the following rule makes sense regardless of the form of expression e:

$$\frac{\Gamma \vdash e \Downarrow A}{\Gamma \vdash e \Uparrow A} \,{\Downarrow}{\Uparrow}\mathsf{b}$$

The opposite direction does not make sense, but by annotating e with its intended type A, we can relate the two judgments in the opposite direction:

$$\frac{\Gamma \vdash e \Uparrow A}{\Gamma \vdash (e : A) \Downarrow A} \;\; \Uparrow\Downarrow_\mathsf{b}$$

The rule $\Uparrow\Downarrow_\mathsf{b}$ says that if expression e is annotated with type A, we may take A as the type of e without having to guess, or "synthesize," it, but only after verifying that e indeed has type A.

Now we can classify expressions into two kinds: *intro(duction)* expressions I and *elim(ination)* expressions E. We always check an intro expression I against some type A; hence $\Gamma \vdash I \Uparrow A$ makes sense, but $\Gamma \vdash I \Downarrow A$ is not allowed. For an elim expression E, we can either try to synthesize its type A or check it against some type A; hence both $\Gamma \vdash E \Downarrow A$ and $\Gamma \vdash E \Uparrow A$ make sense. The mutual definition of intro and elim expressions is specified by the rules for bidirectional typechecking:

$$
\begin{array}{lcll}
\text{intro expression} & I & ::= & \lambda x.\,I \mid E \\
\text{elim expression} & E & ::= & x \mid E\,I \mid (I : A)
\end{array}
$$

As you might have guessed, an expression is an intro expression if its corresponding typing rule is an introduction rule. For example, $\lambda x.\,e$ is an intro expression because its corresponding typing rule is the $\to$ introduction rule $\to$I. Likewise an expression is an elim expression if its corresponding typing rule is an elimination rule. For example, $e\,e'$ is an elim expression because its corresponding typing rule is the $\to$ elimination rule $\to$E, although it requires further consideration to see why e is an elim expression and e' is an intro expression.

For your reference, we give the complete definition of intro and elim expressions by including remaining constructs of the simply typed λ-calculus. As in λ-abstractions, we do not need type annotations in left injections, right injections, abort expression, and the fixed point construct. We use a case expression as an intro expression instead of an elim expression. We use an abort expression as an intro expression because it is a special case of a case expression. Figure 7.1 shows the definition of intro and elim expressions as well as all typing rules for bidirectional typechecking.

7.3 Exercises

Exercise 7.4. Give algorithmic typing rules for the extended simply typed λ-calculus in Figure 5.1.

Exercise 7.5. Give typing rules for true, false, and if e then e_1 else e_2 under bidirectional typechecking.

Exercise 7.6. $(\lambda x.\,x)\,()$ has type unit. This expression, however, does not typecheck against unit under bidirectional typechecking. Write as much of a derivation $\cdot \vdash (\lambda x.\,x)\,() \Uparrow$ unit as you can, and indicate with an asterisk (*) where the derivation gets stuck.

Exercise 7.7. Annotate some intro expression in $(\lambda x.\,x)\,()$ with a type (*i.e.*, convert an intro expression I into an elim expression $(I : A)$), and typecheck the whole expression using bidirectional typechecking.

intro expression	I	$::=$	$\lambda x.\,I$	$\rightarrow\mathsf{I_b}$
			$\mid (I, I)$	$\times\mathsf{I_b}$
			$\mid \mathsf{inl}\ I$	$+\mathsf{I_{Lb}}$
			$\mid \mathsf{inr}\ I$	$+\mathsf{I_{Rb}}$
			$\mid \mathsf{case}\ E\ \mathsf{of}\ \mathsf{inl}\ x.\,I \mid \mathsf{inr}\ x.\,I$	$+\mathsf{E_b}$
			$\mid ()$	$\mathsf{Unit_b}$
			$\mid \mathsf{abort}\ E$	$\mathsf{Abort_b}$
			$\mid \mathsf{fix}\ x.\,I$	$\mathsf{Fix_b}$
			$\mid E$	$\Downarrow\Uparrow_\mathsf{b}$
elim expression	E	$::=$	x	$\mathsf{Var_b}$
			$\mid E\ I$	$\rightarrow\mathsf{E_b}$
			$\mid \mathsf{fst}\ E$	$\times\mathsf{E_{1b}}$
			$\mid \mathsf{snd}\ E$	$\times\mathsf{E_{2b}}$
			$\mid (I : A)$	$\Uparrow\Downarrow_\mathsf{b}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Downarrow A}\ \mathsf{Var_b} \qquad \frac{\Gamma, x : A \vdash I \Uparrow B}{\Gamma \vdash \lambda x.\,I \Uparrow A \rightarrow B}\ \rightarrow\mathsf{I_b} \qquad \frac{\Gamma \vdash E \Downarrow A \rightarrow B \quad \Gamma \vdash I \Uparrow A}{\Gamma \vdash E\ I \Downarrow B}\ \rightarrow\mathsf{E_b}$$

$$\frac{\Gamma \vdash I_1 \Uparrow A_1 \quad \Gamma \vdash I_2 \Uparrow A_2}{\Gamma \vdash (I_1, I_2) \Uparrow A_1 \times A_2}\ \times\mathsf{I_b} \qquad \frac{\Gamma \vdash E \Downarrow A_1 \times A_2}{\Gamma \vdash \mathsf{fst}\ E \Downarrow A_1}\ \times\mathsf{E_{1b}} \qquad \frac{\Gamma \vdash E \Downarrow A_1 \times A_2}{\Gamma \vdash \mathsf{snd}\ E \Downarrow A_2}\ \times\mathsf{E_{2b}}$$

$$\frac{\Gamma \vdash I \Uparrow A_1}{\Gamma \vdash \mathsf{inl}\ I \Uparrow A_1 + A_2}\ +\mathsf{I_{Lb}} \qquad \frac{\Gamma \vdash I \Uparrow A_2}{\Gamma \vdash \mathsf{inr}\ I \Uparrow A_1 + A_2}\ +\mathsf{I_{Rb}}$$

$$\frac{\Gamma \vdash E \Downarrow A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash I_1 \Uparrow C \quad \Gamma, x_2 : A_2 \vdash I_2 \Uparrow C}{\Gamma \vdash \mathsf{case}\ E\ \mathsf{of}\ \mathsf{inl}\ x_1.\,I_1 \mid \mathsf{inr}\ x_2.\,I_2 \Uparrow C}\ +\mathsf{E_b}$$

$$\frac{}{\Gamma \vdash () \Uparrow \mathsf{unit}}\ \mathsf{Unit_b} \qquad \frac{\Gamma \vdash E \Downarrow \mathsf{void}}{\Gamma \vdash \mathsf{abort}\ E \Uparrow C}\ \mathsf{Abort_b} \qquad \frac{\Gamma, x : A \vdash I \Uparrow A}{\Gamma \vdash \mathsf{fix}\ x.\,I \Uparrow A}\ \mathsf{Fix_b}$$

$$\frac{\Gamma \vdash E \Downarrow A}{\Gamma \vdash E \Uparrow A}\ \Downarrow\Uparrow_\mathsf{b} \qquad \frac{\Gamma \vdash I \Uparrow A}{\Gamma \vdash (I : A) \Downarrow A}\ \Uparrow\Downarrow_\mathsf{b}$$

Figure 7.1: Definition of intro and elim expressions with their typing rules

Chapter 8

Evaluation contexts

This chapter presents an alternative formulation of the operational semantics for the simply typed λ-calculus. Compared with the operational semantics in Chapter 4, the new formulation is less complex, yet better reflects reductions of expressions in a concrete implementation. The new formulation is a basis for an *abstract machine* for the simply typed λ-calculus, which, like the Java virtual machine, is capable of running a program independently of the underlying hardware platform.

8.1 Evaluation contexts

Consider the simply typed λ-calculus given in Chapter 4:

$$
\begin{array}{rrcl}
\text{type} & A & ::= & P \mid A \to A \\
\text{base type} & P & ::= & \mathsf{bool} \\
\text{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\,e \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
\text{value} & v & ::= & \lambda x\!:\!A.\,e \mid \mathsf{true} \mid \mathsf{false}
\end{array}
$$

A reduction judgment $e \mapsto e'$ for the call-by-value strategy is defined inductively by the following reduction rules:

$$
\frac{e_1 \mapsto e_1'}{e_1\,e_2 \mapsto e_1'\,e_2}\ Lam \qquad
\frac{e_2 \mapsto e_2'}{(\lambda x\!:\!A.\,e)\,e_2 \mapsto (\lambda x\!:\!A.\,e)\,e_2'}\ Arg \qquad
\frac{}{(\lambda x\!:\!A.\,e)\,v \mapsto [v/x]e}\ App
$$

$$
\frac{e \mapsto e'}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}\ If
$$

$$
\frac{}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_1}\ If_{\mathsf{true}} \qquad
\frac{}{\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_2}\ If_{\mathsf{false}}
$$

Since only the rules App, If_{true}, and If_{false} have no premise, every derivation tree for a reduction judgment $e \mapsto e'$ must end with an application of one of these rules:

$$
\frac{}{(\lambda x\!:\!A.\,e'')\,v \mapsto [v/x]e''}\ App
$$
$$
\vdots
$$
$$
e \mapsto e'
$$

$$
\frac{}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_1}\ If_{\mathsf{true}} \qquad\qquad
\frac{}{\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_2}\ If_{\mathsf{false}}
$$
$$
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots
$$
$$
e \mapsto e' \qquad\qquad\qquad\qquad\qquad\qquad\qquad e \mapsto e'
$$

89

Thus the reduction of an expression e amounts to locating an appropriate subexpression $(\lambda x \colon A.\, e'')\, v$, if true then e_1 else e_2, or if false then e_1 else e_2 of e and applying a corresponding reduction rule.

As an example, let us reduce the following expression:

$$e \quad = \quad (\text{if } \underline{(\lambda x \colon A.\, e'')\, v} \text{ then } e_1 \text{ else } e_2)\, e'$$

The reduction of e cannot proceed without first reducing the underlined subexpression $(\lambda x \colon A.\, e'')\, v$ by the rule App, as shown in the following derivation tree:

$$\cfrac{\cfrac{\cfrac{}{(\lambda x \colon A.\, e'')\, v \mapsto [v/x]e''} \; App}{\text{if } (\lambda x \colon A.\, e'')\, v \text{ then } e_1 \text{ else } e_2 \mapsto \cdots} \; If}{(\text{if } (\lambda x \colon A.\, e'')\, v \text{ then } e_1 \text{ else } e_2)\, e' \mapsto \cdots} \; Lam$$

Then we may think of e as consisting of two parts: a subexpression, or a redex, $(\lambda x \colon A.\, e'')\, v$ which actually reduces to another expression $[v/x]e''$ by the rule App, and the rest which remains intact during the reduction. Note that the second part is *not* an expression because it is obtained by erasing the redex from e. We write the second part as $(\text{if } \square \text{ then } e_1 \text{ else } e_2)\, e'$ where the hole $\square$ indicates the position of the redex.

We refer to an expression with a hole in it, such as $(\text{if } \square \text{ then } e_1 \text{ else } e_2)\, e'$, as an *evaluation context*. The hole indicates the position of the redex (to be reduced by one of the rules App, If_{true}, and If_{false}) for the next step. Note that we may not use the rule Lam, Arg, or If to reduce the redex, since none of these rules reduces the whole redex in a single step.

Since the hole in an evaluation context indicates the position of a redex, every expression is decomposed into a *unique* evaluation context and a *unique* redex under a particular reduction strategy. For the same reason, not every expression with a hole in it is a valid evaluation context. For example, $(e_1\, e_2)\, \square$ is not a valid evaluation context under the call-by-value strategy because given an expression $(e_1\, e_2)\, e'$, we have to reduce $e_1\, e_2$ before we reduce e'. These two observations show that a particular reduction strategy specifies a *unique* inductive definition of evaluation contexts. The call-by-value strategy results in the following definition:

$$\text{evaluation context} \qquad \kappa \quad ::= \quad \square \mid \kappa\, e \mid (\lambda x \colon A.\, e)\, \kappa \mid \text{if } \kappa \text{ then } e \text{ else } e$$

$\kappa\, e$ is an evaluation context for $e'\, e$ where e' needs to be further reduced; $(\lambda x \colon A.\, e)\, \kappa$ is an evaluation context for $(\lambda x \colon A.\, e)\, e'$ where e' needs to be further reduced. Similarly if κ then e_1 else e_2 is an evaluation context for if e' then e_1 else e_2 where e' needs to be further reduced.

Let us write $\kappa[\![e]\!]$ for an expression obtained by filling the hole in κ with e. Here are a few examples:

$$
\begin{aligned}
\square[\![(\lambda x \colon A.\, e'')\, v]\!] &= (\lambda x \colon A.\, e'')\, v \\
(\text{if } \square \text{ then } e_1 \text{ else } e_2)[\![(\lambda x \colon A.\, e'')\, v]\!] &= (\text{if } (\lambda x \colon A.\, e'')\, v \text{ then } e_1 \text{ else } e_2) \\
((\text{if } \square \text{ then } e_1 \text{ else } e_2)\, e')[\![(\lambda x \colon A.\, e'')\, v]\!] &= (\text{if } (\lambda x \colon A.\, e'')\, v \text{ then } e_1 \text{ else } e_2)\, e'
\end{aligned}
$$

A formal definition of $\kappa[\![e]\!]$ is given as follows:

$$
\begin{aligned}
\square[\![e]\!] &= e \\
(\kappa\, e')[\![e]\!] &= \kappa[\![e]\!]\, e' \\
((\lambda x \colon A.\, e')\, \kappa)[\![e]\!] &= (\lambda x \colon A.\, e')\, \kappa[\![e]\!] \\
(\text{if } \kappa \text{ then } e_1 \text{ else } e_2)[\![e]\!] &= \text{if } \kappa[\![e]\!] \text{ then } e_1 \text{ else } e_2
\end{aligned}
$$

Now consider an expression which is known to reduce to another expression. We can write it as $\kappa[\![e]\!]$ for a unique evaluation context κ and a unique redex e. Since $\kappa[\![e]\!]$ is known to reduce to another expression, e must also reduce to another expression e'. We write $e \mapsto_\beta e'$ to indicate that the reduction of e to e' uses the rule App, If_{true}, or If_{false}. Then the following reduction rule alone is enough to completely specify a reduction strategy because the order of reduction is implicitly determined by the definition of evaluation contexts:

$$\cfrac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]} \; Red_\beta$$

$$\text{evaluation context} \qquad \kappa \quad ::= \quad \Box \mid \kappa\, e \mid (\lambda x\!:\!A.\, e)\, \kappa \mid \text{if } \kappa \text{ then } e \text{ else } e$$

$$
\begin{aligned}
(\lambda x\!:\!A.\, e)\, v \quad &\mapsto_\beta \quad [v/x]e \\
\text{if true then } e_1 \text{ else } e_2 \quad &\mapsto_\beta \quad e_1 \\
\text{if false then } e_1 \text{ else } e_2 \quad &\mapsto_\beta \quad e_2
\end{aligned}
$$

$$\dfrac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]} \; Red_\beta$$

Figure 8.1: Call-by-value operational semantics using evaluation contexts

$$\text{evaluation context} \qquad \kappa \quad ::= \quad \Box \mid \kappa\, e \mid \text{if } \kappa \text{ then } e \text{ else } e$$

$$
\begin{aligned}
(\lambda x\!:\!A.\, e)\, e' \quad &\mapsto_\beta \quad [e'/x]e \\
\text{if true then } e_1 \text{ else } e_2 \quad &\mapsto_\beta \quad e_1 \\
\text{if false then } e_1 \text{ else } e_2 \quad &\mapsto_\beta \quad e_2
\end{aligned}
$$

$$\dfrac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]} \; Red_\beta$$

Figure 8.2: Call-by-name operational semantics using evaluation contexts

The reduction relation $\mapsto_\beta$ is defined by the following equations:

$$
\begin{aligned}
(\lambda x\!:\!A.\, e)\, v \quad &\mapsto_\beta \quad [v/x]e \\
\text{if true then } e_1 \text{ else } e_2 \quad &\mapsto_\beta \quad e_1 \\
\text{if false then } e_1 \text{ else } e_2 \quad &\mapsto_\beta \quad e_2
\end{aligned}
$$

Figure 8.1 summarizes how to use evaluation contexts to specify the call-by-value operational semantics. An example of a reduction sequence is shown below. In each step, we underline the redex and show how to decompose a given expression into a unique evaluation context and a unique redex.

$$
\begin{aligned}
&\quad (\text{if } \underline{(\lambda x\!:\!\text{bool.}\, x)\, \text{true}} \text{ then } \lambda y\!:\!\text{bool.}\, y \text{ else } \lambda z\!:\!\text{bool.}\, z)\, \text{true} \\
=\;&\quad ((\text{if } \Box \text{ then } \lambda y\!:\!\text{bool.}\, y \text{ else } \lambda z\!:\!\text{bool.}\, z)\, \text{true})[\![(\lambda x\!:\!\text{bool.}\, x)\, \text{true}]\!] \\
\mapsto\;&\quad ((\text{if } \Box \text{ then } \lambda y\!:\!\text{bool.}\, y \text{ else } \lambda z\!:\!\text{bool.}\, z)\, \text{true})[\![\text{true}]\!] \hspace{2cm} Red_\beta \\
=\;&\quad (\text{if true then } \lambda y\!:\!\text{bool.}\, y \text{ else } \lambda z\!:\!\text{bool.}\, z)\, \text{true} \\
=\;&\quad (\Box\, \text{true})[\![\underline{\text{if true then } \lambda y\!:\!\text{bool.}\, y \text{ else } \lambda z\!:\!\text{bool.}\, z}]\!] \\
\mapsto\;&\quad (\Box\, \text{true})[\![\lambda y\!:\!\text{bool.}\, y]\!] \hspace{4.6cm} Red_\beta \\
=\;&\quad \underline{(\lambda y\!:\!\text{bool.}\, y)\, \text{true}} \\
=\;&\quad \Box[\![(\lambda y\!:\!\text{bool.}\, y)\, \text{true}]\!] \\
\mapsto\;&\quad \Box[\![\text{true}]\!] \hspace{6.3cm} Red_\beta \\
=\;&\quad \text{true}
\end{aligned}
$$

In order to obtain the call-by-name operational semantics, we only have to change the inductive definition of evaluation contexts and the reduction relation $\mapsto_\beta$, as shown in Figure 8.2. With a proper understanding of evaluation contexts, it should also be straightforward to incorporate those reduction rules in Chapter 5 into the definition of evaluation contexts and the reduction relation $\mapsto_\beta$. The reader is encouraged to try to augment the definition of evaluation contexts and the reduction relation $\mapsto_\beta$. See Figures 8.3 and 8.4 for the result.

Exercise 8.1. Give a definition of evaluation contexts corresponding to the weird reduction strategy specified in Exercise 3.10.

8.2 Type safety

As usual, type safety consists of progress and type preservation:

$$\text{evaluation context} \quad \kappa \quad ::= \quad \cdots \mid (\kappa, e) \mid (v, \kappa) \mid \mathsf{fst}\ \kappa \mid \mathsf{snd}\ \kappa \mid$$
$$\mathsf{inl}_A\ \kappa \mid \mathsf{inr}_A\ \kappa \mid \mathsf{case}\ \kappa\ \mathsf{of}\ \mathsf{inl}\ x.\,e \mid \mathsf{inr}\ x.\,e$$

$$
\begin{aligned}
\mathsf{fst}\ (v_1, v_2) &\longmapsto_\beta & v_1 \\
\mathsf{snd}\ (v_1, v_2) &\longmapsto_\beta & v_2 \\
\mathsf{case}\ \mathsf{inl}_A\ v\ \mathsf{of}\ \mathsf{inl}\ x_1.\,e_1 \mid \mathsf{inr}\ x_2.\,e_2 &\longmapsto_\beta & [v/x_1]e_1 \\
\mathsf{case}\ \mathsf{inr}_A\ v\ \mathsf{of}\ \mathsf{inl}\ x_1.\,e_1 \mid \mathsf{inr}\ x_2.\,e_2 &\longmapsto_\beta & [v/x_2]e_2 \\
\mathsf{fix}\ x\!:\!A.\,e &\longmapsto_\beta & [\mathsf{fix}\ x\!:\!A.\,e/x]e
\end{aligned}
$$

Figure 8.3: Extension for the eager reduction strategy

$$\text{evaluation context} \quad \kappa \quad ::= \quad \cdots \mid \mathsf{fst}\ \kappa \mid \mathsf{snd}\ \kappa \mid \mathsf{case}\ \kappa\ \mathsf{of}\ \mathsf{inl}\ x.\,e \mid \mathsf{inr}\ x.\,e$$

$$
\begin{aligned}
\mathsf{fst}\ (e_1, e_2) &\longmapsto_\beta & e_1 \\
\mathsf{snd}\ (e_1, e_2) &\longmapsto_\beta & e_2 \\
\mathsf{case}\ \mathsf{inl}_A\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\,e_1 \mid \mathsf{inr}\ x_2.\,e_2 &\longmapsto_\beta & [e/x_1]e_1 \\
\mathsf{case}\ \mathsf{inr}_A\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\,e_1 \mid \mathsf{inr}\ x_2.\,e_2 &\longmapsto_\beta & [e/x_2]e_2 \\
\mathsf{fix}\ x\!:\!A.\,e &\longmapsto_\beta & [\mathsf{fix}\ x\!:\!A.\,e/x]e
\end{aligned}
$$

Figure 8.4: Extension for the lazy reduction strategy

Theorem 8.2 (Progress). *If* $\cdot \vdash e : A$ *for some type* A, *then either* e *is a value or there exist* e' *such that* $e \mapsto e'$.

Theorem 8.3 (Type preservation). *If* $\Gamma \vdash e : A$ *and* $e \mapsto e'$, *then* $\Gamma \vdash e' : A$.

Since the rule Red_β uses not only a subexpression of a given expression but also an evaluation context for it, the proof of type safety requires a new typing judgments for evaluation contexts. We write $\Gamma \vdash \kappa : A \Rightarrow C$ to mean that given an expression of type A, the evaluation context κ produces an expression of type C:

$$\Gamma \vdash \kappa : A \Rightarrow C \qquad \Leftrightarrow \qquad \textit{if } \Gamma \vdash e : A, \textit{ then } \Gamma \vdash \kappa[\![e]\!] : C$$

We write $\kappa : A \Rightarrow C$ for $\cdot \vdash \kappa : A \Rightarrow C$.

The following inference rules are all admissible under the above definition of $\Gamma \vdash \kappa : A \Rightarrow C$. That is, we can prove that the premises imply the conclusion in each inference rule.

$$\frac{}{\Gamma \vdash \Box : A \Rightarrow A}\ \Box_{\mathsf{ctx}} \qquad \frac{\Gamma \vdash \kappa : A \Rightarrow B {\rightarrow} C \quad \Gamma \vdash e : B}{\Gamma \vdash \kappa\ e : A \Rightarrow C}\ \mathsf{Lam}_{\mathsf{ctx}}$$

$$\frac{\Gamma \vdash \lambda x\!:\!B.\,e : B {\rightarrow} C \quad \Gamma \vdash \kappa : A \Rightarrow B}{\Gamma \vdash (\lambda x\!:\!B.\,e)\ \kappa : A \Rightarrow C}\ \mathsf{Arg}_{\mathsf{ctx}} \qquad \frac{\Gamma \vdash \kappa : A \Rightarrow \mathsf{bool} \quad \Gamma \vdash e_1 : C \quad \Gamma \vdash e_2 : C}{\Gamma \vdash \mathsf{if}\ \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A \Rightarrow C}\ \mathsf{If}_{\mathsf{ctx}}$$

Proposition 8.4. *The rules* $\Box_{\mathsf{ctx}}$, $\mathsf{Lam}_{\mathsf{ctx}}$, $\mathsf{Arg}_{\mathsf{ctx}}$, *and* $\mathsf{If}_{\mathsf{ctx}}$ *are admissible.*

Proof. By using the definition of $\Gamma \vdash \kappa : A \Rightarrow C$. We show the case for the rule $\mathsf{Lam}_{\mathsf{ctx}}$.

Case $\dfrac{\Gamma \vdash \kappa : A \Rightarrow B {\rightarrow} C \quad \Gamma \vdash e : B}{\Gamma \vdash \kappa\ e : A \Rightarrow C}\ \mathsf{Lam}_{\mathsf{ctx}}$

$\Gamma \vdash \kappa : A \Rightarrow B {\rightarrow} C$ and $\Gamma \vdash e : B$ assumptions

$\Gamma \vdash e' : A$ assumption

$\Gamma \vdash \kappa[\![e']\!] : B {\rightarrow} C$ from $\Gamma \vdash \kappa : A \Rightarrow B {\rightarrow} C$ and $\Gamma \vdash e' : A$

$\Gamma \vdash \kappa[\![e']\!]\ e : C$ by the rule $\rightarrow$E

$\Gamma \vdash (\kappa\ e)[\![e']\!] : C$ from $\kappa[\![e']\!]\ e = (\kappa\ e)[\![e']\!]$

$\Gamma \vdash \kappa\ e : A \Rightarrow C$ from $\Gamma \vdash e' : A$ and $\Gamma \vdash (\kappa\ e)[\![e']\!] : C$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\Box$

The proof of Theorem 8.2 is similar to the proof of Theorem 4.3. The proof of Theorem 8.3 uses the following lemma whose proof uses Lemma 4.8:

Lemma 8.5. *If* $\Gamma \vdash \kappa[\![e]\!] : C$, *then* $\Gamma \vdash e : A$ *and* $\Gamma \vdash \kappa : A \Rightarrow C$ *for some type* A.

Proof. By structural induction on κ. We show the case for $\kappa = \kappa'\, e'$.

Case $\kappa = \kappa'\, e'$

$\Gamma \vdash \kappa[\![e]\!] : C$	assumption
$\Gamma \vdash (\kappa'[\![e]\!])\, e' : C$	$\kappa[\![e]\!] = (\kappa'[\![e]\!])\, e'$
$\Gamma \vdash \kappa'[\![e]\!] : B \to C$ and $\Gamma \vdash e' : B$ for some type B	by Lemma 4.8
$\Gamma \vdash e : A$ and $\Gamma \vdash \kappa' : A \Rightarrow B \to C$ for some type A	by induction hypothesis on κ'
$\Gamma \vdash \kappa'\, e' : A \Rightarrow C$	by the rule $\mathsf{Lam_{ctx}}$
$\Gamma \vdash \kappa : A \Rightarrow C$	$\square$

8.3 Abstract machine C

The concept of evaluation context leads to a concise formulation of the operational semantics, but it is not suitable for an actual implementation of the simply typed λ-calculus. The main reason is that the rule Red_β tacitly assumes an automatic decomposition of a given expression into a unique evaluation context and a unique redex, but it may in fact require an explicit analysis of the given expression in several steps. For example, in order to rewrite

$$e \;=\; (\text{if } (\lambda x\!:\!A.\, e'')\, v \text{ then } e_1 \text{ else } e_2)\, e'$$

as $((\text{if } \square \text{ then } e_1 \text{ else } e_2)\, e')[\![(\lambda x\!:\!A.\, e'')\, v]\!]$, we would analyze e in several steps:

$$
\begin{aligned}
e \;&=\; \square[\![(\text{if } (\lambda x\!:\!A.\, e'')\, v \text{ then } e_1 \text{ else } e_2)\, e']\!] \\
&=\; (\square\, e')[\![\text{if } (\lambda x\!:\!A.\, e'')\, v \text{ then } e_1 \text{ else } e_2]\!] \\
&=\; ((\text{if } \square \text{ then } e_1 \text{ else } e_2)\, e')[\![(\lambda x\!:\!A.\, e'')\, v]\!]
\end{aligned}
$$

The abstract machine C is another formulation of the operational semantics in which such an analysis is explicit.

Roughly speaking, the abstract machine C replaces an evaluation context by a *stack* of *frames* such that each frame corresponds to a specific step in the analysis of a given expression:

$$
\begin{aligned}
\text{frame} \quad \phi \;&::=\; \square\, e \mid (\lambda x\!:\!A.\, e)\, \square \mid \text{if } \square \text{ then } e_1 \text{ else } e_2 \\
\text{stack} \quad \sigma \;&::=\; \square \mid \sigma\, ;\phi
\end{aligned}
$$

Frames are special cases of evaluation contexts which are not defined inductively. Thus we may write $\phi[\![e]\!]$ for an expression obtained by filling the hole in ϕ with e. A stack of frames also represents an evaluation context in that given an expression, it determines a unique expression. To be specific, a stack σ and an expression e determine a unique expression $\sigma[\![e]\!]$ defined inductively as follows:

$$
\begin{aligned}
\square[\![e]\!] \;&=\; e \\
(\sigma\, ;\phi)[\![e]\!] \;&=\; \sigma[\![\phi[\![e]\!]]\!]
\end{aligned}
$$

If we write σ as $\square\, ;\phi_1\, ;\phi_2\, ;\cdots\, ;\phi_n$ for $n \geq 0$, $\sigma[\![e]\!]$ may be written as

$$\square[\![\phi_1[\![\phi_2[\![\cdots[\![\phi_n[\![e]\!]]\!]\cdots]\!]]\!]]\!].$$

Now, for example, an implicit analysis of $e = (\text{if } (\lambda x\!:\!A.\, e'')\, v \text{ then } e_1 \text{ else } e_2)\, e'$ shown above can be made explicit by using a stack of frames:

$$e \;=\; (\square\, ;\square\, e'\, ;\text{if } \square \text{ then } e_1 \text{ else } e_2)[\![(\lambda x\!:\!A.\, e'')\, v]\!]$$

Note that the top frame of a stack $\sigma\, ;\phi$ is ϕ and that the bottom of a stack is always $\square$.

A state of the abstract machine C is specified by a stack σ and an expression e, in which case the machine can be thought of as reducing an expression $\sigma[\![e]\!]$. In addition, the state includes a flag to indicate whether e needs to be further analyzed or has already been reduced to a value. Thus we use the following definition of states:

$$\text{state} \quad s \;::=\; \sigma \blacktriangleright e \mid \sigma \blacktriangleleft v$$

- $\sigma \blacktriangleright e$ means that the machine is currently reducing $\sigma[\![e]\!]$, but has yet to analyze e.

- $\sigma \blacktriangleleft v$ means that the machine is currently reducing $\sigma[\![v]\!]$ and has already analyzed v. That is, it is returning v to the top frame of σ.

Thus, if an expression e evaluates to a value v, a state $\sigma \blacktriangleright e$ will eventually lead to another state $\sigma \blacktriangleleft v$. As a special case, the initial state of the machine evaluating e is always $\square \blacktriangleright e$ and the final state $\square \blacktriangleleft v$ if e evaluates to v.

A state transition in the abstract machine C is specified by a reduction judgment $s \mapsto_\mathsf{C} s'$; we write $\mapsto_\mathsf{C}^*$ for the reflexive and transitive closure of $\mapsto_\mathsf{C}$. The guiding principle for state transitions is to maintain the invariant that $e \mapsto^* v$ holds if and only if $\sigma \blacktriangleright e \mapsto_\mathsf{C}^* \sigma \blacktriangleleft v$ holds for any stack σ. The rules for the reduction judgment $s \mapsto_\mathsf{C} s'$ are as follows:

$$\frac{}{\sigma \blacktriangleright v \mapsto_\mathsf{C} \sigma \blacktriangleleft v} \; Val_\mathsf{C} \qquad \frac{}{\sigma \blacktriangleright e_1\, e_2 \mapsto_\mathsf{C} \sigma;\square\, e_2 \blacktriangleright e_1} \; Lam_\mathsf{C}$$

$$\frac{}{\sigma;\square\, e_2 \blacktriangleleft \lambda x{:}A.\, e \mapsto_\mathsf{C} \sigma;(\lambda x{:}A.\, e)\, \square \blacktriangleright e_2} \; Arg_\mathsf{C} \qquad \frac{}{\sigma;(\lambda x{:}A.\, e)\, \square \blacktriangleleft v \mapsto_\mathsf{C} \sigma \blacktriangleright [v/x]e} \; App_\mathsf{C}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto_\mathsf{C} \sigma;\mathsf{if}\ \square\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \blacktriangleright e} \; If_\mathsf{C}$$

$$\frac{}{\sigma;\mathsf{if}\ \square\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \blacktriangleleft \mathsf{true} \mapsto_\mathsf{C} \sigma \blacktriangleright e_1} \; If_{\mathsf{true}\mathsf{C}} \qquad \frac{}{\sigma;\mathsf{if}\ \square\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \blacktriangleleft \mathsf{false} \mapsto_\mathsf{C} \sigma \blacktriangleright e_2} \; If_{\mathsf{false}\mathsf{C}}$$

An example of a reduction sequence is shown below. Note that it begins with a state $\square \blacktriangleright e$ and ends with a state $\square \blacktriangleleft v$.

$$
\begin{array}{lll}
 & \square \blacktriangleright (\mathsf{if}\ (\lambda x{:}\mathsf{bool}.\, x)\ \mathsf{true}\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z)\ \mathsf{true} & Lam_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true} \blacktriangleright \mathsf{if}\ (\lambda x{:}\mathsf{bool}.\, x)\ \mathsf{true}\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z & If_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true};\mathsf{if}\ \square\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z \blacktriangleright (\lambda x{:}\mathsf{bool}.\, x)\ \mathsf{true} & Lam_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true};\mathsf{if}\ \square\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z;\square\ \mathsf{true} \blacktriangleright \lambda x{:}\mathsf{bool}.\, x & Val_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true};\mathsf{if}\ \square\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z;\square\ \mathsf{true} \blacktriangleleft \lambda x{:}\mathsf{bool}.\, x & Arg_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true};\mathsf{if}\ \square\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z;(\lambda x{:}\mathsf{bool}.\, x)\ \square \blacktriangleright \mathsf{true} & Val_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true};\mathsf{if}\ \square\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z;(\lambda x{:}\mathsf{bool}.\, x)\ \square \blacktriangleleft \mathsf{true} & App_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true};\mathsf{if}\ \square\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z \blacktriangleright \mathsf{true} & Val_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true};\mathsf{if}\ \square\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z \blacktriangleleft \mathsf{true} & If_{\mathsf{true}\mathsf{C}} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true} \blacktriangleright \lambda y{:}\mathsf{bool}.\, y & Val_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;\square\ \mathsf{true} \blacktriangleleft \lambda y{:}\mathsf{bool}.\, y & Arg_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;(\lambda y{:}\mathsf{bool}.\, y)\ \square \blacktriangleright \mathsf{true} & Val_\mathsf{C} \\
\mapsto_\mathsf{C} & \square;(\lambda y{:}\mathsf{bool}.\, y)\ \square \blacktriangleleft \mathsf{true} & App_\mathsf{C} \\
\mapsto_\mathsf{C} & \square \blacktriangleright \mathsf{true} & Val_\mathsf{C} \\
\mapsto_\mathsf{C} & \square \blacktriangleleft \mathsf{true} &
\end{array}
$$

8.4 Correctness of the abstract machine C

This section presents a proof of the correctness of the abstract machine C as stated in the following theorem:

Theorem 8.6. $e \mapsto^* v$ *if and only if* $\square \blacktriangleright e \mapsto_\mathsf{C}^* \square \blacktriangleleft v$.

A more general version of the theorem allows any stack σ in place of $\square$, but we do not prove it here. For the sake of simplicity, we also do not consider expressions of type bool altogether.

It is a good and challenging exercise to prove the theorem. The main difficulty lies in finding several lemmas necessary for proving the theorem, not in constructing their proofs. The reader is encouraged to guess these lemmas without having to write their proofs.

The proof uses a generalization of $\kappa[\![\cdot]\!]$ and $\sigma[\![\cdot]\!]$ over evaluation contexts:

$$
\begin{array}{rcl}
\square[\![\kappa']\!] & = & \kappa' \\
(\kappa\ e)[\![\kappa']\!] & = & \kappa[\![\kappa']\!]\ e \\
((\lambda x{:}A.\, e)\ \kappa)[\![\kappa']\!] & = & (\lambda x{:}A.\, e)\ \kappa[\![\kappa']\!]
\end{array}
\qquad\qquad
\begin{array}{rcl}
\square[\![\kappa]\!] & = & \kappa \\
(\sigma;\phi)[\![\kappa]\!] & = & \sigma[\![\phi[\![\kappa]\!]]\!]
\end{array}
$$

Note that $\kappa[\![\kappa']\!]$ and $\sigma[\![\kappa]\!]$ are evaluation contexts.

Proposition 8.7. $\kappa[\![\kappa'[\![e]\!]]\!] = \kappa[\![\kappa']\!][\![e]\!]$.

Proof. By structural induction on κ. We show two cases.

Case $\kappa = \square$:
$\square[\![\kappa'[\![e]\!]]\!] = \kappa'[\![e]\!] = \square[\![\kappa']\!][\![e]\!]$

Case $\kappa = \kappa''\ e''$:
$(\kappa''\ e'')[\![\kappa'[\![e]\!]]\!] = \kappa''[\![\kappa'[\![e]\!]]\!]\ e'' = \kappa''[\![\kappa']\!][\![e]\!]\ e'' = (\kappa''[\![\kappa']\!]\ e'')[\![e]\!] = (\kappa''\ e'')[\![\kappa']\!][\![e]\!]$ □

Proposition 8.8. $\sigma[\![\kappa[\![e]\!]]\!] = \sigma[\![\kappa]\!][\![e]\!]$.

Proof. By structural induction on σ. The second case uses Proposition 8.7.

Case $\sigma = \square$:
$\square[\![\kappa[\![e]\!]]\!] = \kappa[\![e]\!] = \square[\![\kappa]\!][\![e]\!]$.

Case $\sigma = \sigma'; \phi$:
$(\sigma'; \phi)[\![\kappa[\![e]\!]]\!] = \sigma'[\![\phi[\![\kappa[\![e]\!]]\!]]\!] = \sigma'[\![\phi[\![\kappa]\!][\![e]\!]]\!] = \sigma'[\![\phi[\![\kappa]\!]]\!][\![e]\!] = (\sigma'; \phi)[\![\kappa]\!][\![e]\!]$. □

Lemma 8.9. *For σ and κ, there exists σ' such that $\sigma \blacktriangleright \kappa[\![e]\!] \mapsto_{\mathsf{C}}^* \sigma' \blacktriangleright e$ and $\sigma[\![\kappa]\!] = \sigma'[\![\square]\!]$ for any expression e.*

Proof. By structural induction on κ. We show two cases.

Case $\kappa = \square$:
We let $\sigma' = \sigma$.

Case $\kappa = \kappa'\ e'$:

$\sigma \blacktriangleright (\kappa'\ e')[\![e]\!] = \sigma \blacktriangleright \kappa'[\![e]\!]\ e' \mapsto_{\mathsf{C}} \sigma; \square\ e' \blacktriangleright \kappa'[\![e]\!]$	by the rule Lam_{C}
$\sigma; \square\ e' \blacktriangleright \kappa'[\![e]\!] \mapsto_{\mathsf{C}}^* \sigma' \blacktriangleright e$ and $(\sigma; \square\ e')[\![\kappa']\!] = \sigma'[\![\square]\!]$	by induction hypothesis
$\sigma[\![\kappa]\!] = \sigma[\![\kappa'\ e']\!] = \sigma[\![(\square\ e')[\![\kappa']\!]]\!] = (\sigma; \square\ e')[\![\kappa']\!] = \sigma'[\![\square]\!]$	□

Lemma 8.10. *Suppose $\sigma[\![e]\!] = \kappa[\![f\ v]\!]$ where f is a λ-abstraction. Then one of the following cases holds:*
(1) $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma' \blacktriangleright \kappa'[\![f\ v]\!]$ and $\sigma'[\![\kappa']\!] = \kappa$
(2) $\sigma = \sigma'; \square\ v$ and $e = f$ and $\sigma'[\![\square]\!] = \kappa$
(3) $\sigma = \sigma'; f\ \square$ and $e = v$ and $\sigma'[\![\square]\!] = \kappa$

Proof. By structural induction on σ. We show two cases.

Case $\sigma = \square$:

$\sigma[\![e]\!] = e = \kappa[\![f\ v]\!]$	assumption
$\sigma \blacktriangleright e = \square \blacktriangleright \kappa[\![f\ v]\!]$	
(1) $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \square \blacktriangleright \kappa[\![f\ v]\!]$ and $\square[\![\kappa]\!] = \kappa$	

Case $\sigma = \sigma'; \square\ e'$:
$\sigma[\![e]\!] = (\sigma'; \square\ e')[\![e]\!] = \sigma'[\![(\square\ e')[\![e]\!]]\!] = \sigma'[\![e\ e']\!] = \kappa[\![f\ v]\!]$

Subcase (1) $\sigma' \blacktriangleright e\ e' \mapsto_{\mathsf{C}}^* \sigma'' \blacktriangleright \kappa'[\![f\ v]\!]$ and $\sigma''[\![\kappa']\!] = \kappa$:   by induction hypothesis on σ'

$\sigma' \blacktriangleright e\ e' \mapsto_{\mathsf{C}} \sigma'; \square\ e' \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma'' \blacktriangleright \kappa'[\![f\ v]\!]$	assumption
(1) $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma'' \blacktriangleright \kappa'[\![f\ v]\!]$ and $\sigma''[\![\kappa']\!] = \kappa$	

$\sigma' = \sigma''$ and $e\ e' = \kappa'[\![f\ v]\!]$, and $\kappa' = \square$	assumption
$e = f$ and $e' = v$	
(2) $\sigma = \sigma'; \square\ v$ and $e = f$ and $\sigma'[\![\square]\!] = \sigma''[\![\kappa']\!] = \kappa$	

$\sigma' = \sigma''$ and $e\,e' = \kappa'[\![f\,v]\!]$, and $\kappa' = \kappa''\,e'$ assumption
$e = \kappa''[\![f\,v]\!]$
(1) $\sigma \blacktriangleright e \mapsto^*_C \sigma \blacktriangleright \kappa''[\![f\,v]\!]$ and

$$\sigma[\![\kappa'']\!] = (\sigma''; \square\,e')[\![\kappa'']\!] = \sigma''[\![\kappa''\,e']\!] = \sigma''[\![\kappa']\!] = \kappa$$

$\sigma' = \sigma''$ and $e\,e' = \kappa'[\![f\,v]\!]$, and $\kappa' = e\,\kappa''$ assumption
e is a λ-abstraction and $e' = \kappa''[\![f\,v]\!]$
(1) $\sigma \blacktriangleright e = \sigma'; \square\,e' \blacktriangleright e \mapsto_C \sigma'; \square\,e' \blacktriangleleft e \mapsto_C \sigma'; e\,\square \blacktriangleright e' = \sigma'; e\,\square \blacktriangleright \kappa''[\![f\,v]\!]$
$$\text{and } (\sigma'; e\,\square)[\![\kappa'']\!] = \sigma'[\![e\,\kappa'']\!] = \sigma''[\![\kappa']\!] = \kappa$$

Subcases (2) and (3) impossible $e\,e' \neq f$ and $e\,e' \neq v$ $\square$

Lemma 8.11. *Suppose $\sigma[\![e]\!] = \kappa[\![f\,v]\!]$ where f is a λ-abstraction and $f\,v \mapsto_\beta e'$. Then $\sigma \blacktriangleright e \mapsto^*_C \sigma^* \blacktriangleright e'$ and $\sigma^*[\![e']\!] = \kappa[\![e']\!]$.*

Proof. By Lemma 8.10, we need to consider the following three cases:

(1) $\sigma \blacktriangleright e \mapsto^*_C \sigma' \blacktriangleright \kappa'[\![f\,v]\!]$ and $\sigma'[\![\kappa']\!] = \kappa$

$$
\begin{aligned}
\sigma \blacktriangleright e \quad &\mapsto^*_C \quad \sigma' \blacktriangleright \kappa'[\![f\,v]\!] \\
&\mapsto^*_C \quad \sigma'' \blacktriangleright f\,v &&\text{where } \sigma'[\![\kappa']\!] = \sigma''[\![\square]\!] \text{ by Lemma 8.9} \\
&\mapsto_C \quad \sigma''; \square\,v \blacktriangleright f \\
&\mapsto_C \quad \sigma''; \square\,v \blacktriangleleft f \\
&\mapsto_C \quad \sigma''; f\,\square \blacktriangleright v \\
&\mapsto_C \quad \sigma''; f\,\square \blacktriangleleft v \\
&\mapsto_C \quad \sigma'' \blacktriangleright e'
\end{aligned}
$$
$\sigma''[\![e']\!] = \sigma''[\![\square[\![e']\!]]\!] = \sigma''[\![\square]\!][\![e']\!] = \sigma'[\![\kappa']\!][\![e']\!] = \kappa[\![e']\!]$ by Proposition 8.8
We let $\sigma^* = \sigma''$.

(2) $\sigma = \sigma'; \square\,v$ and $e = f$ and $\sigma'[\![\square]\!] = \kappa$

$$
\begin{aligned}
\sigma \blacktriangleright e \quad &= \quad \sigma'; \square\,v \blacktriangleright f \\
&\mapsto^*_C \quad \sigma' \blacktriangleright e'
\end{aligned}
$$
$\sigma'[\![e']\!] = \sigma'[\![\square[\![e']\!]]\!] = \sigma'[\![\square]\!][\![e']\!] = \kappa[\![e']\!]$ by Proposition 8.8
We let $\sigma^* = \sigma'$.

(3) $\sigma = \sigma'; f\,\square$ and $e = v$ and $\sigma'[\![\square]\!] = \kappa$

$$
\begin{aligned}
\sigma \blacktriangleright e \quad &= \quad \sigma'; f\,\square \blacktriangleright v \\
&\mapsto^*_C \quad \sigma' \blacktriangleright e'
\end{aligned}
$$
$\sigma'[\![e']\!] = \sigma'[\![\square[\![e']\!]]\!] = \sigma'[\![\square]\!][\![e']\!] = \kappa[\![e']\!]$ by Proposition 8.8
We let $\sigma^* = \sigma'$. $\square$

Corollary 8.12. *Suppose $e_1 \mapsto e_2$ and $\sigma[\![e]\!] = e_1$. Then there exist σ' and e' such that $\sigma \blacktriangleright e \mapsto^*_C \sigma' \blacktriangleright e'$ and $\sigma'[\![e']\!] = e_2$.*

We leave it to the reader to prove all results given below.

Proposition 8.13. *Suppose $e \mapsto^* v$ and $\sigma[\![e']\!] = e$. Then $\sigma \blacktriangleright e' \mapsto^*_C \square \blacktriangleleft v$.*

Corollary 8.14. *If $e \mapsto^* v$, then $\square \blacktriangleright e \mapsto^*_C \square \blacktriangleleft v$.*

Proposition 8.15.
    If $\sigma \blacktriangleright e \mapsto_C \sigma' \blacktriangleright e'$, then $\sigma[\![e]\!] \mapsto^* \sigma'[\![e']\!]$.
    If $\sigma \blacktriangleright e \mapsto_C \sigma' \blacktriangleleft v'$, then $\sigma[\![e]\!] \mapsto^* \sigma'[\![v']\!]$.

Corollary 8.16.
    If $\sigma \blacktriangleright e \mapsto^*_C \sigma' \blacktriangleright e'$, then $\sigma[\![e]\!] \mapsto^* \sigma'[\![e']\!]$.
    If $\sigma \blacktriangleright e \mapsto^*_C \sigma' \blacktriangleleft v'$, then $\sigma[\![e]\!] \mapsto^* \sigma'[\![v']\!]$.

Corollary 8.17. *If $\square \blacktriangleright e \mapsto^*_C \square \blacktriangleleft v$, then $e \mapsto^* v$.*

Corollaries 8.14 and 8.17 prove Theorem 8.6.

8.5 Safety of the abstract machine C

The safety of the abstract machine C is proven independently of its correctness. We use two judgments to describe the state of C with three inference rules given below:

- s okay means that s is an "okay" state. That is, C is ready to analyze a given expression.

- s stop means that s is a "stop" state. That is, C has finished reducing a given expression.

$$\frac{\sigma[\![\Box]\!] : A \Rightarrow C \quad \cdot \vdash e : A}{\sigma \blacktriangleright e \text{ okay}} \text{ Okay}_{\blacktriangleright} \qquad \frac{\sigma[\![\Box]\!] : A \Rightarrow C \quad \cdot \vdash v : A}{\sigma \blacktriangleleft v \text{ okay}} \text{ Okay}_{\blacktriangleleft} \qquad \frac{\cdot \vdash v : A}{\Box \blacktriangleleft v \text{ stop}} \text{ Stop}_{\blacktriangleleft}$$

The first clause in the following theorem may be thought of as the progress property of the abstract machine C; the second clause may be thought of as the "state" preservation property.

Theorem 8.18 (Safety of the abstract machine C).
If s okay, then either s stop or there exists s' such that $s \mapsto_C s'$.
If s okay and $s \mapsto_C s'$, then s' okay.

8.6 Exercises

Exercise 8.19. Prove Theorems 8.2 and 8.3.

Exercise 8.20. Consider the simply typed λ-calculus extended with product types, sum types, and the fixed point construct.

$$
\begin{array}{lll}
\text{expression} & e \quad ::= & x \mid \lambda x\!:\!A.\,e \mid e\,e \mid (e,e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid () \mid \\
& & \mathsf{inl}_A\ e \mid \mathsf{inr}_A\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x.\,e \mid \mathsf{inr}\ x.\,e \mid \mathsf{fix}\ x\!:\!A.\,e \mid \\
& & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
\text{value} & v \quad ::= & \lambda x\!:\!A.\,e \mid (v,v) \mid () \mid \mathsf{inl}_A\ v \mid \mathsf{inr}_A\ v \mid \mathsf{true} \mid \mathsf{false}
\end{array}
$$

Assuming the call-by-value strategy, extend the definition of frames and give additional rules for the reduction judgment $s \mapsto_C s'$ for the abstract machine C. See Figure 8.5 for an answer.

Exercise 8.21. Prove Theorem 8.18.

frame $\quad \phi \quad ::= \quad \square\,e \mid v\,\square \mid (\square, e) \mid (v, \square) \mid \mathsf{fst}\,\square \mid \mathsf{snd}\,\square \mid$
$\qquad\qquad\qquad \mathsf{inl}_A\,\square \mid \mathsf{inr}_A\,\square \mid \mathsf{case}\,\square\,\mathsf{of}\,\mathsf{inl}\,x.\,e \mid \mathsf{inr}\,x.\,e \mid \mathsf{if}\,\square\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2$

$$\frac{}{\sigma \blacktriangleright (e_1, e_2) \mapsto_\mathsf{C} \sigma; (\square, e_2) \blacktriangleright e_1}\;Pair_\mathsf{C} \qquad \frac{}{\sigma; (\square, e_2) \blacktriangleleft v_1 \mapsto_\mathsf{C} \sigma; (v_1, \square) \blacktriangleright e_2}\;Pair'_\mathsf{C}$$

$$\frac{}{\sigma; (v_1, \square) \blacktriangleleft v_2 \mapsto_\mathsf{C} \sigma \blacktriangleleft (v_1, v_2)}\;Pair''_\mathsf{C}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{fst}\,e \mapsto_\mathsf{C} \sigma; \mathsf{fst}\,\square \blacktriangleright e}\;Fst_\mathsf{C} \qquad \frac{}{\sigma; \mathsf{fst}\,\square \blacktriangleleft (v_1, v_2) \mapsto_\mathsf{C} \sigma \blacktriangleleft v_1}\;Fst'_\mathsf{C}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{snd}\,e \mapsto_\mathsf{C} \sigma; \mathsf{snd}\,\square \blacktriangleright e}\;Snd_\mathsf{C} \qquad \frac{}{\sigma; \mathsf{snd}\,\square \blacktriangleleft (v_1, v_2) \mapsto_\mathsf{C} \sigma \blacktriangleleft v_2}\;Snd'_\mathsf{C}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{inl}_A\,e \mapsto_\mathsf{C} \sigma; \mathsf{inl}_A\,\square \blacktriangleright e}\;Inl_\mathsf{C} \qquad \frac{}{\sigma; \mathsf{inl}_A\,\square \blacktriangleleft v \mapsto_\mathsf{C} \sigma \blacktriangleleft \mathsf{inl}_A\,v}\;Inl'_\mathsf{C}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{inr}_A\,e \mapsto_\mathsf{C} \sigma; \mathsf{inr}_A\,\square \blacktriangleright e}\;Inr_\mathsf{C} \qquad \frac{}{\sigma; \mathsf{inr}_A\,\square \blacktriangleleft v \mapsto_\mathsf{C} \sigma \blacktriangleleft \mathsf{inr}_A\,v}\;Inr'_\mathsf{C}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{case}\,e\,\mathsf{of}\,\mathsf{inl}\,x_1.\,e_1 \mid \mathsf{inr}\,x_2.\,e_2 \mapsto_\mathsf{C} \sigma; \mathsf{case}\,\square\,\mathsf{of}\,\mathsf{inl}\,x_1.\,e_1 \mid \mathsf{inr}\,x_2.\,e_2 \blacktriangleright e}\;Case_\mathsf{C}$$

$$\frac{}{\sigma; \mathsf{case}\,\square\,\mathsf{of}\,\mathsf{inl}\,x_1.\,e_1 \mid \mathsf{inr}\,x_2.\,e_2 \blacktriangleleft \mathsf{inl}_A\,v \mapsto_\mathsf{C} \sigma \blacktriangleright [v/x_1]e_1}\;Case'_\mathsf{C}$$

$$\frac{}{\sigma; \mathsf{case}\,\square\,\mathsf{of}\,\mathsf{inl}\,x_1.\,e_1 \mid \mathsf{inr}\,x_2.\,e_2 \blacktriangleleft \mathsf{inr}_A\,v \mapsto_\mathsf{C} \sigma \blacktriangleright [v/x_2]e_2}\;Case''_\mathsf{C}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{fix}\,x{:}A.\,e \mapsto_\mathsf{C} \sigma \blacktriangleright [\mathsf{fix}\,x{:}A.\,e/x]e}\;Fix_\mathsf{C}$$

Figure 8.5: Abstract machine C for product types, sum types, and the fixed point construct

Chapter 9

Environments and Closures

The operational semantics of the simply typed (or untyped) λ-calculus discussed so far hinges on substitutions in reducing such expressions as applications, case expressions, and the fixed point construct. Since the definition of a substitution $[e'/x]e$ analyzes the structure of e to find all occurrences of x, a naive implementation of substitutions can be extremely inefficient in terms of time, especially because of the potential size of e. Even worse, x may not appear at all in e, in which case all the work put into the analysis of e is wasted.

This chapter presents another form of operational semantics, called *environment semantics*, which overcomes the inefficiency of the naive implementation of substitutions. The environment semantics does not entirely eliminate the need for substitutions, but it performs substitutions only if necessary by postponing them as much as possible. The development of the environment semantics also leads to the introduction of another important concept called *closures*, which are compact representations of closed λ-abstractions (*i.e.*, those containing no free variables) generated during evaluations.

Before presenting the environment semantics, we develop a new form of judgment for "evaluating" expressions (as opposed to "reducing" expressions). In comparison with the reduction judgment, the new judgment lends itself better to explaining the key idea behind the environment semantics.

9.1   Evaluation judgment

As in Chapter 8, we consider the fragment of the simply typed λ-calculus consisting of the boolean type and function types:

$$
\begin{array}{rrcl}
\text{type} & A & ::= & P \mid A \to A \\
\text{base type} & P & ::= & \text{bool} \\
\text{expression} & e & ::= & x \mid \lambda x : A.\, e \mid e\, e \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \\
\text{value} & v & ::= & \lambda x : A.\, e \mid \text{true} \mid \text{false}
\end{array}
$$

In a certain sense, a reduction judgment $e \mapsto e'$ takes a single *small* step toward completing the evaluation of e, since the evaluation of e to a value requires a sequence of such steps in general. For this reason, the operational semantics based on the reduction judgment $e \mapsto e'$ is often called a *small-step* semantics.

An opposite approach is to take a single *big* step with which we immediately finish evaluating a given expression. To realize this approach, we introduce an *evaluation judgment* of the form $e \hookrightarrow v$:

$$ e \hookrightarrow v \qquad \Leftrightarrow \qquad e \text{ evaluates to } v $$

The intuition behind the evaluation judgment is that $e \hookrightarrow v$ conveys the same meaning as $e \mapsto^* v$ (which we will actually prove in Theorem 9.2). An operational semantics based on the evaluation judgment is often called a *big-step* semantics.

We refer to an inference rule deducing an evaluation judgment as an *evaluation rule*. Unlike a reduction judgment which is never applied to a value (*i.e.*, no reduction judgment of the form $v \mapsto e$), an evaluation judgment $v \hookrightarrow v$ is always valid because $v \mapsto^* v$ holds for any value v. The three reduction rules *Lam*, *Arg*, and *App* for applications (under the call-by-value strategy) are now merged into a single evaluation rule with three premises:

$$\frac{}{\lambda x{:}A.\,e \hookrightarrow \lambda x{:}A.\,e} \ \textbf{Lam} \qquad \frac{e_1 \hookrightarrow \lambda x{:}A.\,e \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e \hookrightarrow v}{e_1\, e_2 \hookrightarrow v} \ \textbf{App}$$

$$\frac{}{\mathsf{true} \hookrightarrow \mathsf{true}} \ \textbf{True} \qquad \frac{}{\mathsf{false} \hookrightarrow \mathsf{false}} \ \textbf{False} \qquad \frac{e \hookrightarrow \mathsf{true} \quad e_1 \hookrightarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \hookrightarrow v} \ \textbf{If}_{\mathsf{true}} \qquad \frac{e \hookrightarrow \mathsf{false} \quad e_2 \hookrightarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \hookrightarrow v} \ \textbf{If}_{\mathsf{false}}$$

Note that there is only one rule for each form of expression. In other words, the evaluation rules are syntax-directed. Thus we may invert an evaluation rule so that its conclusion justifies the use of its premises. (See Lemma 4.8 for a similar example.) For example, $e_1\, e_2 \hookrightarrow v$ asserts the existence of $\lambda x{:}A.\,e$ and v_2 such that $e_1 \hookrightarrow \lambda x{:}A.\,e$, $e_2 \hookrightarrow v_2$, and $[v_2/x]e \hookrightarrow v$.

The following derivation (with evaluation rule names omitted) shows how to evaluate $(\lambda x{:}\mathsf{bool}.\,x)\,((\lambda y{:}\mathsf{bool}.\,y)\,\mathsf{true})$ to true in a single "big" step:

$$\frac{\dfrac{}{\lambda x{:}\mathsf{bool}.\,x \hookrightarrow \lambda x{:}\mathsf{bool}.\,x} \qquad \dfrac{\dfrac{}{\lambda y{:}\mathsf{bool}.\,y \hookrightarrow \lambda y{:}\mathsf{bool}.\,y} \quad \dfrac{}{\mathsf{true} \hookrightarrow \mathsf{true}} \quad \dfrac{}{[\mathsf{true}/y]y \hookrightarrow \mathsf{true}}}{(\lambda y{:}\mathsf{bool}.\,y)\,\mathsf{true} \hookrightarrow \mathsf{true}} \qquad \dfrac{}{[\mathsf{true}/x]x \hookrightarrow \mathsf{true}}}{(\lambda x{:}\mathsf{bool}.\,x)\,((\lambda y{:}\mathsf{bool}.\,y)\,\mathsf{true}) \hookrightarrow \mathsf{true}}$$

Exercise 9.1. For the fragment of the simply typed λ-calculus consisting of variables, λ-abstractions, and applications, give rules for the evaluation judgment $e \hookrightarrow v$ corresponding to the call-by-name reduction strategy. Also give rules for the weird reduction strategy specified in Exercise 3.10.

Theorem 9.2 states the relationship between evaluation judgments and reduction judgments; the proof consists of proofs of Propositions 9.3 and 9.4:

Theorem 9.2. $e \hookrightarrow v$ if and only if $e \mapsto^* v$.

Proposition 9.3. If $e \hookrightarrow v$, then $e \mapsto^* v$.

Proposition 9.4. If $e \mapsto^* v$, then $e \hookrightarrow v$.

The proof of Proposition 9.3 proceeds by rule induction on the judgment $e \hookrightarrow v$ and uses Lemma 9.5. The proof of Lemma 9.5 essentially uses mathematical induction on the length of the reduction sequence $e \mapsto^* e'$, but we recast the proof in terms of rule induction with the following inference rules (as in Exercise 3.11):

$$\frac{}{e \mapsto^* e} \ \textit{Refl} \qquad \frac{e \mapsto e'' \quad e'' \mapsto^* e'}{e \mapsto^* e'} \ \textit{Trans}$$

Lemma 9.5. Suppose $e \mapsto^* e'$.
    (1) $e\, e'' \mapsto^* e'\, e''$.
    (2) $(\lambda x{:}A.\,e'')\,e \mapsto^* (\lambda x{:}A.\,e'')\,e'$.
    (3) if e then e_1 else $e_2 \mapsto^*$ if e' then e_1 else e_2.

Proof. By rule induction on the judgment $e \mapsto^* e'$. We consider the clause (1). The other two clauses are proven in a similar way.

Case $\dfrac{}{e \mapsto^* e}$ *Refl* where $e' = e$:

$e\, e'' \mapsto^* e'\, e''$ from $e\, e'' = e'\, e''$ and the rule *Refl*

Case $\dfrac{e \mapsto e_t \quad e_t \mapsto^* e'}{e \mapsto^* e'}$ *Trans*

$e_t\, e'' \mapsto^* e'\, e''$ by induction hypothesis on $e_t \mapsto^* e'$

$e\, e'' \mapsto e_t\, e''$ from $\dfrac{e \mapsto e_t}{e\, e'' \mapsto e_t\, e''}$ *Lam*

$e\, e'' \mapsto^* e'\, e''$ from $\dfrac{e\, e'' \mapsto e_t\, e'' \quad e_t\, e'' \mapsto^* e'\, e''}{e\, e'' \mapsto^* e'\, e''}$ *Trans*

$\blacksquare$

Lemma 9.6. If $e \mapsto^* e'$ and $e' \mapsto^* e''$, then $e \mapsto^* e''$.

Proof. See Exercise 3.11. □

Proof of Proposition 9.3. By rule induction on the judgment $e \hookrightarrow v$. If $e = v$, then $e \mapsto^* v$ holds by the rule *Refl*. Hence we need to consider the cases for the rules **App**, **If**$_{\text{true}}$, and **If**$_{\text{false}}$. We show the case for the rule **App**.

Case $\dfrac{e_1 \hookrightarrow \lambda x{:}A.\, e' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e' \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$ **App** where $e = e_1\, e_2$:

$e_1 \mapsto^* \lambda x{:}A.\, e'$ by induction hypothesis on $e_1 \hookrightarrow \lambda x{:}A.\, e'$

$e_1\, e_2 \mapsto^* (\lambda x{:}A.\, e')\, e_2$ by Lemma 9.5

$e_2 \mapsto^* v_2$ by induction hypothesis on $e_2 \hookrightarrow v_2$

$(\lambda x{:}A.\, e')\, e_2 \mapsto^* (\lambda x{:}A.\, e')\, v_2$ by Lemma 9.5

$[v_2/x]e' \mapsto^* v$ by induction hypothesis on $[v_2/x]e' \hookrightarrow v$

$(\lambda x{:}A.\, e')\, v_2 \mapsto^* v$ $\dfrac{\dfrac{}{(\lambda x{:}A.\, e')\, v_2 \mapsto [v_2/x]e'}\ App \quad [v_2/x]e' \mapsto^* v}{(\lambda x{:}A.\, e')\, v_2 \mapsto^* v}\ Trans$

$e_1\, e_2 \mapsto^* v$ from Lemma 9.6 and $e_1\, e_2 \mapsto^* (\lambda x{:}A.\, e')\, e_2$,

    $(\lambda x{:}A.\, e')\, e_2 \mapsto^* (\lambda x{:}A.\, e')\, v_2$,

    $(\lambda x{:}A.\, e')\, v_2 \mapsto^* v$. □

The proof of Proposition 9.4 proceeds by rule induction on the judgment $e \mapsto^* v$, but is not as straightforward as the proof of Proposition 9.3. Consider the case $\dfrac{e \mapsto e' \quad e' \mapsto^* v}{e \mapsto^* v}$ *Trans* . By induction hypothesis on $e' \mapsto^* v$, we obtain $e' \hookrightarrow v$. Then we need to prove $e \hookrightarrow v$ using $e \mapsto e'$ and $e' \hookrightarrow v$, which is not addressed by the proposition being proven. Thus we are led to prove the following lemma before proving Proposition 9.4:

Lemma 9.7. *If $e \mapsto e'$ and $e' \hookrightarrow v$, then $e \hookrightarrow v$.*

Proof. By rule induction on the judgment $e \mapsto e'$ (not on $e' \hookrightarrow v$). We show a representative case:

Case $\dfrac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}$ *Lam* where $e = e_1\, e_2$ and $e' = e_1'\, e_2$:

$\dfrac{e_1' \hookrightarrow \lambda x{:}A.\, e_1'' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e_1'' \hookrightarrow v}{e_1'\, e_2 \hookrightarrow v}$ **App**   by the syntax-directedness of the evaluation rules

$e_1 \hookrightarrow \lambda x{:}A.\, e_1''$ by induction hypothesis on $e_1 \mapsto e_1'$ with $e_1' \hookrightarrow \lambda x{:}A.\, e_1''$

$\dfrac{e_1 \hookrightarrow \lambda x{:}A.\, e_1'' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e_1'' \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$ **App** □

Proof of Proposition 9.4. By rule induction on the judgment $e \mapsto^* v$.

Case $\dfrac{}{e \mapsto^* e}$ *Refl* where $e = v$:

$e \hookrightarrow v$ by the rule **Lam**, **True**, or **False**

Case $\dfrac{e \mapsto e' \quad e' \mapsto^* v}{e \mapsto^* v}$ *Trans*

$e' \hookrightarrow v$ by induction hypothesis on $e' \mapsto^* v$

$e \hookrightarrow v$ by Lemma 9.7 with $e \mapsto e'$ and $e' \hookrightarrow v$ □

9.2 Environment semantics

The key idea behind the environment semantics is to postpone a substitution $[v/x]e$ in the rule **App** by storing a pair of v and x in an *environment* and then continuing to evaluate e without modifying it. When we later encounter an occurrence of x within e and need to evaluate it, we look up the environment to retrieve the actual value v for x. We use the following inductive definition of environment:

$$\text{environment} \quad \eta \quad ::= \quad \cdot \mid \eta, x \hookrightarrow v$$

$\cdot$ denotes an empty environment, and $x \hookrightarrow v$ means that variable x is to be replaced by value v. As in the definition of typing context, we assume that variables in an environment are all distinct.

We use an *environment evaluation judgment* of the form $\eta \vdash e \hookrightarrow v$:[1]

$$\eta \vdash e \hookrightarrow v \qquad \Leftrightarrow \qquad e \text{ evaluates to } v \text{ under environment } \eta$$

As an example, let us evaluate $[\text{true}/x]$if x then e_1 else e_2 using the environment semantics. For the sake of simplicity, we begin with an empty environment:

$$\cdot \vdash [\text{true}/x]\text{if } x \text{ then } e_1 \text{ else } e_2 \hookrightarrow ?$$

Instead of applying the substitution right away, we evaluate if x then e_1 else e_2 under an augmented environment $x \hookrightarrow \text{true}$ (which is an abbreviation of $\cdot, x \hookrightarrow \text{true}$):

$$\frac{\cdots}{x \hookrightarrow \text{true} \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 \hookrightarrow ?}$$

To evaluate the conditional expression x, we look up the environment to retrieve its value:

$$\frac{x \hookrightarrow \text{true} \vdash x \hookrightarrow \text{true} \quad \cdots}{x \hookrightarrow \text{true} \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 \hookrightarrow ?}$$

Since the conditional expression evaluates to true, we take the if branch without changing the environment:

$$\frac{x \hookrightarrow \text{true} \vdash x \hookrightarrow \text{true} \quad x \hookrightarrow \text{true} \vdash e_1 \hookrightarrow ?}{x \hookrightarrow \text{true} \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 \hookrightarrow ?}$$

If we let $e_1 = x$ and $e_2 = x$, we obtain the following derivation tree:

$$\frac{x \hookrightarrow \text{true} \vdash x \hookrightarrow \text{true} \quad x \hookrightarrow \text{true} \vdash x \hookrightarrow \text{true}}{x \hookrightarrow \text{true} \vdash \text{if } x \text{ then } x \text{ else } x \hookrightarrow \text{true}}$$

Note that the evaluation does not even look at expression x in the else branch (because it does not need to), and thus accesses the environment only twice: one for x in the conditional expression and one for x in the if branch. In contrast, an ordinary evaluation judgment if x then x else $x \hookrightarrow \text{true}$ would apply a substitution $[\text{true}/x]x$ three times, including the case for x in the else branch (which is unnecessary after all).

Now let us develop the rules for the environment evaluation judgment. We begin with the following (innocent-looking) set of rules:

$$\frac{x \hookrightarrow v \in \eta}{\eta \vdash x \hookrightarrow v} \text{ Var}_e \qquad \frac{}{\eta \vdash \lambda x{:}A.\, e \hookrightarrow \lambda x{:}A.\, e} \text{ Lam}_e$$

$$\frac{\eta \vdash e_1 \hookrightarrow \lambda x{:}A.\, e \quad \eta \vdash e_2 \hookrightarrow v_2 \quad \eta, x \hookrightarrow v_2 \vdash e \hookrightarrow v}{\eta \vdash e_1\, e_2 \hookrightarrow v} \text{ App}_e$$

$$\frac{}{\eta \vdash \text{true} \hookrightarrow \text{true}} \text{ True}_e \qquad \frac{}{\eta \vdash \text{false} \hookrightarrow \text{false}} \text{ False}_e$$

$$\frac{\eta \vdash e \hookrightarrow \text{true} \quad \eta \vdash e_1 \hookrightarrow v}{\eta \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \hookrightarrow v} \text{ If}_{\text{truee}} \qquad \frac{\eta \vdash e \hookrightarrow \text{false} \quad \eta \vdash e_2 \hookrightarrow v}{\eta \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \hookrightarrow v} \text{ If}_{\text{falsee}}$$

The rule Var_e accesses environment η to retrieve the value associated with variable x. The third premise of the rule App_e augments environment η with $x \hookrightarrow v_2$ before starting to evaluating expression e.

It turns out, however, that two of these rules are faulty! (Which ones?) In order to identify the source of the problem, let us evaluate

$$(\lambda x{:}\text{bool}.\, \lambda y{:}\text{bool}.\, \text{if } x \text{ then } y \text{ else false}) \text{ true}$$

---

[1]Note the use of the turnstile symbol $\vdash$. Like a typing judgment $\Gamma \vdash e : A$, an environment evaluation judgment is an example of a hypothetical judgment in which $x \hookrightarrow v$ in η has exactly the same meaning as in an ordinary evaluation judgment $x \hookrightarrow v$, but is used as a hypothesis. Put another way, there is a good reason for using the syntax $x \hookrightarrow v$ for elements of environments.

September 30, 2011

using the environment semantics. The result must be the same closed λ-abstraction that the following evaluation judgment yields:

$$(\lambda x\!:\!\mathsf{bool}.\ \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false})\ \mathsf{true} \hookrightarrow \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}$$

To simplify the presentation, let us instead evaluate f true under the following environment:

$$\eta \;=\; f \hookrightarrow \lambda x\!:\!\mathsf{bool}.\ \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}$$

Then we expect that the following judgment holds:

$$\eta \vdash f\ \mathsf{true} \hookrightarrow \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}$$

The judgment, however, does not hold because f true evaluates to a λ-abstraction with a free variable x in it:

$$\frac{\begin{array}{l} \eta \vdash f \hookrightarrow \lambda x\!:\!\mathsf{bool}.\ \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false} \\ \eta \vdash \mathsf{true} \hookrightarrow \mathsf{true} \\ \eta, x \hookrightarrow \mathsf{true} \vdash \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false} \hookrightarrow \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false} \end{array}}{\eta \vdash f\ \mathsf{true} \hookrightarrow \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}}\ \mathbf{App_e}$$

Why does the resultant λ-abstraction contain a free variable x in it? The reason is that the rule $\mathbf{Lam_e}$ (which is used by the third premise in the above derivation) fails to take into account the fact that values for all free variables in $\lambda x\!:\!A.\ e$ are stored in a given environment. Thus the result of evaluating $\lambda x\!:\!A.\ e$ under environment η should be not just $\lambda x\!:\!A.\ e$, but $\lambda x\!:\!A.\ e$ *together with* additional information on values for free variables in $\lambda x\!:\!A.\ e$, which is precisely the environment η itself! We write a pair of $\lambda x\!:\!A.\ e$ and η as $[\eta, \lambda x\!:\!A.\ e]$, which is called a closure because the presence of η turns $\lambda x\!:\!A.\ e$ into a closed expression. Accordingly we redefine the set of values and fix the rule $\mathbf{Lam_e}$ as follows:

$$\begin{array}{llll} \text{value} & v & ::= & [\eta, \lambda x\!:\!A.\ e] \mid \mathsf{true} \mid \mathsf{false} \end{array}$$

$$\frac{}{\eta \vdash \lambda x\!:\!A.\ e \hookrightarrow [\eta, \lambda x\!:\!A.\ e]}\ \mathbf{Lam_e}$$

Now values are always closed. Note that e and v in $\eta \vdash e \hookrightarrow v$ no longer belong to the same syntactic category, since v may contain closures. That is, a value v as defined above is not necessarily an expression. In contrast, e and v in $e \hookrightarrow v$ belong to the same syntactic category, namely expressions, since neither e nor v contains closures.

Now that its first premise yields a closure, the rule $\mathbf{App_e}$ also needs to be fixed. Suppose that e_1 evaluates to $[\eta', \lambda x\!:\!A.\ e]$ and e_2 to v_2. Since η' contains values for all free variables in $\lambda x\!:\!A.\ e$, we augment η' with $x \hookrightarrow v_2$ to obtain an environment containing values for all free variables in e. Thus we evaluate e under $\eta', x \hookrightarrow v_2$:

$$\frac{\eta \vdash e_1 \hookrightarrow [\eta', \lambda x\!:\!A.\ e] \quad \eta \vdash e_2 \hookrightarrow v_2 \quad \eta', x \hookrightarrow v_2 \vdash e \hookrightarrow v}{\eta \vdash e_1\ e_2 \hookrightarrow v}\ \mathbf{App_e}$$

Note that the environment η under which $\lambda x\!:\!A.\ e$ is obtained is not used in evaluating e.

With the new definition of the rules $\mathbf{Lam_e}$ and $\mathbf{App_e}$, f true evaluates to a closure equivalent to $\lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}$. The following derivation uses an environment η defined as $f \hookrightarrow [\cdot, \lambda x\!:\!\mathsf{bool}.\ \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}]$.

$$\frac{\begin{array}{l} \eta \vdash f \hookrightarrow [\cdot, \lambda x\!:\!\mathsf{bool}.\ \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}] \\ \eta \vdash \mathsf{true} \hookrightarrow \mathsf{true} \\ x \hookrightarrow \mathsf{true} \vdash \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false} \hookrightarrow [x \hookrightarrow \mathsf{true}, \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}] \end{array}}{\eta \vdash f\ \mathsf{true} \hookrightarrow [x \hookrightarrow \mathsf{true}, \lambda y\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}]}\ \mathbf{App_e}$$

In order to show the correctness of the environment semantics, we define two mutually recursive mappings * and $@$:

$$
\begin{aligned}
[\eta, \lambda x\!:\!A.\, e]^\star &= (\lambda x\!:\!A.\, e) \, @ \, \eta \\
\text{true}^\star &= \text{true} \\
\text{false}^\star &= \text{false}
\end{aligned}
\qquad
\begin{aligned}
e \, @ \, \cdot &= e \\
e \, @ \, \eta, x \hookrightarrow v &= [v^\star/x](e \, @ \, \eta)
\end{aligned}
$$

* takes a value v to convert it into a corresponding closed value in the original simply typed λ-calculus. $@$ takes an expression e and an environment η to replace each free variable x in e by $v^\star$ if $x \hookrightarrow v$ is in η; that is, it applies to e those postponed substitutions represented by η.

The following propositions state the correctness of the environment semantics:

Proposition 9.8. *If $\eta \vdash e \hookrightarrow v$, then $e \, @ \, \eta \hookrightarrow v^\star$.*

Proposition 9.9. *If $e \hookrightarrow v$, then $\cdot \vdash e \hookrightarrow v'$ and $v'^\star = v$.*

In order to simplify their proofs, we introduce an equivalence relation $\equiv_{\mathsf{c}}$:

Definition 9.10.
$v \equiv_{\mathsf{c}} v'$ *if and only if $v^\star = v'^\star$.*
$\eta \equiv_{\mathsf{c}} \eta'$ *if and only if $x \hookrightarrow v \in \eta$ means $x \hookrightarrow v' \in \eta'$ such that $v \equiv_{\mathsf{c}} v'$, and vice versa.*

Intuitively $v \equiv_{\mathsf{c}} v'$ means that v and v' (which may contain closures) represent the same value in the simply typed λ-calculus.

Lemma 9.11.

$$
\begin{aligned}
(e \, e') \, @ \, \eta &= (e \, @ \, \eta) \, (e' \, @ \, \eta) \\
(\lambda x\!:\!A.\, e) \, @ \, \eta &= \lambda x\!:\!A.\, (e \, @ \, \eta) \\
(\text{if } e \text{ then } e_1 \text{ else } e_2) \, @ \, \eta &= \text{if } e \, @ \, \eta \text{ then } e_1 \, @ \, \eta \text{ else } e_2 \, @ \, \eta
\end{aligned}
$$

Proof of Proposition 9.8. By rule induction on the judgment $\eta \vdash e \hookrightarrow v$. □

Lemma 9.12. *If $\eta \vdash e \hookrightarrow v$ and $\eta \equiv_{\mathsf{c}} \eta'$, then $\eta' \vdash e \hookrightarrow v'$ and $v \equiv_{\mathsf{c}} v'$.*

Lemma 9.13. *If $\eta \vdash [v/x]e \hookrightarrow v'$, then $\eta, x \hookrightarrow v \vdash e \hookrightarrow v''$ and $v' \equiv_{\mathsf{c}} v''$.*

Lemma 9.14. *If $\eta \vdash e \, @ \, \eta' \hookrightarrow v$, then $\eta, \eta' \vdash e \hookrightarrow v'$ and $v \equiv_{\mathsf{c}} v'$.*

Proof of Proposition 9.9. By rule induction on the judgment $e \hookrightarrow v$. □

9.3 Abstract machine E

The environment evaluation judgment $\eta \vdash e \hookrightarrow v$ exploits environments and closures to dispense with substitutions when evaluating expressions. Still, however, it is not suitable for a practical implementation of the operational semantics because a single judgment $\eta \vdash e \hookrightarrow v$ accounts for the entire evaluation of a given expression. This section develops an abstract machine E which, like the abstract machine C, is based on a reduction judgment (derived from the environment evaluation judgment), and, unlike the abstract machine C, makes no use of substitutions.

As in the abstract machine C, there are two kinds of states in the abstract machine E. The key difference is that the state analyzing a given expression now requires an environment; the definition of stack is also slightly different because of the use of environments:

- $\sigma \blacktriangleright e \, @ \, \eta$ means that the machine is currently analyzing e under the environment η. In order to evaluate a variable in e, we look up the environment η.

- $\sigma \blacktriangleleft v$ means that the machine is currently returning v to the stack σ. We do not need an environment for v because the evaluation of v has been finished.

If an expression e evaluates to a value v, the initial state of the machine would be $\square \blacktriangleright e \,@\, \cdot$ and the final state $\square \blacktriangleleft v$ where $\square$ denotes an empty stack.

The formal definition of the abstract machine E is given as follows:

$$
\begin{array}{rrcl}
\text{value} & v & ::= & [\eta, \lambda x\!:\!A.\, e] \mid \text{true} \mid \text{false} \\
\text{environment} & \eta & ::= & \cdot \mid \eta, x \hookrightarrow v \\
\text{frame} & \phi & ::= & \square_\eta\, e \mid [\eta, \lambda x\!:\!A.\, e]\, \square \mid \text{if } \square_\eta \text{ then } e_1 \text{ else } e_2 \\
\text{stack} & \sigma & ::= & \square \mid \sigma; \phi \\
\text{state} & s & ::= & \sigma \blacktriangleright e \,@\, \eta \mid \sigma \blacktriangleleft v
\end{array}
$$

An important difference from the abstract machine C is that a hole within a frame may now need an environment:

- A frame $\square_\eta\, e$ indicates that an application $e'\, e$ is being reduced and that the environment under which to evaluate $e'\, e$ is η. Hence, after finishing the reduction of e', we reduce e under environment η.

- A frame if $\square_\eta$ then e_1 else e_2 indicates that a conditional construct if e then e_1 else e_2 is being reduced and that the environment under which to evaluate if e then e_1 else e_2 is η. Hence, after finishing the reduction of e, we reduce either e_1 or e_2 (depending on the result of reducing e) under environment η.

Then why do we not need an environment in a frame $[\eta, \lambda x\!:\!A.\, e]\, \square$? Recall from the rule $\mathbf{App_e}$ that after evaluating e_1 to $[\eta, \lambda x\!:\!A.\, e]$ and e_2 to v_2, we evaluate e under an environment $\eta, x \hookrightarrow v_2$. Thus η inside the closure $[\eta, \lambda x\!:\!A.\, e]$ is the environment to be used after finishing the reduction of whatever expression is to fill the hole $\square$, and there is no need to annotate $\square$ with another environment.

With this intuition in mind, we are now ready to develop the reduction rules for the abstract machine E. We use a reduction judgment $s \mapsto_{\mathsf{E}} s'$ for a state transition; we write $\mapsto_{\mathsf{E}}^*$ for the reflexive and transitive closure of $\mapsto_{\mathsf{E}}$. Pay close attention to the use of an environment $\eta, x \hookrightarrow v$ in the rule App_{E}.

$$
\dfrac{x \hookrightarrow v \in \eta}{\sigma \blacktriangleright x \,@\, \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleleft v} \; Var_{\mathsf{E}} \qquad \dfrac{}{\sigma \blacktriangleright \lambda x\!:\!A.\, e \,@\, \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleleft [\eta, \lambda x\!:\!A.\, e]} \; Closure_{\mathsf{E}}
$$

$$
\dfrac{}{\sigma \blacktriangleright e_1\, e_2 \,@\, \eta \mapsto_{\mathsf{E}} \sigma; \square_\eta\, e_2 \blacktriangleright e_1 \,@\, \eta} \; Lam_{\mathsf{E}} \qquad \dfrac{}{\sigma; \square_\eta\, e_2 \blacktriangleleft [\eta', \lambda x\!:\!A.\, e] \mapsto_{\mathsf{E}} \sigma; [\eta', \lambda x\!:\!A.\, e]\, \square \blacktriangleright e_2 \,@\, \eta} \; Arg_{\mathsf{E}}
$$

$$
\dfrac{}{\sigma; [\eta, \lambda x\!:\!A.\, e]\, \square \blacktriangleleft v \mapsto_{\mathsf{E}} \sigma \blacktriangleright e \,@\, \eta, x \hookrightarrow v} \; App_{\mathsf{E}}
$$

$$
\dfrac{}{\sigma \blacktriangleright \text{true} \,@\, \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleleft \text{true}} \; True_{\mathsf{E}} \qquad \dfrac{}{\sigma \blacktriangleright \text{false} \,@\, \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleleft \text{false}} \; False_{\mathsf{E}}
$$

$$
\dfrac{}{\sigma \blacktriangleright \text{if } e \text{ then } e_1 \text{ else } e_2 \,@\, \eta \mapsto_{\mathsf{E}} \sigma; \text{if } \square_\eta \text{ then } e_1 \text{ else } e_2 \blacktriangleright e \,@\, \eta} \; If_{\mathsf{E}}
$$

$$
\dfrac{}{\sigma; \text{if } \square_\eta \text{ then } e_1 \text{ else } e_2 \blacktriangleleft \text{true} \mapsto_{\mathsf{E}} \sigma \blacktriangleright e_1 \,@\, \eta} \; If_{\mathsf{true}\mathsf{E}}
$$

$$
\dfrac{}{\sigma; \text{if } \square_\eta \text{ then } e_1 \text{ else } e_2 \blacktriangleleft \text{false} \mapsto_{\mathsf{E}} \sigma \blacktriangleright e_2 \,@\, \eta} \; If_{\mathsf{false}\mathsf{E}}
$$

An example of a reduction sequence is shown below:

$$
\begin{array}{lll}
& \Box \blacktriangleright (\lambda x\!:\!\mathsf{bool}.\,\lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false}) \text{ true true @ } \cdot & Lam_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\Box.\text{ true} \blacktriangleright (\lambda x\!:\!\mathsf{bool}.\,\lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false}) \text{ true @ } \cdot & Lam_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\Box.\text{ true};\Box.\text{ true} \blacktriangleright \lambda x\!:\!\mathsf{bool}.\,\lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false @ } \cdot & Closure_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\Box.\text{ true};\Box.\text{ true} \blacktriangleleft [\cdot, \lambda x\!:\!\mathsf{bool}.\,\lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false}] & Arg_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\Box.\text{ true};[\cdot, \lambda x\!:\!\mathsf{bool}.\,\lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false}]\,\Box \blacktriangleright \text{ true @ } \cdot & True_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\Box.\text{ true};[\cdot, \lambda x\!:\!\mathsf{bool}.\,\lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false}]\,\Box \blacktriangleleft \text{ true} & App_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\Box.\text{ true} \blacktriangleright \lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false @ } x \hookrightarrow \text{true} & Closure_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\Box.\text{ true} \blacktriangleleft [x \hookrightarrow \text{true}, \lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false}] & Arg_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;[x \hookrightarrow \text{true}, \lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false}]\,\Box \blacktriangleright \text{ true @ } \cdot & True_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;[x \hookrightarrow \text{true}, \lambda y\!:\!\mathsf{bool}.\,\text{if } x \text{ then } y \text{ else false}]\,\Box \blacktriangleleft \text{ true} & App_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box \blacktriangleright \text{if } x \text{ then } y \text{ else false @ } x \hookrightarrow \text{true}, y \hookrightarrow \text{true} & If_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\text{if } \Box_{x \hookrightarrow \text{true},\, y \hookrightarrow \text{true}} \text{ then } y \text{ else false} \blacktriangleright x \text{ @ } x \hookrightarrow \text{true}, y \hookrightarrow \text{true} & Var_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box;\text{if } \Box_{x \hookrightarrow \text{true},\, y \hookrightarrow \text{true}} \text{ then } y \text{ else false} \blacktriangleleft \text{ true} & If_{\text{trueE}} \\
\mapsto_\mathsf{E} & \Box \blacktriangleright y \text{ @ } x \hookrightarrow \text{true}, y \hookrightarrow \text{true} & Var_\mathsf{E} \\
\mapsto_\mathsf{E} & \Box \blacktriangleleft \text{ true} &
\end{array}
$$

The correctness of the abstract machine E is stated as follows:

Theorem 9.15. $\eta \vdash e \hookrightarrow v$ *if and only if* $\sigma \blacktriangleright e @ \eta \mapsto_\mathsf{E}^* \sigma \blacktriangleleft v$.

9.4 Fixed point construct in the abstract machine E

In Section 5.4, we have seen that a typical functional language based on the call-by-value strategy requires that e in fix $x\!:\!A.\,e$ be a λ-abstraction. In extending the abstraction machine E with the fixed point construct, it is mandatory that e in fix $x\!:\!A.\,e$ be a value (although values other than λ-abstractions or their pairs/tuples for e would not be particularly useful).

Recall the reduction rule for the fixed point construct:

$$\text{fix } x\!:\!A.\,e \mapsto [\text{fix } x\!:\!A.\,e/x]e$$

Since the abstract machine E does not use substitutions, a reduction of fix $x\!:\!A.\,e$ must store $x \hookrightarrow$ fix $x\!:\!A.\,e$ in an environment. Thus we could consider the following reduction rule to incorporate the fixed point construct:

$$\frac{}{\sigma \blacktriangleright \text{fix } x\!:\!A.\,e @ \eta \mapsto_\mathsf{E} \sigma \blacktriangleright e @ \eta, x \hookrightarrow \text{fix } x\!:\!A.\,e} \; Fix_\mathsf{E}$$

Unfortunately the rule Fix_E violates the invariant that an environment associates variables with *values* rather than with general expressions. Since fix $x\!:\!A.\,e$ is not a value, $x \hookrightarrow$ fix $x\!:\!A.\,e$ cannot be a valid element of an environment.

Thus we are led to restrict the fixed point construct to λ-abstractions only. In other words, we consider the fixed point construct of the form fix $f\!:\!A\!\rightarrow\!B.\,\lambda x\!:\!A.\,e$ only. (We use the same idea to allow pairs/tuples of λ-abstractions in the fixed point construct.) Moreover we write fix $f\!:\!A\!\rightarrow\!B.\,\lambda x\!:\!A.\,e$ as fun $f\ x\!:\!A.\,e$ and *regard it as a value*. Then fun $f\ x\!:\!A.\,e$ may be interpreted as follows:

- fun $f\ x\!:\!A.\,e$ denotes a recursive function f with a formal argument x of type A and a body e.

Since fun $f\ x\!:\!A.\,e$ denotes a recursive function, e may contain references to f.

The abstract syntax for the abstract machine E now allows fun $f\ x\!:\!A.\,e$ as an expression and $[\eta, \text{fun } f\ x\!:\!A.\,e]$ as a new form of closure:

$$
\begin{array}{llll}
\text{expression} & e & ::= & \cdots \mid \text{fun } f\ x\!:\!A.\,e \\
\text{value} & v & ::= & \cdots \mid [\eta, \text{fun } f\ x\!:\!A.\,e] \\
\text{frame} & \phi & ::= & \cdots \mid [\eta, \text{fun } f\ x\!:\!A.\,e]\,\Box
\end{array}
$$

A typing rule for fun $f\ x\!:\!A.\ e$ may be obtained as an instance of the rule Fix, but it is also instructive to directly derive the rule according to the interpretation of fun $f\ x\!:\!A.\ e$. Since e may contain references to both f (because f is a recursive function) and x (because x is a formal argument), the typing context for e contains type bindings for both f and x:

$$\frac{\Gamma, f : A \to B, x : A \vdash e : B}{\Gamma \vdash \mathsf{fun}\ f\ x\!:\!A.\ e : A \to B}\ \mathsf{Fun}$$

The reduction rules for fun $f\ x\!:\!A.\ e$ are similar to those for λ-abstractions, except that the rule App_{E}^{R} augments the environment with not only $x \hookrightarrow v$ but also $f \hookrightarrow [\eta, \mathsf{fun}\ f\ x\!:\!A.\ e]$ because f is a recursive function:

$$\frac{}{\sigma \blacktriangleright \mathsf{fun}\ f\ x\!:\!A.\ e\ @\ \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleleft [\eta, \mathsf{fun}\ f\ x\!:\!A.\ e]}\ Closure_{\mathsf{E}}^{R}$$

$$\frac{}{\sigma; \square_{\eta}\ e_2 \blacktriangleleft [\eta', \mathsf{fun}\ f\ x\!:\!A.\ e] \mapsto_{\mathsf{E}} \sigma; [\eta', \mathsf{fun}\ f\ x\!:\!A.\ e]\ \square \blacktriangleright e_2\ @\ \eta}\ Arg_{\mathsf{E}}^{R}$$

$$\frac{}{\sigma; [\eta, \mathsf{fun}\ f\ x\!:\!A.\ e]\ \square \blacktriangleleft v \mapsto_{\mathsf{E}} \sigma \blacktriangleright e\ @\ \eta, f \hookrightarrow [\eta, \mathsf{fun}\ f\ x\!:\!A.\ e], x \hookrightarrow v}\ App_{\mathsf{E}}^{R}$$

9.5 Exercises

Exercise 9.16. Why is it not a good idea to use an environment semantics based on reductions? That is, what is the problem with using a judgment of the form $\eta \vdash e \mapsto e'$?

Exercise 9.17. Extend the abstract machine E for product types and sum types.

Chapter 10

Exceptions and continuations

In the simply typed λ-calculus, a complete reduction of $(\lambda x\!:\!A.\,e)\ v$ to another value v' consists of a sequence of β-reductions. From the perspective of imperative languages, the complete reduction consists of two *local* transfers of control: a function call and a return. We may think of a β-reduction $(\lambda x\!:\!A.\,e)\ v \mapsto [v/x]e$ as initiating a call to $\lambda x\!:\!A.\,e$ with an argument v, and $[v/x]e \mapsto^* v'$ as returning from the call with the result v'.

This chapter investigates two extensions to the simply typed λ-calculus for achieving *non-local* transfers of control. By non-local transfers of control, we mean those reductions that cannot be justified by β-reductions alone. First we briefly consider a primitive form of *exception* which in its mature form enables us to cope with erroneous conditions such as division by zero, pattern match failure, and array boundary error. Exceptions are also an excellent programming aid. For example, if the specification of a program requires a function `foo` that is far from trivial to implement but known to be unused until the late stage of development, we can complete its framework just by declaring `foo` such that its body immediately raises an exception.[1] Then we consider *continuations* which may be thought of as a generalization of evaluation contexts in Chapter 8. The basic idea behind continuations is that evaluation contexts are turned into first-class objects which can be passed as arguments to functions or return values of functions. More importantly, an evaluation context elevated to a first-class object may replace the current evaluation context, thereby achieving a non-local transfer of control.

Continuations in the simply typed λ-calculus are often compared to the goto construct of imperative languages. Like the goto construct, continuations are a powerful control construct whose applications range from a simple optimization of list multiplication (to be discussed in Section 10.2) to an elegant implementation of the machinery for concurrent computations. On the other side of the coin, continuations are often detrimental to code readability and should be used with great care for the same reason that the goto construct is avoided in favor of loop constructs in imperative languages.

Both exceptions and continuations are examples of computational effects, called *control effects*, in that their presence destroys the equivalence between λ-abstractions and mathematical functions. (In comparison, mutable references are often called *store effects*.) As computational effects do not mix well with the lazy reduction strategy, both kinds of control effects are usually built on top of the eager reduction strategy.[2]

10.1 Exceptions

In order to support exceptions in the simply typed λ-calculus, we introduce two new constructs try e with e' and exn:

$$\text{expression} \quad e \quad ::= \quad \cdots \mid \text{try } e \text{ with } e \mid \text{exn}$$

Informally try e with e' starts by evaluating e with an exception handler e'. If e successfully evaluates to a value v, the whole expression also evaluates to the same value v. In this case, e' is never visited and is thus ignored. If the evaluation of e raises an exception by attempting to reduce exn, the exception

---

[1] The exception `Unimplemented` in our programming assignments is a good example.
[2] Haskell uses a separate apparatus called *monad* to deal with computational effects.

handler e' is activated. In this case, the result of evaluating e' serves as the final result of evaluating try e with e'. Note that e' may raise another exception, in which case the new exception propagates to the next try e_{next} with e'_{next} such that e_{next} encloses try e with e'.

Formally the operational semantics is extended with the following reduction rules:

$$\frac{}{\text{exn } e \mapsto \text{exn}} \; Exn \quad \frac{}{(\lambda x\!:\!A.\,e)\,\text{exn} \mapsto \text{exn}} \; Exn'$$

$$\frac{e_1 \mapsto e'_1}{\text{try } e_1 \text{ with } e_2 \mapsto \text{try } e'_1 \text{ with } e_2} \; Try \quad \frac{}{\text{try } v \text{ with } e \mapsto v} \; Try' \quad \frac{}{\text{try exn with } e \mapsto e} \; Try''$$

The rules Exn and Exn' say that whenever an attempt is made to reduce exn, the whole reduction is canceled and exn starts to propagate. For example, the reduction of $((\lambda x\!:\!A.\,e)\,\text{exn})\,e'$ eventually ends up with exn:

$$((\lambda x\!:\!A.\,e)\,\text{exn})\,e' \mapsto \text{exn } e' \mapsto \text{exn}$$

In the rule Try', the reduction bypasses the exception handler e because no exception has been raised. in the rule Try'' the reduction activates the exception handler e because an exception has been raised.

Note that Exn and Exn' are two rules specifically designed for propagating exceptions raised within applications. This implies that for all other kinds of constructs, we have to provide separate rules for propagating exceptions. For example, we need the following rule to handle exceptions raised within conditional constructs:

$$\frac{}{\text{if exn then } e_1 \text{ else } e_2 \mapsto \text{exn}} \; Exn''$$

Exercise 10.1. Assuming the eager reduction strategy, give rules for propagating exceptions raised within those constructs for product types and sum types.

10.2 A motivating example for continuations

A prime example for motivating the development of continuations is a recursive function for list multiplication, *i.e.*, for multiplying all elements in a given list. Let us begin with an SML function implementing list multiplication:

```
fun multiply l =
  let
    fun mult nil = 1
      | mult (n :: l') = n * mult l'
  in
    mult l
  end
```

We wish to optimize `multiply` by exploiting the property that in the presence of a zero in `l`, the return value of `multiply` is also a zero regardless of other elements in `l`. Thus, once we encounter an occurrence of a zero in `l`, we do not have to multiply remaining elements in the list:

```
fun multiply' l =
  let
    fun mult nil = 1
      | mult (0 :: l') = 0
      | mult (n :: l') = n * mult l'
  in
    mult l
  end
```

`multiply'` is definitely an improvement over `multiply`, although if `l` contains no zero, it runs slower than `multiply` because of the cost of comparing each element in `l` with `0`. `multiply'`, however, is not a full optimization of `multiply` exploiting the property of multiplication: due to the recursive

nature of `mult`, it needs to return a zero as many times as the number of elements before the first zero in `l`. Thus an ideal solution would be to exit `mult` altogether after encountering a zero in `l`, *even without returning a zero to previous calls to `mult`*. What makes this possible is two constructs, `callcc` and `throw`, for continuations:[3]

```
fun multiply'' l =
  callcc (fn ret =>
  let
    fun mult nil = 1
      | mult (0 :: l') = throw ret 0
      | mult (n :: l') = n * mult l'
  in
    mult l
  end)
```

Informally `callcc (fn ret => ...` declares a label `ret`, and `throw ret 0` causes a non-local transfer of control to the label `ret` where the evaluation resumes with a value `0`. Hence there occurs no return from `mult` once `throw ret 0` is reached.

Below we give a formal definition of the two constructs `callcc` and `throw`.

10.3 Evaluation contexts as continuations

A continuation is a general concept for describing an "incomplete" computation which yields a "complete" computation only when another computation is *prepended* (or *prefixed*).[4] That is, by joining a computation with a continuation, we obtain a complete computation. A λ-abstraction $\lambda x : A. e$ may be seen as a continuation, since it conceptually takes a computation producing a value v and returns a computation corresponding to $[v/x]e$. Note that $\lambda x : A. e$ itself does not initiate a computation; it is only when an argument v is supplied that it initiates a computation of $[v/x]e$. A better example of continuation is an evaluation context κ which, given an expression e, yields a computation corresponding to $\kappa[\![e]\!]$. Note that like a λ-abstraction, κ itself does not describe a complete computation. In this chapter, we study evaluation contexts as a means of realizing continuations.

Consider the rule Red_β which decomposes a given expression into a unique evaluation context κ and a unique subexpression e:

$$\frac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]} \; Red_\beta$$

Since the decomposition under the rule Red_β is implicit and evaluation contexts are not expressions, there is no way to store κ as an expression. Hence our first goal is to devise a new construct for seizing the current evaluation context.[5] For example, when a given expression is decomposed into κ and e by the rule Red_β, the new construct would return a (new form of) value storing κ. The second goal is to involve such a value in a reduction sequence, as there is no point in creating such a value without using it.

In order to utilize evaluation contexts as continuations in the simply typed λ-calculus, we introduce three new constructs: $\langle \kappa \rangle$, callcc $x. e$, and throw e to e'.

- $\langle \kappa \rangle$ is an expression storing an evaluation context κ; we use angle brackets $\langle \rangle$ to distinguish it as an expression not to be confused with an evaluation context. The only way to generate it is to reduce callcc $x. e$. As a value, $\langle \kappa \rangle$ is called a continuation.

- callcc $x. e$ seizes the current evaluation context κ and stores $\langle \kappa \rangle$ in x before proceeding to reduce e:

$$\frac{}{\kappa[\![\text{callcc } x. e]\!] \mapsto \kappa[\![[\langle \kappa \rangle / x]e]\!]} \; Callcc$$

  In the case that the reduction of e does not use x at all, callcc $x. e$ produces the same result as e.

---

[3] In SML/NJ, open the structure `SMLofNJ.Cont` to test `multiply''`.

[4] Here "prepend" and "prefix" both mean "add to the beginning."

[5] I hate the word *seize* because the *z* sound in it is hard to enunciate. Besides I do not want to remind myself of *Siege* Tanks in Starcraft!

- throw e to e' expects a value v from e and a continuation $\langle \kappa' \rangle$ from e'. Then it starts a reduction of $\kappa'[\![v]\!]$ *regardless of the current evaluation context* κ:

$$\overline{\kappa[\![\text{throw } v \text{ to } \langle \kappa' \rangle]\!] \mapsto \kappa'[\![v]\!]} \;\; \textit{Throw}$$

In general, κ and κ' are unrelated with each other, which implies that the rule *Throw* allows us to achieve a non-local transfer of control. We say that throw v to $\langle \kappa' \rangle$ *throws* a value v to a continuation κ'.

The abstract syntax is extended as follows:

$$
\begin{array}{rcll}
\text{expression} & e & ::= & \cdots \mid \text{callcc } x.\, e \mid \text{throw } e \text{ to } e \mid \langle \kappa \rangle \\
\text{value} & v & ::= & \cdots \mid \langle \kappa \rangle \\
\text{evaluation context} & \kappa & ::= & \cdots \mid \text{throw } \kappa \text{ to } e \mid \text{throw } v \text{ to } \kappa
\end{array}
$$

The use of evaluation contexts throw κ to e and throw v to κ indicates that throw e to e' reduces e before reducing e'.

Exercise 10.2. What is the result of evaluating each expression below?

$$
\begin{array}{rll}
(1) & \text{fst callcc } x.\,(\text{true}, \text{false}) & \mapsto^* \quad ? \\
(2) & \text{fst callcc } x.\,(\text{true}, \text{throw } (\text{false}, \text{false}) \text{ to } x) & \mapsto^* \quad ? \\
(3) & \text{snd callcc } x.\,(\text{throw } (\text{true}, \text{true}) \text{ to } x, \text{false}) & \mapsto^* \quad ?
\end{array}
$$

In the case (1), x is not found in $(\text{true}, \text{false})$, so the expression is equivalent to fst $(\text{true}, \text{false})$. In the case (2), the result of evaluating true is eventually ignored because the reduction of throw $(\text{false}, \text{false})$ to x causes $(\text{false}, \text{false})$ to replace callcc $x.\,(\text{true}, \text{throw } (\text{false}, \text{false}) \text{ to } x)$. Thus, in general, fst callcc $x.\,(e, \text{throw } (\text{false}, e') \text{ to } x)$ evaluates to false regardless of e and e' (provided that the evaluation terminates). In the case (3), false is not even evaluated: before reaching false, the reduction of throw $(\text{true}, \text{true})$ to x causes $(\text{true}, \text{true})$ to replace callcc $x.\,(\text{throw } (\text{true}, \text{true}) \text{ to } x, \text{false})$. Thus, in general, snd callcc $x.\,(\text{throw } (e, \text{true}) \text{ to } x, e')$ evaluates to true regardless of e and e', where e' is never evaluated:

$$
\begin{array}{rll}
(1) & \text{fst callcc } x.\,(\text{true}, \text{false}) & \mapsto^* \quad \text{true} \\
(2) & \text{fst callcc } x.\,(e, \text{throw } (\text{false}, e') \text{ to } x) & \mapsto^* \quad \text{false} \\
(3) & \text{snd callcc } x.\,(\text{throw } (e, \text{true}) \text{ to } x, e') & \mapsto^* \quad \text{true}
\end{array}
$$

Now that we have seen the reduction rules for the new constructs, let us turn our attention to their types. Since $\langle \kappa \rangle$ is a new form of value, we need a new form of type for it. (Otherwise how would we represent its type?) We assign a type A cont to $\langle \kappa \rangle$ if the hole in κ expects a value of type A. That is, if $\kappa : A \Rightarrow C$ holds (see Section 8.2), $\langle \kappa \rangle$ has type A cont:

$$\text{type} \qquad A \quad ::= \quad \cdots \mid A \text{ cont}$$

$$\frac{\kappa : A \Rightarrow C}{\Gamma \vdash \langle \kappa \rangle : A \text{ cont}} \;\; \text{Context}$$

It is important that a type A cont assigned to a continuation $\langle \kappa \rangle$ specifies the type of an expression e to fill the hole in κ, but not the type of the resultant expression $\kappa[\![e]\!]$. For this reason, a continuation is usually said to return an "answer" (of an unknown type) rather than a value of a specific type. For a similar reason, a λ-abstraction serves as a continuation only if it has a designated return type, *e.g.*, *Ans*, denoting "answers."

The typing rules for the other two constructs respect their reduction rules:

$$\frac{\Gamma, x : A \text{ cont} \vdash e : A}{\Gamma \vdash \text{callcc } x.\, e : A} \;\; \text{Callcc} \qquad \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A \text{ cont}}{\Gamma \vdash \text{throw } e_1 \text{ to } e_2 : C} \;\; \text{Throw}$$

The rule Callcc assigns type A cont to x and type A to e *for the same type* A, since if e has type A, then the evaluation context when reducing callcc $x.\, e$ also expects a value of type A for the hole in it. callcc $x.\, e$ has the same type as e because, for example, x may not appear in e at all, in which case callcc $x.\, e$ produces the same result as e. In the rule Throw, it is safe to assign an arbitrary type C to throw e_1 to e_2

because its reduction never finishes: there is no value v such that throw e_1 to $e_2 \mapsto^* v$. In other words, an "answer" can be an arbitrary type.

An important consequence of the rule Callcc is that when evaluating callcc $x.\,x$, a continuation stored in variable x cannot be part of the result of evaluating expression e. For example, callcc $x.\,x$ fails to typecheck because the rule Callcc assigns type A cont to the first x and type A to the second x, but there is no way to unify A cont and A (i.e., A cont $\neq A$). Then how can we pass the continuation stored in variable x to the outside of callcc $x.\,e$? Since there is no way to pass it by evaluating e, the only hope is to throw it to another continuation! (We will see an example in the next section.)

To complete the definition of the three new constructs, we extend the definition of substitution as follows:

$$
\begin{array}{rcll}
[e'/x]\text{callcc } x.\,e & = & \text{callcc } x.\,e & \\
[e'/x]\text{callcc } y.\,e & = & \text{callcc } y.\,[e'/x]e & \text{if } x \neq y, y \notin FV(e') \\
[e'/x]\text{throw } e_1 \text{ to } e_2 & = & \text{throw } [e'/x]e_1 \text{ to } [e'/x]e_2 & \\
[e'/x]\langle \kappa \rangle & = & \langle \kappa \rangle &
\end{array}
$$

Type safety is stated in the same way as in Theorems 8.2 and 8.3.

10.4 Composing two continuations

The goal of this section is to develop a function compose of the following type:

$$\text{compose} : (A \to B) \to B \text{ cont} \to A \text{ cont}$$

Roughly speaking, compose $f \langle \kappa \rangle$ joins an incomplete computation (or just a continuation) described by f with κ to build a new continuation. To be precise, compose $f \langle \kappa \rangle$ returns a continuation κ' such that throwing a value v to κ' has the same effect as throwing $f\, v$ to κ.

Exercise 10.3. Give a definition of compose. You have to solve two problems: how to create a correct continuation by placing callcc $x.\,e$ at the right position and how to return the continuation as the return value of compose.

The key observations are:

- throw v to (compose $f \langle \kappa \rangle$) is operationally equivalent to throw $f\, v$ to $\langle \kappa \rangle$.

- For any evaluation context κ', both throw $f\, v$ to $\langle \kappa \rangle$ and $\kappa'[\![\text{throw } f\, v \text{ to } \langle \kappa \rangle]\!]$ evaluate to the same value. More generally, $\langle \text{throw } f\, \square \text{ to } \langle \kappa \rangle \rangle$ and $\langle \kappa'[\![\text{throw } f\, \square \text{ to } \langle \kappa \rangle]\!] \rangle$ are semantically no different.

Thus we define compose in such a way that compose $f \langle \kappa \rangle$ returns $\langle \text{throw } f\, \square \text{ to } \langle \kappa \rangle \rangle$.

First we replace $\square$ in throw $f\, \square$ to $\langle \kappa \rangle$ by callcc $x.\,\cdots$ to create a continuation $\langle \kappa'[\![\text{throw } f\, \square \text{ to } \langle \kappa \rangle]\!] \rangle$ (for a certain evaluation context κ') which is semantically no different from $\langle \text{throw } f\, \square \text{ to } \langle \kappa \rangle \rangle$:

$$\text{compose} \quad = \quad \lambda f : A \to B.\, \lambda k : B \text{ cont. throw } f\,(\text{callcc } x.\,\cdots) \text{ to } k$$

Then x stores the very continuation that compose $f\, k$ needs to return.

Now how do we return x? Obviously callcc $x.\,\cdots$ cannot return x because x has type A cont while $\cdots$ must have type A, which is strictly smaller than A cont. Therefore the only way to return x from $\cdots$ is to throw it to the continuation starting from the hole in $\lambda f : A \to B.\, \lambda k : B \text{ cont.}\,\square$:

$$
\begin{aligned}
\text{compose} \quad = \quad & \lambda f : A \to B.\, \lambda k : B \text{ cont.} \\
& \quad \text{callcc } y.\, \text{throw } f\,(\text{callcc } x.\, \text{throw } x \text{ to } y) \text{ to } k
\end{aligned}
$$

Note that y has type A cont cont. Since x has type A cont, compose ends up throwing x to a continuation which expects another continuation!

10.5 Exercises

Exercise 10.4. Extend the abstract machine C with new rules for the reduction judgment $s \mapsto_C s'$ so as to support exceptions. Use a new state $\sigma \blacktriangleleft\blacktriangleleft$ exn to means that the machine is currently propagating an exception.

Chapter 11

Subtyping

Subtyping is a fundamental concept in programming language theory. It is especially important in the design of an object-oriented language in which the relation between a superclass and its subclasses may be seen as a subtyping relation. This chapter develops the theory of subtyping by considering various subtyping relations and discussing the semantics of subtyping.

11.1   Principle of subtyping

The *principle of subtyping* is a principle specifying when a type is a *subtype* of another type. It states that A is a subtype of B if an expression of type A may be used wherever an expression of type B is expected. Formally we write $A \leq B$ if A is a subtype of B, or equivalently, if B is a *supertype* of A.

The principle of subtyping justifies two *subtyping rules*, *i.e.*, inference rules for deducing subtyping relations:

$$\frac{}{A \leq A} \; Refl_{\leq} \qquad \frac{A \leq B \quad B \leq C}{A \leq C} \; Trans_{\leq}$$

The rules $Refl_{\leq}$ and $Trans_{\leq}$ express reflexivity and transitivity of the subtyping relation $\leq$, respectively. The *rule of subsumption* is a typing rule which enables us to change the type of an expression to its supertype:

$$\frac{\Gamma \vdash e : A \quad A \leq B}{\Gamma \vdash e : B} \; \mathsf{Sub}$$

It is easy to justify the rule Sub using the principle of subtyping. Suppose $\Gamma \vdash e : A$ and $A \leq B$. Since e has type A (under typing context Γ), the subtyping relation $A \leq B$ allows us to use e wherever an expression of type B is expected, which implies that e is effectively of type B.

There are two kinds of semantics for subtyping: *subset semantics* and *coercion semantics*. Under the subset semantics, $A \leq B$ holds if type A literally constitutes a "subset" of type B. That is, $A \leq B$ holds if a value of type A can also be viewed as a value of type B. The coercion semantics permits $A \leq B$ if there exists a unique method to convert values of type A to values of type B.

As an example, consider three base type nat for natural numbers, int for integers, and float for floating point numbers:

$$\text{base type} \qquad P \quad ::= \quad \text{nat} \mid \text{int} \mid \text{float}$$

If nat and int use the same representation, say, a 32-bit word, a value of type nat also represents an integer of type int. Hence a natural number of type nat can be viewed as an integer of type int, which means that nat $\leq$ int holds under the subset semantics. If float uses a 64-bit word to represent a floating point number, a value of type int is not a special value of type float because int uses a representation

incompatible with 64-bit floating point numbers. Thus int is not a subtype of float under the subset semantics, even though integers are a subset of floating point numbers in mathematics. Under the coercion semantics, however, int $\leq$ float holds if there is a function, *e.g.*, int2float, converting 32-bit integers to 64-bit floating point numbers.

In the next section, we will assume the subset semantics which does not alter the operational semantics and is simpler than the coercion semantics. We will discuss the coercion semantics in detail in Section 11.3

11.2   Subtyping relations

To explain subtyping relations on various types, let us assume two base types nat and int, and a subtyping relation nat $\leq$ int:

$$
\begin{array}{lllll}
\text{type} & A & ::= & P \mid A \to A \mid A \times A \mid A + A \mid \text{ref } A \\
\text{base type} & P & ::= & \text{nat} \mid \text{int}
\end{array}
$$

A subtyping relation on two product types tests the relation between corresponding components:

$$
\frac{A \leq A' \quad B \leq B'}{A \times B \leq A' \times B'} \; Prod_{\leq}
$$

For example, nat $\times$ nat is a subtype of int $\times$ int:

$$
\frac{\text{nat} \leq \text{int} \quad \text{nat} \leq \text{int}}{\text{nat} \times \text{nat} \leq \text{int} \times \text{int}} \; Prod_{\leq}
$$

Intuitively a pair of natural numbers can be viewed as a pair of integers because a natural number is a special form of integer. Similarly we can show that a subtyping relation on two sum types tests the relation between corresponding components as follows:

$$
\frac{A \leq A' \quad B \leq B'}{A + B \leq A' + B'} \; Sum_{\leq}
$$

The subtyping rule for function types requires a bit of thinking. Consider two functions $f : A \to \text{nat}$ and $f' : A \to \text{int}$. An application of f' to an expression e of type A has type int (under a certain typing context Γ):

$$
\Gamma \vdash f' \, e : \text{int}
$$

If we replace f' by f, we get an application of type nat, but by the rule of subsumption, the resultant application can be assigned type int as well:

$$
\frac{\Gamma \vdash f \, e : \text{nat} \quad \text{nat} \leq \text{int}}{\Gamma \vdash f \, e : \text{int}} \; Sub
$$

Therefore, for the purpose of typechecking, it is always safe to use f wherever f' is expected, which implies that $A \to \text{nat}$ is a subtype of $A \to \text{int}$. The converse, however, does not hold because there is no assumption of int $\leq$ nat. The result is generalized to the following subtyping rule:

$$
\frac{B \leq B'}{A \to B \leq A \to B'} \; Fun'_{\leq}
$$

The rule $Fun'_{\leq}$ says that subtyping on function types is *covariant* in return types in the sense that the premise places the two return types in the same direction as in the conclusion (*i.e.*, left B and right B'). Then we can say that by the rules $Prod_{\leq}$ and $Sum_{\leq}$, subtyping on product types and sum types is also covariant in both components.

Now consider two functions $g : \mathsf{nat} \to A$ and $g' : \mathsf{int} \to A$. Perhaps surprisingly, $\mathsf{nat} \to A$ is *not* a subtype of $\mathsf{int} \to A$ whereas $\mathsf{int} \to A$ is a subtype of $\mathsf{nat} \to A$. To see why, let us consider an application of g to an expression e of type nat (under a certain typing context Γ):

$$\frac{\Gamma \vdash g : \mathsf{nat} \to A \quad \Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash g\ e : A} \to \mathsf{E}$$

Since e can be assigned type int by the rule of subsumption, replacing g by g' does not change the type of the application:

$$\frac{\Gamma \vdash g' : \mathsf{int} \to A \quad \dfrac{\Gamma \vdash e : \mathsf{nat} \quad \mathsf{nat} \le \mathsf{int}}{\Gamma \vdash e : \mathsf{int}}\ \mathsf{Sub}}{\Gamma \vdash g'\ e : A} \to \mathsf{E}$$

Therefore $\mathsf{int} \to A$ is a subtype of $\mathsf{nat} \to A$. To see why the converse does not hold, consider another application of g' to an expression e' of type int:

$$\frac{\Gamma \vdash g' : \mathsf{int} \to A \quad \Gamma \vdash e' : \mathsf{int}}{\Gamma \vdash g'\ e' : A} \to \mathsf{E}$$

If we replace g' by g, the resultant application does not even typecheck because e' cannot be assigned type nat:

$$\frac{\Gamma \vdash g : \mathsf{nat} \to A \quad \dfrac{\Gamma \vdash e' : \mathsf{int}}{\Gamma \vdash e' : \mathsf{nat}}\ \text{???}}{\Gamma \vdash g\ e' : A} \to \mathsf{E}$$

We generalize the result to the following subtyping rule:

$$\frac{A' \le A}{A \to B \le A' \to B}\ \mathit{Fun}''_{\le}$$

The rule $\mathit{Fun}''_{\le}$ says that subtyping on function types is *contravariant* in argument types in the sense that the premise reverses the position of the two argument types from the conclusion (*i.e.*, left A' and right A).

We combine the rules $\mathit{Fun}'_{\le}$ and $\mathit{Fun}''_{\le}$ into the following subtyping rule:

$$\boxed{\ \frac{A' \le A \quad B \le B'}{A \to B \le A' \to B'}\ \mathit{Fun}_{\le}\ }$$

The rule $\mathit{Fun}_{\le}$ says that subtyping on function types is contravariant in argument types and covariant in return types.

Subtyping on reference types is unusual in that it is neither covariant nor contravariant. Let us figure out the relation between two types A and B when $\mathsf{ref}\ A \le \mathsf{ref}\ B$ holds:

$$\frac{\text{???}}{\mathsf{ref}\ A \le \mathsf{ref}\ B}\ \mathit{Ref}_{\le}$$

Suppose that an expression e has type $\mathsf{ref}\ A$ and another expression e' type $\mathsf{ref}\ B$. By the principle of subtyping, we should be able to use e wherever e' is used. Since e' has a reference type, there are two ways of using e': dereferencing it and assigning a new value to it.

As an example of the first case, consider a well-typed expression $f\ (!e')$ where f has type $B \to C$ for some type C. By the assumption $\mathsf{ref}\ A \le \mathsf{ref}\ B$, expression $f\ (!e)$ is also well-typed. Since f has type $B \to C$ and $!e$ has type A, the type of $!e$ changes from A to B, which implies $A \le B$. As an example of the second case, consider a well-typed expression $e' := v$ where v has type B. By the assumption $\mathsf{ref}\ A \le \mathsf{ref}\ B$, expression $e := v$ is also well-typed. Since v has type B and e has type $\mathsf{ref}\ A$, the type of v changes from B to A, which implies $B \le A$. These two observations lead to the following subtyping rule for reference types:

$$\frac{A \leq B \quad B \leq A}{\mathsf{ref}\ A \leq \mathsf{ref}\ B}\ Ref_{\leq}$$

Thus we say that subtyping on reference types is *non-variant* (*i.e.*, neither covariant nor contravariant).

Another way of looking at the rule $Ref_{\leq}$ is to interpret $\mathsf{ref}\ A$ as an abbreviation of a function type. We may think of $\mathsf{ref}\ A$ as $? \to A$ for some unknown type $?$ because dereferencing an expression of type $\mathsf{ref}\ A$ requires no additional argument (hence $? \to$) and returns an expression of type A (hence $\to A$). Therefore $\mathsf{ref}\ A \leq \mathsf{ref}\ B$ implies $? \to A \leq ? \to B$, which in turn implies $A \leq B$ by the rule $Fun_{\leq}$. We may also think $\mathsf{ref}\ A$ as $A \to \mathsf{unit}$ because assigning a new value to an expression of type $\mathsf{ref}\ A$ requires a value of type A (hence $A \to$) and returns a unit (hence $\to \mathsf{unit}$). Therefore $\mathsf{ref}\ A \leq \mathsf{ref}\ B$ implies $A \to \mathsf{unit} \leq B \to \mathsf{unit}$, which in turn implies $B \leq A$ by the rule $Fun_{\leq}$.

While we have not investigated array types, subtyping on array types follows the same pattern as subtyping on reference types, since an array, like a reference, allows both read and write operations on it. If we use an array type $\mathsf{array}\ A$ for arrays of elements of type A, we obtain the following subtyping rule:

$$\frac{A \leq B \quad B \leq A}{\mathsf{array}\ A \leq \mathsf{array}\ B}\ Array_{\leq}$$

Interestingly the Java language adopts a subtyping rule in which subtyping on array rules is covariant in element types:

$$\frac{A \leq B}{\mathsf{array}\ A \leq \mathsf{array}\ B}\ Array_{\leq}'$$

While it is controversial whether the rule $Array_{\leq}'$ is a flaw in the design of the Java language, using the rule $Array_{\leq}'$ for subtyping on array types incurs a runtime overhead which would otherwise be unnecessary. To be specific, lack of the condition $B \leq A$ in the premise implies that whenever a value of type B is written to an array of type $\mathsf{array}\ A$, the runtime system must verify a subtyping relation $B \leq A$, which incurs a runtime overhead of dynamic tag-checks.

11.3 Coercion semantics for subtyping

Under the coercion semantics, a subtyping relation $A \leq B$ holds if there exists a unique method to convert values of type A to values of type B. As a witness to the existence of such a method, we usually use a λ-abstraction, called a *coercion function*, of type $A \to B$. We use a *coercion subtyping judgment*

$$A \leq B \Rightarrow f$$

to mean that $A \leq B$ holds under the coercion semantics with a coercion function f of type $A \to B$. For example, a judgment $\mathsf{int} \leq \mathsf{float} \Rightarrow \mathsf{int2float}$ holds if a coercion function $\mathsf{int2float}$ converts integers of type int to floating point number of type float.

The subtyping rules for the coercion subtyping judgment are given as follows:

$$\frac{}{A \leq A \Rightarrow \lambda x{:}A.\,x}\ Refl_{\leq}^{\mathsf{C}} \qquad \frac{A \leq B \Rightarrow f \quad B \leq C \Rightarrow g}{A \leq C \Rightarrow \lambda x{:}A.\,g\,(f\,x)}\ Trans_{\leq}^{\mathsf{C}}$$

$$\frac{A \leq A' \Rightarrow f \quad B \leq B' \Rightarrow g}{A \times B \leq A' \times B' \Rightarrow \lambda x{:}A \times B.\,(f\,(\mathsf{fst}\,x), g\,(\mathsf{snd}\,x))}\ Prod_{\leq}^{\mathsf{C}}$$

$$\frac{A \leq A' \Rightarrow f \quad B \leq B' \Rightarrow g}{A{+}B \leq A'{+}B' \Rightarrow \lambda x{:}A{+}B.\,\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ y_1.\,\mathsf{inl}_{B'}\,(f\,y_1)\ |\ \mathsf{inr}\ y_2.\,\mathsf{inr}_{A'}\,(g\,y_2)}\ Sum_{\leq}^{\mathsf{C}}$$

$$\frac{A' \leq A \Rightarrow f \quad B \leq B' \Rightarrow g}{A \to B \leq A' \to B' \Rightarrow \lambda h{:}A \to B.\,\lambda x{:}A'.\,g\,(h\,(f\,x))}\ Fun_{\leq}^{\mathsf{C}}$$

Unlike the subset semantics which does not change the operational semantics, the coercion seman-tics affects the way that expressions are evaluated. Suppose that we are evaluating a well-typed expres-sion e with the following typing derivation which uses a coercion subtyping judgment:

$$\frac{\Gamma \vdash e : A \quad A \leq B \Rightarrow f}{\Gamma \vdash e : B} \ \mathsf{Sub}^{\mathsf{C}}$$

Since the rule $\mathsf{Sub}^{\mathsf{C}}$ tacitly promotes the type of e from A to B, we do not have to insert an explicit call to the coercion function f to make e typecheck. The result of evaluating e, however, is correct only if an explicit call to f is made after evaluating e. For example, if e is an argument to another function g of type $B \to C$ (for some type C), $g\ e$ certainly typechecks but may go wrong at runtime. Therefore the type system inserts an explicit call to a coercion function each time the rule $\mathsf{Sub}^{\mathsf{C}}$ is used. In the above case, the type system replaces e by $f\ e$ after typechecking e using the rule $\mathsf{Sub}^{\mathsf{C}}$.

A potential problem with the coercion semantics is that the same subtyping relation may have sev-eral coercion functions which all have the same type but exhibit different behavior. As an example, consider the following subtyping relations:

$$\mathsf{int} \leq \mathsf{float} \Rightarrow \mathsf{int2float} \quad \mathsf{int} \leq \mathsf{string} \Rightarrow \mathsf{int2string} \quad \mathsf{float} \leq \mathsf{string} \Rightarrow \mathsf{float2string}$$

$\mathsf{int} \leq \mathsf{string}$ and $\mathsf{float} \leq \mathsf{string}$ imply that integers and floating point numbers are automatically con-verted to strings in all contexts expecting strings. Then the same subtyping judgment $\mathsf{int} \leq \mathsf{string}$ has two coercions functions: $\mathsf{int2string}$ and $\lambda x : \mathsf{int}.\ \mathsf{float2string}\ (\mathsf{int2float}\ x)$. The two coercion functions, how-ever, behave differently. For example, the first converts 0 to `"0"` whereas the second converts 0 to `"0.0"`.

We say that a type system for subtypes is *coherent* if all coercion functions for the same subtyping relation exhibit the same behavior. In the example above, we can recover coherence by specifying that float2string converts 0.0 to `"0"`, instead of `"0.0"`, and similarly for all other forms of floating point numbers. We do not further discuss coherence which is difficult to prove for more complex type systems.

Chapter 12

Recursive Types

In programming in a practical functional language, there often arises a need for *recursive data structures* (or *inductive data structures*) whose components are data structures of the same kind but of smaller size. For example, a tree is a recursive data structure because children of the root node are smaller trees of the same kind. We may even think of natural numbers as a recursive data structure because a non-zero natural number can be expressed as a successor of another natural number.

The type system developed so far, however, cannot account for recursive data structures. Intuitively types for recursive data structures require recursive definitions at the level of types, but the previous type system does not provide such a language construct. (Recursive definitions at the level of expressions can be expressed using the fixed point construct.) This chapter introduces a new language construct for declaring *recursive types* which express recursive definitions at the level of types.

With recursive types, we can declare types for recursive data structures. For example, we declare a recursive type ntree for binary trees of natural numbers (of type nat) with the following recursive definition:

$$\mathsf{ntree} \quad \cong \quad \mathsf{nat} + (\mathsf{ntree} \times \mathsf{ntree})$$

The definition says that ntree is either a single natural number of type nat (corresponding to leaf nodes) or two such binary trees of type ntree (corresponding to internal nodes).

There are two approaches to formalizing recursive types: *equi-recursive* and *iso-recursive* approaches which differ in the interpretation of $\cong$ in recursive definitions of types. Under the equi-recursive approach, $\cong$ stands for an equality relation. For example, the recursive definition of ntree specifies that ntree and $\mathsf{nat} + (\mathsf{ntree} \times \mathsf{ntree})$ are equal and thus interchangeable: ntree is automatically (*i.e.*, without the intervention of programmers) converted to $\mathsf{nat} + (\mathsf{ntree} \times \mathsf{ntree})$ and vice versa whenever necessary to make a given expression typecheck. Under the iso-recursive approach, $\cong$ stands for an isomorphic relation: two types in a recursive definition cannot be identified, but can be converted to each other by certain functions. For example, the recursive definition of ntree implicitly declares two functions for converting between ntree and $\mathsf{nat} + (\mathsf{ntree} \times \mathsf{ntree})$:

$$
\begin{aligned}
\mathsf{fold}_{\mathsf{ntree}} &\quad : \quad \mathsf{nat} + (\mathsf{ntree} \times \mathsf{ntree}) \rightarrow \mathsf{ntree} \\
\mathsf{unfold}_{\mathsf{ntree}} &\quad : \quad \mathsf{ntree} \rightarrow \mathsf{nat} + (\mathsf{ntree} \times \mathsf{ntree})
\end{aligned}
$$

To create a value of type ntree, we first create a value of type $\mathsf{nat} + (\mathsf{ntree} \times \mathsf{ntree})$ and then apply function $\mathsf{fold}_{\mathsf{ntree}}$; to analyze a value of type ntree, we first apply function $\mathsf{unfold}_{\mathsf{ntree}}$ and then analyze the resultant value using a case expression.

Below we formalize recursive types under the iso-recursive approach. We will also see that SML uses the iso-recursive approach to deal with datatype declarations.

12.1 Definition

Consider the recursive definition of ntree. We may think of ntree as the solution to the following equation where α is a *type variable* standing for "any type" as in SML:

$$\alpha \quad \cong \quad \mathsf{nat} + (\alpha \times \alpha)$$

Since substituting ntree for α yields the original recursive definition of ntree, ntree is indeed the solution to the above equation. We choose to write $\mu\alpha.\mathsf{nat}+(\alpha \times \alpha)$ for the solution to the above equation where α is a fresh type variable. Then we can redefine ntree as follows:

$$\mathsf{ntree} \quad = \quad \mu\alpha.\mathsf{nat}+(\alpha \times \alpha)$$

Generalizing the example of ntree, we use a recursive type $\mu\alpha.A$ for the solution to the equation $\alpha \cong A$ where A may contain occurrences of type variable α:

$$\mathsf{type} \qquad A \quad ::= \quad \cdots \mid \alpha \mid \mu\alpha.A$$

The intuition is that $C = \mu\alpha.A$ means $C \cong [C/\alpha]A$. For example, $\mathsf{ntree} = \mu\alpha.\mathsf{nat}+(\alpha \times \alpha)$ means $\mathsf{ntree} \cong \mathsf{nat}+(\mathsf{ntree} \times \mathsf{ntree})$. Since $\mu\alpha.A$ declares a fresh type variable α which is valid only within A, not every recursive type qualifies as a valid type. For example, $\mu\alpha.\alpha+\beta$ is not a valid recursive type unless it is part of another recursive type declaring type variable β. In order to be able to check the validity of a given recursive type, we define a typing context as an ordered set of type bindings and type declarations:

$$\text{typing context} \qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A \mid \Gamma, \alpha \text{ type}$$

We use a new judgment $\Gamma \vdash A$ type, called a *type judgment*, to check that A is a valid type under typing context Γ:

$$\frac{\alpha \text{ type} \in \Gamma}{\Gamma \vdash \alpha \text{ type}} \text{ TyVar} \qquad \frac{\Gamma, \alpha \text{ type} \vdash A \text{ type}}{\Gamma \vdash \mu\alpha.A \text{ type}} \text{ Ty}\mu$$

Given a recursive type $C = \mu\alpha.A$, we need to be able to convert $[C/\alpha]A$ to C and vice versa so as to create or analyze a value of type C. Thus, under the iso-recursive approach, a declaration of a recursive type $C = \mu\alpha.A$ implicitly introduces two primitive constructs fold_C and unfold_C specialized for type C. Operationally we may think of fold_C and unfold_C as behaving like functions of the following types:

$$\begin{aligned}\mathsf{fold}_C \quad &: \quad [C/\alpha]A \to C \\ \mathsf{unfold}_C \quad &: \quad C \to [C/\alpha]A\end{aligned}$$

As fold_C and unfold_C are actually not functions but primitive constructs which always require an additional expression as an argument (*i.e.*, we cannot treat fold_C as a first-class object), the abstract syntax is extended as follows:

$$\begin{aligned}\text{expression} \quad e \quad &::= \quad \cdots \mid \mathsf{fold}_C\ e \mid \mathsf{unfold}_C\ e \\ \text{value} \quad v \quad &::= \quad \cdots \mid \mathsf{fold}_C\ v\end{aligned}$$

The typing rules for fold_C and unfold_C are derived from the operational interpretation of fold_C and unfold_C given above:

$$\frac{C = \mu\alpha.A \quad \Gamma \vdash e : [C/\alpha]A \quad \Gamma \vdash C \text{ type}}{\Gamma \vdash \mathsf{fold}_C\ e : C} \text{ Fold} \qquad \frac{C = \mu\alpha.A \quad \Gamma \vdash e : C}{\Gamma \vdash \mathsf{unfold}_C\ e : [C/\alpha]A} \text{ Unfold}$$

The following reduction rules are based on the eager reduction strategy:

$$\frac{e \mapsto e'}{\mathsf{fold}_C\ e \mapsto \mathsf{fold}_C\ e'} \text{ Fold}$$

$$\frac{e \mapsto e'}{\mathsf{unfold}_C\ e \mapsto \mathsf{unfold}_C\ e'} \text{ Unfold} \qquad \frac{}{\mathsf{unfold}_C\ \mathsf{fold}_C\ v \mapsto v} \text{ Unfold}^2$$

Exercise 12.1. Propose reduction rules for the lazy reduction strategy.

Exercise 12.2. Why do we need no reduction rule for $\mathsf{fold}_C\ \mathsf{unfold}_C\ v$?

12.2 Recursive data structures

This section presents a few examples of translating datatype declarations of SML to recursive types. The key idea is two-fold: (1) each datatype declaration in SML implicitly introduces a recursive type; (2) each data constructor belonging to datatype C implicitly uses fold_C and each pattern match for datatype C implicitly uses unfold_C.

Let us begin with a non-recursive datatype which does not have to be translated to a recursive type:

$$\texttt{datatype bool} \;=\; \texttt{True} \,|\, \texttt{False}$$

Since a value of type bool is either `True` *or* `False`, we translate bool to a sum type unit+unit so as to express the existence of two alternatives; the use of type unit indicates that data constructors `True` and `False` require no argument:

$$
\begin{aligned}
\texttt{bool} &\;=\; \text{unit}+\text{unit}\\
\texttt{True} &\;=\; \text{inl}_{\text{unit}}\,()\\
\texttt{False} &\;=\; \text{inr}_{\text{unit}}\,()\\
\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 &\;=\; \text{case } e \text{ of inl \_.} e_1 \mid \text{inr \_.} e_2
\end{aligned}
$$

Thus data constructors, which are separated by $|$ in a datatype declaration, become separated by $+$ when translated to a type in the simply typed λ-calculus.

Now consider a recursive datatype for natural numbers:

$$\texttt{datatype nat} \;=\; \texttt{Zero} \,|\, \texttt{Succ of nat}$$

A recursive type for nat is the solution to the equation $\texttt{nat} \cong \text{unit}+\texttt{nat}$ where the left unit corresponds to `Zero` and the right nat corresponds to `Succ`:

$$\texttt{nat} \;=\; \mu\alpha.\text{unit}+\alpha$$

Then both data constructors `Zero` and `Succ` first prepare a value of type unit+nat and then "fold" it to create a value of type nat:

$$
\begin{aligned}
\texttt{Zero} &\;=\; \text{fold}_{\text{nat}}\,\text{inl}_{\text{nat}}\,()\\
\texttt{Succ } e &\;=\; \text{fold}_{\text{nat}}\,\text{inr}_{\text{unit}}\,e
\end{aligned}
$$

A pattern match for datatype nat works in the opposite way: it first "unfolds" a value of type nat to obtain a value of type unit+nat which is then analyzed by a case expression:

$$\texttt{case } e \texttt{ of Zero} \Rightarrow e_1 \mid \texttt{Succ } x \Rightarrow e_2 \;=\; \text{case unfold}_{\text{nat}}\,e \text{ of inl \_.} e_1 \mid \text{inr } x.\, e_2$$

Similarly a recursive datatype for lists of natural numbers is translated as follows:

$$
\begin{aligned}
\texttt{datatype nlist} &\;=\; \texttt{Nil} \,|\, \texttt{Cons of nat} \times \texttt{nlist}\\
\texttt{nlist} &\;=\; \mu\alpha.\text{unit}+(\text{nat}\times\alpha)\\
\texttt{Nil} &\;=\; \text{fold}_{\text{nlist}}\,\text{inl}_{\text{nat}\times\text{nlist}}\,()\\
\texttt{Cons } e &\;=\; \text{fold}_{\text{nlist}}\,\text{inr}_{\text{unit}}\,e\\
\texttt{case } e \texttt{ of Nil} \Rightarrow e_1 \mid \texttt{Cons } x \Rightarrow e_2 &\;=\; \text{case unfold}_{\text{nlist}}\,e \text{ of inl \_.} e_1 \mid \text{inr } x.\, e_2
\end{aligned}
$$

As an example of a recursive type that does not use a sum type, let us consider a datatype for streams of natural numbers:

$$
\begin{aligned}
\texttt{datatype nstream} &\;=\; \texttt{Nstream of unit} \rightarrow \texttt{nat} \times \texttt{nstream}\\
\texttt{nstream} &\;=\; \mu\alpha.\text{unit} \rightarrow \text{nat} \times \alpha
\end{aligned}
$$

When "unfolded," a value of type nstream yields a function of type $\text{unit} \rightarrow \text{nat} \times \text{nstream}$ which returns a natural number and another stream. For example, the following λ-abstraction has type $\text{nstream} \rightarrow \text{nat} \times \text{nstream}$:

$$\lambda s\,{:}\,\texttt{nstream}.\,\text{unfold}_{\text{nstream}}\,s\,()$$

The following function, of type $\text{nat} \rightarrow \text{nstream}$, returns a stream of natural numbers beginning with its argument:

$$\lambda n\,{:}\,\texttt{nat}.\,(\text{fix } f\,{:}\,\texttt{nat} \rightarrow \texttt{nstream}.\,\lambda x\,{:}\,\texttt{nat}.\,\text{fold}_{\text{nstream}}\,\lambda y\,{:}\,\texttt{unit}.\,(x, f\,(\texttt{Succ } x)))\,n$$

12.3 Typing the untyped λ-calculus

A further application of recursive types is a translation of the untyped λ-calculus to the simply typed λ-calculus augmented with recursive types. Specifically we wish to translate the untyped λ-calculus to the simply typed λ-calculus with the following definition:

$$
\begin{array}{lll}
\text{type} & A & ::= \quad A \to A \mid \alpha \mid \mu\alpha.A \\
\text{expression} & e & ::= \quad x \mid \lambda x\!:\!A.\, e \mid e\, e \mid \mathsf{fold}_A\, e \mid \mathsf{unfold}_A\, e
\end{array}
$$

Note that unlike the pure simply typed λ-calculus, the definition of types does not include base types.

We translate an expression e in the untyped λ-calculus to an expression e° in the simply typed λ-calculus. We treat all expressions in the untyped λ-calculus alike by assigning a unique type Ω (*i.e.*, e° is to have type Ω). Then the key to the translation is to find such a unique type Ω.

It is not difficult to find such a type Ω when recursive types are available. If every expression is assigned type Ω, we may think that $\lambda x.\, e$ is assigned type $\Omega \to \Omega$ *as well as* type Ω. Or, in order for $e_1\, e_2$ to be assigned type Ω, e_1 must be assigned *not only* type Ω but also type $\Omega \to \Omega$ because e_2 is assigned type Ω. Thus Ω must be identified with $\Omega \to \Omega$ (*i.e.*, $\Omega \cong \Omega \to \Omega$) and is defined as follows:

$$
\Omega \quad = \quad \mu\alpha.\alpha \to \alpha
$$

Then expressions in the untyped λ-calculus are translated as follows:

$$
\begin{array}{lll}
x^\circ & = & x \\
(\lambda x.\, e)^\circ & = & \mathsf{fold}_\Omega\, \lambda x\!:\!\Omega.\, e^\circ \\
(e_1\, e_2)^\circ & = & (\mathsf{unfold}_\Omega\, e_1{}^\circ)\, e_2{}^\circ
\end{array}
$$

Proposition 12.3. $\cdot \vdash e^\circ : \Omega$ *holds for any expression e in the untyped λ-calculus.*

Proposition 12.4. *If $e \mapsto e'$, then $e^\circ \mapsto^* e'^\circ$.*

In Proposition 12.4, extra reduction steps in $e^\circ \mapsto^* e'^\circ$ are due to applications of the rule *Unfold²*.

An interesting consequence of the translation is that despite the absence of the fixed point construct, the reduction of an expression in the simply typed λ-calculus with recursive types may not terminate! For example, the reduction of $((\lambda x.\, x\, x)\, (\lambda x.\, x\, x))^\circ$ does not terminate because $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ reduces to itself. In fact, we can even write recursive functions — all we have to do is to translate the fixed point combinator fix (see Section 3.5)!

12.4 Exercises

Exercise 12.5. Consider the simply typed λ-calculus augmented with recursive types. We use a function type $A \to B$ for non-recursive functions from type A to type B. Now let us introduce another function type $A \Rightarrow B$ for *recursive* functions from type A to type B. Define $A \Rightarrow B$ in terms of ordinary function types and recursive types.

Chapter 13

Polymorphism

In programming language theory, *polymorphism* (where *poly* means "many" and *morph* "shape") refers to the mechanism by which the same piece of code can be reused for different types of objects. C++ templates are a good example of a language construct for polymorphism: the same C++ template can be instantiated to different classes which operate on different types of objects all in a uniform way. The recent version of Java (J2SE 5.0) also supports *generics* which provides polymorphism in a similar way to C++ templates.

There are two kinds of polymorphism: *parametric polymorphism* and *ad hoc polymorphism*. Parametric polymorphism enables us to write a piece of code that operates on *all* types of objects all in a *uniform* way. Such a piece of code provides a high degree of generality by accepting all types of objects, but cannot exploit specific properties of different types of objects.[1] *Ad hoc* polymorphism, in contrast, allows a piece of code to exhibit different behavior depending on the type of objects it operates on. The operator + of SML is an example of *ad hoc* polymorphism: both `int * int -> int` and `real * real -> real` are valid types for +, which manipulates integers and floating point numbers differently. In this chapter, we restrict ourselves to parametric polymorphism.

We begin with *System F*, an extension of the untyped λ-calculus with polymorphic types. Despite its syntactic simplicity and rich expressivity, System F is not a good framework for practical functional languages because the problem of assigning a polymorphic type (of System F) to an expression in the untyped λ-calculus is undecidable (*i.e.*, there is no algorithm for solving the problem for all input expressions). We will then take an excursion to the *predicate polymorphic λ-calculus*, another extension of the untyped λ-calculus with polymorphic types which is a sublanguage of System F and is thus less expressive than System F. (Interestingly it uses slightly more complex syntax.) Our study of polymorphism will culminate in the formulation of the polymorphic type system of SML, called *let-polymorphism*, which is a variant of the type system of the predicate polymorphic λ-calculus. Hence the study of System F is our first step toward the polymorphic type system of SML!

13.1   System F

Consider a λ-abstraction $\lambda x. x$ of the untyped λ-calculus. We wish to extend the definition of the untyped λ-calculus so that we can assign a type to $\lambda x. x$. Assigning a type to $\lambda x. x$ involves two tasks: binding variable x to a type and deciding the type of the resultant expression.

In the case of the simply typed λ-calculus, we have to choose a specific type for variable x, say, bool. Then the resultant λ-abstraction $\lambda x\!:\!\text{bool}. x$ has type bool$\rightarrow$bool. Ideally, however, we do not want to stipulate a specific type for x because $\lambda x. x$ is an identity function that works for any type. For example, $\lambda x. x$ is an identify function for an integer type int, but once the type of $\lambda x. x$ is fixed as bool$\rightarrow$bool, we cannot use it for integers. Hence a better answer would be to bind x to an "any type" α and assign type $\alpha\rightarrow\alpha$ to $\lambda x\!:\!\alpha. x$.

Now every variable in the untyped λ-calculus is assigned an "any type," and there arises a need to distinguish between different "any types." As an example, consider a λ-abstraction $\lambda x. \lambda y. (x, y)$ where

---

[1]Generics in the Java language does not fully support parametric polymorphism: it accepts only Java objects (of class `Object`), and does not accept primitive types such as `int`.

(x, y) denotes a pair of x and y. Since both x and y may assume an "any type," we could assign the same "any type" to x and y as follows:

$$\lambda x\!:\!\alpha.\,\lambda y\!:\!\alpha.\,(x, y) : \alpha \to \alpha \to \alpha \times \alpha$$

Although it is fine to assign the same "any type" to both x and y, it does not give the most general type for $\lambda x.\,\lambda y.\,(x, y)$ because x and y do not have to assume the same type in general. Instead we need to assign a different "any type" β to y so that x and y remain independent of each other:

$$\lambda x\!:\!\alpha.\,\lambda y\!:\!\beta.\,(x, y) : \alpha \to \beta \to \alpha \times \beta$$

Since each variable in a given expression may need a fresh "any type," we introduce a new construct $\Lambda\alpha.\,e$, called a *type abstraction*, for declaring α as a fresh "any type," or a fresh *type variable*. That is, a type abstraction $\Lambda\alpha.\,e$ declares a type variable α for use in expression e; we may rename α in a way analogous to α-conversions on λ-abstractions. If e has type A, then $\Lambda\alpha.\,e$ is assigned a polymorphic type $\forall\alpha.A$ which reads *"for all α, A."* Note that A in $\forall\alpha.A$ may use α (*e.g.*, $A = \alpha \to \alpha$). Then $\lambda x.\,\lambda y.\,(x, y)$ is converted to the following expression:

$$\Lambda\alpha.\,\Lambda\beta.\,\lambda x\!:\!\alpha.\,\lambda y\!:\!\beta.\,(x, y) : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \times \beta$$

$\to$ has a higher operator precedence than $\forall$, so $\forall\alpha.A \to B$ is equal to $\forall\alpha.(A \to B)$, not $(\forall\alpha.A) \to B$. Hence $\forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \times \beta$ is equal to $\forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha \times \beta)$.

Back to the example of the identity function which is now written as $\Lambda\alpha.\,\lambda x\!:\!\alpha.\,x$ of type $\forall\alpha.\alpha \to \alpha$, let us apply it to a boolean truth true of type bool. First we need to convert $\Lambda\alpha.\,\lambda x\!:\!\alpha.\,x$ to an identity function $\lambda x\!:\!\text{bool}.\,x$ by instantiating α to a specific type bool. To this end, we introduce a new construct $e\,[\![A]\!]$, called a *type application*, such that $(\Lambda\alpha.\,e)\,[\![A]\!]$ reduces to $[A/\alpha]e$ which substitutes A for α in e:

$$(\Lambda\alpha.\,e)\,[\![A]\!] \mapsto [A/\alpha]e$$

(Thus the only difference of a type application $e\,[\![A]\!]$ from an ordinary application $e_1\,e_2$ is that a type application substitutes a type for a type variable instead of an expression for an ordinary variable.) Then $(\Lambda\alpha.\,\lambda x\!:\!\alpha.\,x)\,[\![\text{bool}]\!]$ reduces to an identity function specialized for type bool, and an ordinary application $(\Lambda\alpha.\,\lambda x\!:\!\alpha.\,x)\,[\![\text{bool}]\!]$ true finishes the job.

System F is essentially an extension of the untyped λ-calculus with type abstractions and type applications. Although variables in λ-abstractions are always annotated with their types, we do not consider System F as an extension of the simply typed λ-calculus because System F does not have to assume base types. The abstract syntax for System F is as follows:

type	A	$::=$	$A \to A \mid \alpha \mid \forall\alpha.A$
expression	e	$::=$	$x \mid \lambda x\!:\!A.\,e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,[\![A]\!]$
value	v	$::=$	$\lambda x\!:\!A.\,e \mid \Lambda\alpha.\,e$

The reduction rules for type applications are analogous to those for ordinary applications except that there is no reduction rule for types:

$$\frac{e \mapsto e'}{e\,[\![A]\!] \mapsto e'\,[\![A]\!]}\ \textit{Tlam} \qquad \frac{}{(\Lambda\alpha.\,e)\,[\![A]\!] \mapsto [A/\alpha]e}\ \textit{Tapp}$$

$[A/\alpha]e$ substitutes A for α in e; we omit its pedantic definition here. For ordinary applications, we reuse the reductions rules for the simply typed λ-calculus.

There are two important observations to make about the abstract syntax. First the syntax for type applications implies that type variables may be instantiated to all kinds of types, including even polymorphic types. For example, the identity function $\Lambda\alpha.\,\lambda x\!:\!\alpha.\,x$ may be applied to its own type $\forall\alpha.\alpha \to \alpha$! Such flexibility in type applications is the source of rich expressivity of System F, but on the other hand, it also makes System F a poor choice as a framework for practical functional languages. Second we define a type abstraction $\Lambda\alpha.\,e$ as a value, even though it appears to be computationally equivalent to e.

As an example, let us write a function compose for composing two functions. One approach is to require that all type variables in the type of compose be instantiated before producing a λ-abstraction:

$$\begin{aligned}
\text{compose} \quad &: \quad \forall\alpha.\forall\beta.\forall\gamma.(\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma) \\
\text{compose} \quad &= \quad \Lambda\alpha.\,\Lambda\beta.\,\Lambda\gamma.\,\lambda f\!:\!\alpha\to\beta.\,\lambda g\!:\!\beta\to\gamma.\,\lambda x\!:\!\alpha.\,g\,(f\,x)
\end{aligned}$$

Alternatively we may require that only the first two type variables α and β be instantiated before producing a λ-abstraction which returns a type abstraction expecting the third type variable γ:

$$\begin{aligned}
\text{compose} \quad &: \quad \forall\alpha.\forall\beta.(\alpha\to\beta)\to\forall\gamma.(\beta\to\gamma)\to(\alpha\to\gamma) \\
\text{compose} \quad &= \quad \Lambda\alpha.\,\Lambda\beta.\,\lambda f\!:\!\alpha\to\beta.\,\Lambda\gamma.\,\lambda g\!:\!\beta\to\gamma.\,\lambda x\!:\!\alpha.\,g\,(f\,x)
\end{aligned}$$

As for the type system, System F is *not* a straightforward extension of the simply typed λ-calculus because of the inclusion of type variables. In the simply typed λ-calculus, $x : A$ qualifies as a valid type binding regardless of type A and the order of type bindings in a typing context does not matter by Proposition 4.1. In System F, $x : A$ may not qualify as a valid type binding if type A contains type variables. For example, without type abstractions declaring type variables α and β, we may not use $\alpha\to\beta$ as a type and hence $x : \alpha\to\beta$ is not a valid type binding. This observation leads to the conclusion that in System F, a typing context consists not only of type bindings but also of a new form of declarations for indicating which type variables are valid and which are not; moreover the order of elements in a typing context now *does* matter because of type variables.

We define a typing context as an ordered set of type bindings and *type declarations*; a type declaration α type declares α as a type, or equivalently, α as a valid type variable:

$$\text{typing context} \qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A \mid \Gamma, \alpha \text{ type}$$

We simplify the presentation by assuming that variables and type variables in a typing context are all distinct. We consider a type variable α as valid only if its type declaration appears to its left. For example, $\Gamma_1, \alpha \text{ type}, x : \alpha\to\alpha$ is a valid typing context because α in $x : \alpha\to\alpha$ has been declared as a type variable in α type (provided that Γ_1 is also a valid typing context). $\Gamma_1, x : \alpha\to\alpha, \alpha \text{ type}$ is, however, not a valid typing context because α is used in $x : \alpha\to\alpha$ before it is declared as a type variable in α type.

The type system of System F uses two forms of judgments: a typing judgment $\Gamma \vdash e : A$ whose meaning is the same as in the simply typed λ-calculus, and a *type judgment* $\Gamma \vdash A$ type which means that A is a valid type with respect to typing context Γ.[2] We need type judgments because the definition of syntactic category type in the abstract syntax is incapable of differentiating valid type variables from invalid ones. We refer to an inference rule deducing a type judgment as a *type rule*.

The type system of System F uses the following rules:

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma \vdash B \text{ type}}{\Gamma \vdash A\to B \text{ type}}\ \text{Ty}\!\to \qquad \frac{\alpha \text{ type} \in \Gamma}{\Gamma \vdash \alpha \text{ type}}\ \text{TyVar} \qquad \frac{\Gamma, \alpha \text{ type} \vdash A \text{ type}}{\Gamma \vdash \forall\alpha.A \text{ type}}\ \text{Ty}\forall$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{Var} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x\!:\!A.\,e : A\to B}\ \to\!\text{I} \qquad \frac{\Gamma \vdash e : A\to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\,e' : B}\ \to\!\text{E}$$

$$\frac{\Gamma, \alpha \text{ type} \vdash e : A}{\Gamma \vdash \Lambda\alpha.\,e : \forall\alpha.A}\ \forall\text{I} \qquad \frac{\Gamma \vdash e : \forall\alpha.B \quad \Gamma \vdash A \text{ type}}{\Gamma \vdash e\,[\![A]\!] : [A/\alpha]B}\ \forall\text{E}$$

A proof of a type judgment $\Gamma \vdash A$ type does not use type bindings in Γ. In the rule $\to$I, the typing context $\Gamma, x : A$ assumes that A is a valid type with respect to Γ. Hence the rule $\to$I does not need a separate premise $\Gamma \vdash A$ type. The rule $\forall$I, called the $\forall$ Introduction rule, introduces a polymorphic type $\forall\alpha.A$ from the judgment in the premise. The rule $\forall$E, called the $\forall$ Elimination rule, eliminates a polymorphic type $\forall\alpha.B$ by substituting a valid type A for type variable α. Note that the typing rule $\forall$E uses a substitution of a type into another *type* whereas the reduction rule *Tapp* uses a substitution of a type into an *expression*.

---

[2] A type judgment $\Gamma \vdash A$ type is also an example of a hypothetical judgment which deduces a "judgment" A type using each "judgment" A_i type in Γ as a hypothesis.

As an example of a typing derivation, let us find the type of an identity function specialized for type bool; we assume that $\Gamma \vdash$ bool type holds for any typing context Γ (see the type rule TyBool below):

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\alpha \text{ type}, x:\alpha \vdash x:\alpha}\text{ Var}}{\alpha \text{ type} \vdash \lambda x{:}\alpha.\, x:\alpha \to \alpha}\to\!\mathsf{I}}{\cdot \vdash \Lambda\alpha.\, \lambda x{:}\alpha.\, x:\forall\alpha.\alpha\to\alpha}\forall\mathsf{I} \quad \cfrac{}{\cdot \vdash \text{ bool type}}\text{ TyBool}}{\cdot \vdash (\Lambda\alpha.\, \lambda x{:}\alpha.\, x)\,[\![\text{bool}]\!] : [\text{bool}/\alpha](\alpha \to \alpha)}\forall\mathsf{E}$$

Since $[\text{bool}/\alpha](\alpha \to \alpha)$ is equal to bool $\to$ bool, the type application has type bool $\to$ bool.

The proof of type safety of System F needs three substitution lemmas as there are three kinds of substitutions: $[A/\alpha]B$ for the rule $\forall\mathsf{E}$, $[A/\alpha]e$ for the rule *Tapp*, and $[e'/x]e$ for the rule *App*. We write $[A/\alpha]\Gamma$ for substituting A for α in all type bindings in Γ.

Lemma 13.1 (Type substitution into types).
If $\Gamma \vdash A$ type and Γ, α type, $\Gamma' \vdash B$ type, then $\Gamma, [A/\alpha]\Gamma' \vdash [A/\alpha]B$ type.

Lemma 13.2 (Type substitution into expressions).
If $\Gamma \vdash A$ type and Γ, α type, $\Gamma' \vdash e : B$, then $\Gamma, [A/\alpha]\Gamma' \vdash [A/\alpha]e : [A/\alpha]B$.

Lemma 13.3 (Expression substitution).
If $\Gamma \vdash e : A$ and $\Gamma, x : A, \Gamma' \vdash e' : C$, then $\Gamma, \Gamma' \vdash [e/x]e' : C$.

In Lemmas 13.1 and 13.2, we have to substitute A into Γ', which may contain types involving α. In Lemma 13.2, we have to substitute A into e and B, both of which may contain types involving α. Lemma 13.3 reflects the fact that typing contexts are ordered sets.

The proof of type safety of System F is similar to the proof for the simply typed λ-calculus. We need to extend the canonical forms lemma (Lemma 4.5) and the inversion lemma (Lemma 4.8):

Lemma 13.4 (Canonical forms).
If v is a value of type $\forall\alpha.A$, then v is a type abstraction $\Lambda\alpha.\, e$.

Lemma 13.5 (Inversion). *Suppose $\Gamma \vdash e : C$.*
If $e = \Lambda\alpha.\, e'$, then $C = \forall\alpha.A$ and Γ, α type $\vdash e' : A$.

Theorem 13.6 (Progress). *If $\cdot \vdash e : A$ for some type A, then either e is a value or there exists e' such that $e \mapsto e'$.*

Theorem 13.7 (Type preservation). *If $\Gamma \vdash e : A$ and $e \mapsto e'$, then $\Gamma \vdash e' : A$.*

13.2   Type reconstruction

The type systems of the simply typed λ-calculus and System F require that all variables in λ-abstractions be annotated with their types. While it certainly simplifies the proof of type safety (and the study of type-theoretic properties in general), such a requirement on variables is not a good idea when it comes to designing practical functional languages. One reason is that annotating all variables with their types does not always improve code readability. On the contrary, excessive type annotations often reduces code readability! For example, one would write an SML function adding two integers as `fn x => fn y => x + y`, which is no less readable than a fully type-annotated function `fn x : int => fn y : int => x + y`. A more important reason is that in many cases, types of variables can be inferred, or *reconstructed*, from the context. For example, the presence of `+` in `fn x => fn y => x + y` gives enough information to decide a unique type `int` for both `x` and `y`. Thus we wish to eliminate such a requirement on variables, so as to provide programmers with more flexibility in type annotations, by developing a *type reconstruction* algorithm which automatically infers types for variables.

In the case of System F, the goal of type reconstruction is to convert an expression e in the untyped λ-calculus to a well-typed expression e' in System F such that erasing type annotations (including type abstractions and type applications) in e' yields the original expression e. That is, by reconstructing

types for all variables in e, we obtain a new well-typed expression e' in System F. Formally we define an *erasure* function $erase(\cdot)$ which takes an expression in System F and erases all type annotations in it:

$$
\begin{aligned}
erase(x) &= x \\
erase(\lambda x\!:\!A.\,e) &= \lambda x.\,erase(e) \\
erase(e_1\ e_2) &= erase(e_1)\ erase(e_2) \\
erase(\Lambda\alpha.\,e) &= erase(e) \\
erase(e\,[\![A]\!]) &= erase(e)
\end{aligned}
$$

The erasure function respects the reduction rules for System F in the following sense:

Proposition 13.8. *If $e \mapsto e'$ holds in System F, then $erase(e) \mapsto^* erase(e')$ holds in the untyped λ-calculus.*

The problem of type reconstruction is then to convert an expression e in the untyped λ-calculus to a well-typed expression e' in System F such that $erase(e') = e$. We say that an expression e in the untyped λ-calculus is *typable* in System F if there exists such a well-typed expression e'.

As an example, let us consider an untyped λ-abstraction $\lambda x.\,x\ x$. It is not typable in the simply typed λ-calculus because the first x in $x\ x$ must have a type strictly larger than the second x, which is impossible. It is, however, typable in System F because we can replace the first x in $x\ x$ by a type application. Specifically $\lambda x\!:\!\forall\alpha.\alpha\!\to\!\alpha.\,x\,[\![\forall\alpha.\alpha\!\to\!\alpha]\!]\ x$ is a well-typed expression in System F which erases to $\lambda x.\,x\ x$:

$$
\cfrac{
\cfrac{
\cfrac{x:\forall\alpha.\alpha\!\to\!\alpha \vdash x:\forall\alpha.\alpha\!\to\!\alpha \quad \text{Var} \qquad x:\forall\alpha.\alpha\!\to\!\alpha \vdash \forall\alpha.\alpha\!\to\!\alpha\ \text{type}}{x:\forall\alpha.\alpha\!\to\!\alpha \vdash x\,[\![\forall\alpha.\alpha\!\to\!\alpha]\!] : (\forall\alpha.\alpha\!\to\!\alpha)\!\to\!(\forall\alpha.\alpha\!\to\!\alpha)}\ \forall\text{E} \qquad \cfrac{}{x:\forall\alpha.\alpha\!\to\!\alpha \vdash x:\forall\alpha.\alpha\!\to\!\alpha}\ \text{Var}
}{x:\forall\alpha.\alpha\!\to\!\alpha \vdash x\,[\![\forall\alpha.\alpha\!\to\!\alpha]\!]\ x:\forall\alpha.\alpha\!\to\!\alpha}\ \to\text{E}
}{\cdot \vdash \lambda x\!:\!\forall\alpha.\alpha\!\to\!\alpha.\,x\,[\![\forall\alpha.\alpha\!\to\!\alpha]\!]\ x:(\forall\alpha.\alpha\!\to\!\alpha)\!\to\!(\forall\alpha.\alpha\!\to\!\alpha)}\ \to\text{I}
$$

The proof of $x:\forall\alpha.\alpha\!\to\!\alpha \vdash \forall\alpha.\alpha\!\to\!\alpha\ \text{type}$ is shown below:

$$
\cfrac{
\cfrac{
\cfrac{\alpha\ \text{type} \in x:\forall\alpha.\alpha\!\to\!\alpha, \alpha\ \text{type}}{x:\forall\alpha.\alpha\!\to\!\alpha, \alpha\ \text{type} \vdash \alpha\ \text{type}}\ \text{TyVar} \quad \cfrac{\alpha\ \text{type} \in x:\forall\alpha.\alpha\!\to\!\alpha, \alpha\ \text{type}}{x:\forall\alpha.\alpha\!\to\!\alpha, \alpha\ \text{type} \vdash \alpha\ \text{type}}\ \text{TyVar}
}{x:\forall\alpha.\alpha\!\to\!\alpha, \alpha\ \text{type} \vdash \alpha\!\to\!\alpha\ \text{type}}\ \text{Ty}\!\to
}{x:\forall\alpha.\alpha\!\to\!\alpha \vdash \forall\alpha.\alpha\!\to\!\alpha\ \text{type}}\ \text{Ty}\forall
$$

(The proof does not use the type binding $x:\forall\alpha.\alpha\!\to\!\alpha$.) Hence a type reconstruction algorithm for System F, *if any*, would convert $\lambda x.\,x\ x$ to $\lambda x\!:\!\forall\alpha.\alpha\!\to\!\alpha.\,x\,[\![\forall\alpha.\alpha\!\to\!\alpha]\!]\ x$.

It turns out that not every expression in the untyped λ-calculus is typable in System F. For example, omega $= (\lambda x.\,x\ x)\ (\lambda x.\,x\ x)$ is not typable: there is no well-typed expression in System F that erases to omega. The proof exploits the *normalization* property of System F which states that the reduction of a well-typed expression in System F always terminates. Thus a type reconstruction algorithm for System F first decides if a given expression e is typable or not in System F; if e is typable, the algorithm yields a corresponding expression in System F.

Unfortunately the problem of type reconstruction in System F is undecidable: there is no algorithm for deciding whether a given expression in the untyped λ-calculus is typable or not in System F. Our plan now is to find a compromise between rich expressivity and decidability of type reconstruction — we wish to identify a sublanguage of System F that supports polymorphic types and also has a decidable type construction algorithm. Section 13.4 presents such a sublanguage, called the *predicative polymorphic λ-calculus*, which is extended to the polymorphic type system of SML in Section 13.5.

13.3 Programming in System F

We have seen in Section 3.4 how to encode common datatypes in the untyped λ-calculus. While these expressions correctly encode their respective datatypes, unavailability of a type system makes it difficult to express the intuition behind the encoding of each datatype. Besides it is often tedious and even unreliable to check the correctness of an encoding without recourse to a type system.

In this section, we rewrite these untyped expressions into well-typed expressions in System F. A direct definition of a datatype in terms of types in System F provides the intuition behind its encoding,

and availability of type annotations within expressions makes it easy to check the correctness of the encoding.

Let us begin with base types bool and nat for Church booleans and numerals, respectively. The intuition behind Church booleans is that a boolean value chooses one of two different options. The following definition of the base type bool is based on the decision to assign the same type α to both options:

$$\text{bool} \quad = \quad \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$$

Then boolean values true and false, both of type bool, are encoded as follows:

$$\begin{aligned} \text{true} &= \Lambda\alpha.\,\lambda t{:}\alpha.\,\lambda f{:}\alpha.\,t \\ \text{false} &= \Lambda\alpha.\,\lambda t{:}\alpha.\,\lambda f{:}\alpha.\,f \end{aligned}$$

The intuition behind Church numerals is that a Church numeral $\widehat{n}$ takes a function f and returns another function f^n which applies f exactly n times. In order for f^n to be well-typed, its argument type and return type must be identical. Hence we define the base type nat in System F as follows:[3]

$$\text{nat} \quad = \quad \forall \alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$$

Then a zero zero of type nat and a successor function succ of type nat $\rightarrow$ nat are encoded as follows:

$$\begin{aligned} \text{zero} &= \Lambda\alpha.\,\lambda f{:}\alpha{\rightarrow}\alpha.\,\lambda x{:}\alpha.\,x \\ \text{succ} &= \lambda n{:}\text{nat}.\,\Lambda\alpha.\,\lambda f{:}\alpha{\rightarrow}\alpha.\,\lambda x{:}\alpha.\,(n\,[\![\alpha]\!]\,f)\,(f\,x) \end{aligned}$$

The definition of a product type $A \times B$ in System F exploits the fact that in essence, a value of type $A \times B$ contains a value of type A and another value of type B. If we think of $A \rightarrow B \rightarrow \alpha$ as a type for a function taking two arguments of types A and B and returning a value of type α, a value of type $A \times B$ contains everything necessary for applying such a function, which is expressed in the following definition of $A \times B$:

$$A \times B = \forall \alpha.(A \rightarrow B \rightarrow \alpha) \rightarrow \alpha$$

Pairs and projections are encoded as follows; note that without type annotations, these expressions degenerate to pairs and projections for the untyped λ-calculus given in Section 3.4:

$$\begin{aligned} \text{pair} &: \forall\alpha.\forall\beta.\alpha\rightarrow\beta\rightarrow\alpha\times\beta &=& \quad \Lambda\alpha.\,\Lambda\beta.\,\lambda x{:}\alpha.\,\lambda y{:}\beta.\,\Lambda\gamma.\,\lambda f{:}\alpha{\rightarrow}\beta{\rightarrow}\gamma.\,f\,x\,y \\ \text{fst} &: \forall\alpha.\forall\beta.\alpha\times\beta\rightarrow\alpha &=& \quad \Lambda\alpha.\,\Lambda\beta.\,\lambda p{:}\alpha\times\beta.\,p\,[\![\alpha]\!]\,(\lambda x{:}\alpha.\,\lambda y{:}\beta.\,x) \\ \text{snd} &: \forall\alpha.\forall\beta.\alpha\times\beta\rightarrow\beta &=& \quad \Lambda\alpha.\,\Lambda\beta.\,\lambda p{:}\alpha\times\beta.\,p\,[\![\beta]\!]\,(\lambda x{:}\alpha.\,\lambda y{:}\beta.\,y) \end{aligned}$$

The type unit is a general product type with no element and is thus defined as $\forall\alpha.\alpha\rightarrow\alpha$ which is obtained by removing A and B from the definition of $A \times B$. The encoding of a unit () is obtained by removing x and y from the encoding of pair:

$$() \quad : \quad \text{unit} \quad = \quad \Lambda\alpha.\,\lambda x{:}\alpha.\,x$$

The definition of a sum type $A+B$ in System F reminds us of the typing rule $+$E for sum types: given a function f of type $A \rightarrow \alpha$ and another function g of type $B \rightarrow \alpha$, a value v of type $A+B$ applies the right function (either f or g) to the value contained in v:

$$A+B = \forall\alpha.(A \rightarrow \alpha) \rightarrow (B \rightarrow \alpha) \rightarrow \alpha$$

Injections and case expressions are translations of the typing rules $+_\text{L}$, $+_\text{R}$, and $+$E:

$$\begin{aligned} \text{inl} &: \forall\alpha.\forall\beta.\alpha\rightarrow\alpha{+}\beta \\ \text{inr} &: \forall\alpha.\forall\beta.\beta\rightarrow\alpha{+}\beta \\ \text{case} &: \forall\alpha.\forall\beta.\forall\gamma.\alpha{+}\beta\rightarrow(\alpha\rightarrow\gamma)\rightarrow(\beta\rightarrow\gamma)\rightarrow\gamma \end{aligned}$$

Exercise 13.9. Encode inl, inr, and case in System F.

The type void is a general sum type with no element and is thus defined as $\forall\alpha.\alpha$ which is obtained by removing A and B from the definition of $A+B$. Needless to say, there is no expression of type void in System F. (Why?)

---

[3] We may also interpret nat as nat $= \forall\alpha.(\alpha\rightarrow\alpha)\rightarrow\alpha\rightarrow\alpha$ such that a Church numeral $\widehat{n}$ takes a successor function succ of type $\alpha\rightarrow\alpha$ and a zero zero of type α to return succ^n 0 of type α.

13.4 Predicative polymorphic λ-calculus

This section presents the predicative polymorphic λ-calculus which is a sublanguage of System F with a decidable type construction algorithm. It is still not a good framework for practical functional languages because polymorphic types are virtually useless! Nevertheless it helps us a lot to motivate the development of let-polymorphism, the most popular polymorphic type system found in modern functional languages.

The key observation is that undecidability of type reconstruction in System F is traced back to the self-referential nature of polymorphic types: we augment the set of types with new elements called type variables and polymorphic types, but the syntax for type applications allows type variables to range over not only existing types (such as function types) but also these new elements which include polymorphic types themselves. That is, there is no restriction on type A in a type application $e\,[\![A]\!]$ where type A, which is to be substituted for a type variable, can be not only a function type but also another polymorphic type.

The predicative polymorphic λ-calculus recovers decidability of type reconstruction by prohibiting type variables from ranging over polymorphic types. We stratify types into two kinds: *monotypes* which exclude polymorphic types and *polytypes* which include all kinds of types:

$$
\begin{array}{rlll}
\text{monotype} & A & ::= & A \to A \mid \alpha \\
\text{polytype} & U & ::= & A \mid \forall \alpha.U \\
\text{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,[\![A]\!] \\
\text{value} & v & ::= & \lambda x\!:\!A.\,e \mid \Lambda\alpha.\,e \\
\text{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \text{type}
\end{array}
$$

A polytype is always written as $\forall \alpha.\forall \beta.\cdots\cdot\forall \gamma.A \to A'$ where $A \to A'$ cannot contain polymorphic types. For example, $(\forall \alpha.\alpha \to \alpha) \to (\forall \alpha.\beta \to \beta)$ is not a polytype whereas $\forall \alpha.\forall \beta.(\alpha \to \alpha) \to (\beta \to \beta)$ is. We say that a polytype is written in *prenex form* because a *type quantifier* $\forall \alpha$ may appear only as part of its prefix.

The main difference of the predicative polymorphic λ-calculus from System F is that a type application $e\,[\![A]\!]$ now accepts only a monotype A. (In System F, there is no distinction between monotypes and polytypes, and a type application can accept polymorphic types.) A type application $e\,[\![A]\!]$ itself, however, has a polytype if e has a polytype $\forall \alpha.U$ where U is another polytype (see the typing rule $\forall E$ below).

As in System F, the type system of the predicative polymorphic λ-calculus uses two forms of judgments: a typing judgment $\Gamma \vdash e : U$ and a type judgment $\Gamma \vdash A$ type. The difference is that $\Gamma \vdash A$ type now checks if a given type is a valid *monotype*. That is, we do not use a type judgment $\Gamma \vdash U$ type (which is actually unnecessary because every polytype is written in prenex form anyway). Thus the system system uses the following rules; note that the rule $\text{Ty}\forall$ from System F is gone:

$$
\frac{\Gamma \vdash A\ \text{type} \quad \Gamma \vdash B\ \text{type}}{\Gamma \vdash A \to B\ \text{type}}\ \text{Ty}\!\to \qquad \frac{\alpha\ \text{type} \in \Gamma}{\Gamma \vdash \alpha\ \text{type}}\ \text{TyVar}
$$

$$
\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{Var} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x\!:\!A.\,e : A \to B}\ \to\!\text{I} \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\,e' : B}\ \to\!\text{E}
$$

$$
\frac{\Gamma, \alpha\ \text{type} \vdash e : U}{\Gamma \vdash \Lambda\alpha.\,e : \forall \alpha.U}\ \forall\text{I} \qquad \frac{\Gamma \vdash e : \forall \alpha.U \quad \Gamma \vdash A\ \text{type}}{\Gamma \vdash e\,[\![A]\!] : [A/\alpha]U}\ \forall\text{E}
$$

Unfortunately the use of a monotype A in a λ-abstraction $\lambda x\!:\!A.\,e$ defeats the purpose of introducing polymorphic types into the type system: even though we can now write an expression of a polytype U, we can never instantiate type variables in U more than once! Suppose, for example, that we wish to apply a polymorphic identity function $\text{id} = \Lambda\alpha.\,\lambda x\!:\!\alpha.\,x$ to two different types, say, bool and int. In the untyped λ-calculus, we would bind a variable f to an identity function and then apply f twice:

$$(\lambda f.\,\text{pair}\ (f\ \text{true})\ (f\ 0))\ (\lambda x.\,x)$$

In the predicative polymorphic λ-calculus, it is impossible to reuse id more than once in this way, since f must be given a monotype while id has a polytype:

$$(\lambda f \colon \forall \alpha.\alpha \to \alpha.\, (f\, [\![\text{bool}]\!]\, \text{true},\, f\, [\![\text{int}]\!]\, 0))\, \text{id} \qquad \text{(ill-typed)}$$

(Here we use pairs for product types.) If we apply id to a monotype bool (or int), $f\, 0$ (or f true) in the body fails to typecheck:

$$(\lambda f \colon \text{bool} \to \text{bool}.\, (f\, \text{true},\, f\, 0))\, (\text{id}\, [\![\text{bool}]\!]) \qquad \text{(ill-typed)}$$

Thus the only interesting way to use a polymorphic function (of a polytype) is to use it "monomorphically" by converting it to a function of a certain monotype!

Let-polymorphism extends the predicative polymorphic λ-calculus with a new construct that enables us to use a polymorphic expression *polymorphically* in the sense that type variables in it can be instantiated more than once. The new construct preserves decidability of type reconstruction, so let-polymorphism is a good compromise between expressivity and decidability of type reconstruction.

13.5 Let-polymorphism

Let-polymorphism extends the predicative polymorphic λ-calculus with a new construct, called a let-*binding*, for declaring variables of polytypes. A let-binding let $x \colon U = e$ in e' binds x to a polymorphic expression e of type U and allows multiple occurrences of x in e'. With a let-binding, we can apply a polymorphic identity function to two different (mono)types bool and int as follows:

$$\text{let } f \colon \forall \alpha.\alpha \to \alpha = \Lambda \alpha.\, \lambda x \colon \alpha.\, x \text{ in } (f\, [\![\text{bool}]\!]\, \text{true},\, f\, [\![\text{int}]\!]\, 0))$$

Since variables can now assume polytypes, we use type bindings of the form $x : U$ instead of $x : A$. We require that let-bindings themselves be of monotypes:

$$
\begin{array}{rcl}
\text{expression} & e & ::= \quad \cdots \mid \text{let } x \colon U = e \text{ in } e \\
\text{typing context} & \Gamma & ::= \quad \cdot \mid \Gamma, x : U \mid \Gamma, \alpha \text{ type}
\end{array}
$$

$$\frac{x : U \in \Gamma}{\Gamma \vdash x : U}\ \text{Var} \qquad \frac{\Gamma \vdash e : U \quad \Gamma, x : U \vdash e' : A}{\Gamma \vdash \text{let } x \colon U = e \text{ in } e' : A}\ \text{Let}$$

The reduction of a let-binding let $x \colon U = e$ in e' proceeds by substituting e for x in e':

$$\text{let } x \colon U = e \text{ in } e' \mapsto [e/x]e'$$

Depending on the reduction strategy, we may choose to fully evaluate e before performing the substitution.

Although let $x \colon U = e$ in e' reduces to the same expression that an application $(\lambda x \colon A.\, e')\, e$ reduces to, it is *not* syntactic sugar for $(\lambda x \colon A.\, e')\, e$: when e has a polytype U, let $x \colon U = e$ in e' may typecheck by the rule Let, but in general, $(\lambda x \colon A.\, e')\, e$ does not typecheck because monotype A does not match the type of e. Therefore, in order to use a polymorphic expression polymorphically, *we must bind it to a variable using a* let-*binding instead of a* λ-*abstraction.*

Then why do we not just allow a λ-abstraction $\lambda x \colon U.\, e$ binding x to a polytype (which would degenerate let $x \colon U = e$ in e' into syntactic sugar)? The reason is that with an additional assumption that e may have a polytype (*e.g.*, $\lambda x \colon U.\, x$), such a λ-abstraction collapses the distinction between monotypes and polytypes. That is, polytypes constitute types of System F:

$$
\begin{array}{rclcrcl}
\text{monotype} & A & ::= & U \to U \mid \alpha & & & \\
\text{polytype} & U & ::= & A \mid \forall \alpha.U & \Longleftrightarrow & \text{type} \quad U & ::= \quad U \to U \mid \alpha \mid \forall \alpha.U
\end{array}
$$

We may construe a let-binding as a restricted use of a λ-abstraction $\lambda x \colon U.\, e$ (binding x to a polytype) such that it never stands alone as a first-class object and must be applied to a polymorphic expression immediately. At the cost of flexibility in applying such λ-abstractions, let-polymorphism retains decidability of type reconstruction without destroying the distinction between monotypes and polytypes and also without sacrificing too much expressivity. After all, we can still enjoy both polymorphism and decidability of type reconstruction, which is the reason why let-polymorphism is so popular among mainstream functional languages.

September 30, 2011

13.6 Implicit polymorphism

The polymorphic type systems considered so far are all "explicit" in that polymorphic types are introduced explicitly by type abstractions and that type variables are instantiated explicitly by type applications. An explicit polymorphic type system has the property that every well-typed polymorphic expression has a unique polymorphic type.

The type system of SML uses a different approach to polymorphism: it makes no use of type abstractions and type applications, but allows an expression to have multiple types by requiring no type annotations in λ-abstractions. That is, polymorphic types arise "implicitly" from lack of type annotations in λ-abstractions.

As an example, consider an identity function $\lambda x.\, x$. It can be assigned such types as $\mathsf{bool}\to\mathsf{bool}$, $\mathsf{int}\to\mathsf{int}$, $(\mathsf{int}\to\mathsf{int})\to(\mathsf{int}\to\mathsf{int})$, and so on. These types are all distinct, but are subsumed by the same polytype $\forall\alpha.\alpha\to\alpha$ in the sense that they are results of instantiating α in $\forall\alpha.\alpha\to\alpha$. We refer to $\forall\alpha.\alpha\to\alpha$ as the *principal type* of $\lambda x.\, x$, which may be thought of as the most general type for $\lambda x.\, x$, as opposed to specific types such as $\mathsf{bool}\to\mathsf{bool}$ and $\mathsf{int}\to\mathsf{int}$. The type reconstruction algorithm of SML infers a unique principal type for every well-typed expression. Below we discuss the type system of SML and defer details of the type reconstruction algorithm to Section 13.8.

In essence, the type system of SML uses let-polymorphism without type annotations (in λ-abstractions and let-bindings), type abstractions, and type applications:

$$
\begin{array}{llll}
\text{monotype} & A & ::= & A\to A \mid \alpha \\
\text{polytype} & U & ::= & A \mid \forall\alpha.U \\
\text{expression} & e & ::= & x \mid \lambda x.\, e \mid e\, e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
\text{value} & v & ::= & \lambda x.\, e \\
\text{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : U \mid \Gamma, \alpha\ \mathsf{type}
\end{array}
$$

We use a new typing judgment $\Gamma \triangleright e : U$ to express that untyped expression e is typable with a polytype U. The intuition (which will be made clear in Theorem 13.11) is that if $\Gamma \triangleright e : U$ holds where e is *untyped*, there exists a *typed* expression e' such that $\Gamma \vdash e' : U$ and e' erases to e by the following erasure function:

$$
\begin{array}{lll}
erase(x) & = & x \\
erase(\lambda x\!:\!A.\, e) & = & \lambda x.\, erase(e) \\
erase(e_1\, e_2) & = & erase(e_1)\, erase(e_2) \\
erase(\Lambda\alpha.\, e) & = & erase(e) \\
erase(e\,[\![A]\!]) & = & erase(e) \\
erase(\mathsf{let}\ x\!:\!U = e\ \mathsf{in}\ e') & = & \mathsf{let}\ x = erase(e)\ \mathsf{in}\ erase(e')
\end{array}
$$

That is, if $\Gamma \triangleright e : U$ holds, e has its counterpart in let-polymorphism in Section 13.5. The rules for the typing judgment $\Gamma \triangleright e : U$ are given as follows:

$$
\frac{x : U \in \Gamma}{\Gamma \triangleright x : U}\ \mathsf{Var}
\qquad
\frac{\Gamma, x : A \triangleright e : B}{\Gamma \triangleright \lambda x.\, e : A\to B}\ {\to}\mathsf{I}
\qquad
\frac{\Gamma \triangleright e : A\to B \quad \Gamma \triangleright e' : A}{\Gamma \triangleright e\, e' : B}\ {\to}\mathsf{E}
$$

$$
\frac{\Gamma \triangleright e : U \quad \Gamma, x : U \triangleright e' : A}{\Gamma \triangleright \mathsf{let}\ x = e\ \mathsf{in}\ e' : A}\ \mathsf{Let}
\qquad
\frac{\Gamma, \alpha\ \mathsf{type} \triangleright e : U}{\Gamma \triangleright e : \forall\alpha.U}\ \mathsf{Gen}
\qquad
\frac{\Gamma \triangleright e : \forall\alpha.U \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma \triangleright e : [A/\alpha]U}\ \mathsf{Spec}
$$

Note that unlike in the predicative polymorphic λ-calculus, the rule ${\to}\mathsf{I}$ allows us to assign *any* monotype A to variable x as long as expression e is assigned a valid monotype B. Hence, for example, the same λ-abstraction $\lambda x.\, x$ can now be assigned different monotypes such as $\mathsf{bool}\to\mathsf{bool}$, $\mathsf{int}\to\mathsf{int}$, and $\alpha\to\alpha$. The rules Gen and Spec correspond to the rules $\forall\mathsf{I}$ and $\forall\mathsf{E}$ in the predicative polymorphic λ-calculus (but not in System F because A in the rule Spec is required to be a monotype).

In the rule Gen (for generalizing a type), expression e in the conclusion plays the role of a type abstraction. That is, we can think of e in the conclusion as $erase(\Lambda\alpha.\, e)$. As an example, let us assign a polytype to the polymorphic identity function $\lambda x.\, x$:

$$
\Gamma \triangleright \lambda x.\, x : ?
$$

Intuitively $\lambda x.\,x$ has type $\alpha \to \alpha$ for an "any type" α, so we first assign a *monotype* $\alpha \to \alpha$ under the assumption that α is a valid type variable:

$$\frac{\dfrac{}{\Gamma, \alpha \text{ type}, x : \alpha \rhd x : \alpha} \text{ Var}}{\Gamma, \alpha \text{ type} \rhd \lambda x.\,x : \alpha \to \alpha} \to\!\mathsf{I}$$

Note that $\lambda x.\,x$ has not been assigned a polytype yet. Also note that $\Gamma \rhd \lambda x.\,x : \alpha \to \alpha$ cannot be a valid typing derivation because α is a fresh type variable which is not declared in Γ. Assigning a polytype $\forall \alpha.\alpha \to \alpha$ to $\lambda x.\,x$ is accomplished by the rule Gen:

$$\frac{\dfrac{\dfrac{}{\Gamma, \alpha \text{ type}, x : \alpha \rhd x : \alpha} \text{ Var}}{\Gamma, \alpha \text{ type} \rhd \lambda x.\,x : \alpha \to \alpha} \to\!\mathsf{I}}{\Gamma \rhd \lambda x.\,x : \forall \alpha.\alpha \to \alpha} \text{ Gen}$$

As an example of using two type variables, we assign a polytype $\forall \alpha.\forall \beta.\alpha \to \beta \to (\alpha \times \beta)$ to $\lambda x.\,\lambda y.\,(x, y)$ as follows (where we assume that product types are available):

$$\frac{\dfrac{\dfrac{\dfrac{}{\Gamma, \alpha \text{ type}, \beta \text{ type}, x : \alpha, y : \beta \rhd x : \alpha} \text{ Var} \quad \dfrac{}{\Gamma, \alpha \text{ type}, \beta \text{ type}, x : \alpha, y : \beta \rhd y : \beta} \text{ Var}}{\Gamma, \alpha \text{ type}, \beta \text{ type}, x : \alpha, y : \beta \rhd (x, y) : (\alpha \times \beta)} \times\!\mathsf{I}}{\dfrac{\Gamma, \alpha \text{ type}, \beta \text{ type}, x : \alpha \rhd \lambda y.\,(x, y) : \beta \to (\alpha \times \beta)}{\dfrac{\Gamma, \alpha \text{ type}, \beta \text{ type} \rhd \lambda x.\,\lambda y.\,(x, y) : \alpha \to \beta \to (\alpha \times \beta)}{\dfrac{\Gamma, \alpha \text{ type} \rhd \lambda x.\,\lambda y.\,(x, y) : \forall \beta.\alpha \to \beta \to (\alpha \times \beta)}{\Gamma \rhd \lambda x.\,\lambda y.\,(x, y) : \forall \alpha.\forall \beta.\alpha \to \beta \to (\alpha \times \beta)} \text{ Gen}} \text{ Gen}} \to\!\mathsf{I}} \to\!\mathsf{I}}$$

Generalizing the example, we can assign a polytype to an expression e in two steps. First we introduce as many fresh type variables as necessary to assign a monotype A to e. Then we keep applying the rule Gen to convert, or generalize, A to a polytype U. If A uses fresh type variables $\alpha_1, \alpha_2, \cdots, \alpha_n$, then U is given as $\forall \alpha_1.\forall \alpha_2.\cdots \forall \alpha_n.A$:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \alpha_1 \text{ type}, \alpha_2 \text{ type}, \cdots, \alpha_n \text{ type} \rhd e : A}{\Gamma, \alpha_1 \text{ type}, \alpha_2 \text{ type}, \cdots \rhd e : \forall \alpha_n.A} \text{ Gen}}{\vdots} \text{ Gen}}{\Gamma, \alpha_1 \text{ type}, \alpha_2 \text{ type} \rhd e : \cdots \forall \alpha_n.A} \text{ Gen}}{\Gamma, \alpha_1 \text{ type} \rhd e : \forall \alpha_2.\cdots \forall \alpha_n.A} \text{ Gen}}{\Gamma \rhd e : \forall \alpha_1.\forall \alpha_2.\cdots \forall \alpha_n.A} \text{ Gen}}$$

In the rule Spec (for specializing a type), expression e in the conclusion plays the role of a type application. That is, we can think of e in the conclusion as $erase(e \, [\![A]\!])$. Thus, by applying the rule Spec repeatedly, we can convert, or specialize, any polytype into a monotype.

A typical use of the rule Spec is to specialize the polytype of a variable introduced in a let-binding (in which case expression e in the rule Spec is a variable). Specifically a let-binding $\text{let } x = e \text{ in } e'$ binds variable x to a polymorphic expression e and uses x monomorphically within e' after specializing the type of x to monotypes by the rule Spec. For example, the following typing derivation for $\text{let } f = \lambda x.\,x \text{ in } (f \text{ true}, f \, 0)$ applies the rule Spec to variable f twice, where we abbreviate $\Gamma, f : \forall \alpha.\alpha \to \alpha$ as Γ':

$$\frac{\dfrac{\vdots}{\Gamma \rhd \lambda x.\,x : \forall \alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \rhd f : \forall \alpha.\alpha \to \alpha} \text{ Var}}{\Gamma' \rhd f : \text{bool} \to \text{bool}} \text{ Spec} \quad \dfrac{}{\Gamma' \rhd \text{true} : \text{bool}} \text{ True}}{\Gamma' \rhd f \text{ true} : \text{bool}} \to\!\mathsf{E} \quad \dfrac{\dfrac{\dfrac{}{\Gamma' \rhd f : \forall \alpha.\alpha \to \alpha} \text{ Var}}{\Gamma' \rhd f : \text{int} \to \text{int}} \text{ Spec} \quad \dfrac{}{\Gamma' \rhd 0 : \text{int}} \text{ Int}}{\Gamma' \rhd f \, 0 : \text{int}} \to\!\mathsf{E}}{\Gamma' \rhd (f \text{ true}, f \, 0) : \text{bool} \times \text{int}} \times\!\mathsf{I}}{\Gamma \rhd \text{let } f = \lambda x.\,x \text{ in } (f \text{ true}, f \, 0) : \text{bool} \times \text{int}} \text{ Let}$$

Note that in typechecking $(f \text{ true}, f \, 0)$, it is mandatory to specialize the type of f to a monotype $\text{bool} \to \text{bool}$ or $\text{int} \to \text{int}$, since an application $f \, e$ typechecks by the rule $\to\!\mathsf{E}$ only if f is assigned a monotype.

If expression e in the rule Spec is not a variable, the typing derivation of the premise $\Gamma \triangleright e : \forall \alpha.U$ must end with an application of the rule Gen or another application of the rule Spec. In such a case, we can eventually locate an application of the rule Gen that is immediately followed by an application of the rule Spec:

$$
\cfrac{\cfrac{\vdots}{\cfrac{\Gamma, \alpha\ \mathsf{type} \triangleright e : U}{\Gamma \triangleright e : \forall \alpha.U}\ \mathsf{Gen}} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma \triangleright e : [A/\alpha]U}\ \mathsf{Spec}
$$

Here we introduce a type variable α only to instantiate it to a concrete monotype A immediately, which implies that such a typing derivation is redundant and can be removed. For example, when typechecking $(\lambda x.\, x)$ true, there is no need to take a detour by first assigning a polytype $\forall \alpha.\alpha \to \alpha$ to $\lambda x.\, x$ and then instantiating α to bool. Instead it suffices to assign a monotype bool$\to$bool directly because $\lambda x.\, x$ is eventually applied to an argument of type bool:

$$
\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma, \alpha\ \mathsf{type}, x : \alpha \triangleright x : \alpha}\ \mathsf{Var}}{\Gamma, \alpha\ \mathsf{type} \triangleright \lambda x.\, x : \alpha \to \alpha}\ {\to}\mathsf{I}}{\Gamma \triangleright \lambda x.\, x : \forall \alpha.\alpha \to \alpha}\ \mathsf{Gen} \quad \Gamma \vdash \mathsf{bool}\ \mathsf{type}}{\Gamma \triangleright \lambda x.\, x : \mathsf{bool} \to \mathsf{bool}}\ \mathsf{Spec} \quad \Longrightarrow \quad \cfrac{\cfrac{}{\Gamma, x : \mathsf{bool} \triangleright x : \mathsf{bool}}\ \mathsf{Var}}{\Gamma \triangleright \lambda x.\, x : \mathsf{bool} \to \mathsf{bool}}\ {\to}\mathsf{I}
$$

This observation suggests that it is unnecessary to specialize the type of an expression that is not a variable — we only need to apply the rule Spec to polymorphic variables introduced in let-bindings.

The implicit polymorphic type system of SML is connected with let-polymorphism in Section 13.5 via the following theorems:

Theorem 13.10. *If $\Gamma \vdash e : U$, then $\Gamma \triangleright erase(e) : U$.*

Theorem 13.11. *If $\Gamma \triangleright e : U$, then there exists a typed expression e' such that $\Gamma \vdash e' : U$ and $erase(e') = e$.*

13.7 Value restriction

The type system presented in the previous section is sound only if it does not interact with computational effects such as mutable references and input/output. To see the problem, consider the following expression where we assume constructs for integers, booleans, and mutable references:

$$
\begin{aligned}
&\mathsf{let}\ x = \mathsf{ref}\ (\lambda y.\, y)\ \mathsf{in} \\
&\mathsf{let}\ \_ = x := \lambda y.\, y + 1\ \mathsf{in} \\
&(!x)\ \mathsf{true}
\end{aligned}
$$

We can assign a polytype $\forall \alpha.\mathsf{ref}\ (\alpha \to \alpha)$ to ref $(\lambda y.\, y)$ as follows (where we ignore store typing contexts):

$$
\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma, \alpha\ \mathsf{type}, y : \alpha \triangleright y : \alpha}\ \mathsf{Var}}{\Gamma, \alpha\ \mathsf{type} \triangleright \lambda y.\, y : \alpha \to \alpha}\ {\to}\mathsf{I}}{\Gamma, \alpha\ \mathsf{type} \triangleright \mathsf{ref}\ (\lambda y.\, y) : \mathsf{ref}\ (\alpha \to \alpha)}\ \mathsf{Ref}}{\Gamma \triangleright \mathsf{ref}\ (\lambda y.\, y) : \forall \alpha.\mathsf{ref}\ (\alpha \to \alpha)}\ \mathsf{Gen}
$$

By the rule Spec, then, we can assign either ref (int$\to$int) or ref (bool$\to$bool) to variable x. Now both expressions $x := \lambda y.\, y + 1$ and $(!x)$ true are well-typed, but the reduction of $(!x)$ true must not succeed because it ends up adding a boolean truth true and an integer 1!

In order to avoid the problem arising from the interaction between polymorphism and computational effects, the type system of SML imposes a requirement, called *value restriction*, that expression e in the rule Gen be a syntactic value:

$$
\cfrac{\Gamma, \alpha\ \mathsf{type} \triangleright v : U}{\Gamma \triangleright v : \forall \alpha.U}\ \mathsf{Gen}
$$

The idea is to exploit the fact that computational effects cannot interfere with (polymorphic) values, whose evaluation terminates immediately. Now, for example, ref $(\lambda y.\, y)$ cannot be assigned a polytype because ref $(\lambda y.\, y)$ is not a value and thus its type cannot be generalized by the rule Gen.

As a consequence of value restriction, variable x in a let-binding let $x = e$ in e' can be assigned a polytype only if expression e is a value. If e is not a value, x must be used monomorphically within expression e', *even if e itself does not specify a unique monotype.* This means that we may have to analyze e' in order to decide the monotype to be assigned to x. As an example, consider the following expression:

$$\text{let } x = (\lambda y.\, y)\, (\lambda z.\, z) \text{ in } x \text{ true}$$

As $(\lambda y.\, y)\, (\lambda z.\, z)$ is not a value, variable x must be assigned a monotype. $(\lambda y.\, y)\, (\lambda z.\, z)$, however, does not specify a unique monotype for x; it only specifies that the type of x must be of the form $A \to A$ for some monotype A. Fortunately the application x true fixes such a monotype A as bool and x is assigned a unique monotype bool $\to$ bool. The following expression, in contrast, is ill-typed because variable x is used polymorphically:

$$\text{let } x = (\lambda y.\, y)\, (\lambda z.\, z) \text{ in } (x \text{ true}, x\, 1)$$

The problem here is that x needs to be assigned two monotypes bool $\to$ bool and int $\to$ int simultaneously, which is clearly out of the question.

13.8 Type reconstruction algorithm

This section presents a type reconstruction algorithm for the type system with implicit polymorphism in Section 13.6. Given an untyped expression e, the goal is to infer a polytype U such that $\cdot \rhd e : U$ holds. In addition to being a valid type for e, U also needs to be the most general type for e in the sense that every valid type for e can be obtained as a special case of U by instantiating some type variables in U. Given $\lambda x.\, x$ as input, for example, the algorithm returns the most general polytype $\forall \alpha.\alpha \to \alpha$ instead of a specific monotype such as bool $\to$ bool.

Typically the algorithm creates (perhaps a lot of) temporary type variables before finding the most general type of a given expression. We design the algorithm in such a way that all these temporary type variables are valid (simply because there is no reason to create invalid ones). As a result, we no longer need a type declaration α type in the rule Gen (because α is assumed to be a valid type variable) and a type judgment $\Gamma \vdash A$ type in the rule Spec (because A is a valid type if all type variables in it are valid). Accordingly a typing context now consists only of type bindings:

$$\text{typing context} \qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : U$$

With the assumption that all type variables are valid, the rules Gen and Spec are revised as follows:

$$\frac{\Gamma \rhd e : U \quad \alpha \notin ftv(\Gamma)}{\Gamma \rhd e : \forall \alpha.U} \text{ Gen} \qquad \frac{\Gamma \rhd e : \forall \alpha.U}{\Gamma \rhd e : [A/\alpha]U} \text{ Spec}$$

Here $ftv(\Gamma)$ denotes the set of free type variables in Γ; $ftv(U)$ denotes the set of free type variables in U:

$$
\begin{array}{rcl}
ftv(\cdot) & = & \varnothing \\
ftv(\Gamma, x : U) & = & ftv(\Gamma) \cup ftv(U)
\end{array}
\qquad
\begin{array}{rcl}
ftv(A \to B) & = & ftv(A) \cup ftv(B) \\
ftv(\alpha) & = & \{\alpha\} \\
ftv(\forall \alpha.U) & = & ftv(U) - \{\alpha\}
\end{array}
$$

In the rule Gen, the condition $\alpha \notin ftv(\Gamma)$ checks that α is a fresh type variable. If Γ contains a type binding $x : U$ where α is already in use as a free type variable in U, α cannot be regarded as a fresh type variable and generalizing U to $\forall \alpha.U$ is not justified. In the following example, $\alpha \to \alpha$ may generalize to $\forall \alpha.\alpha \to \alpha$, assigning the desired polytype to the polymorphic identity function $\lambda x.\, x$, because α is not in use in the empty typing context $\cdot$:

$$\frac{\dfrac{\overline{x : \alpha \rhd x : \alpha} \text{ Var}}{\cdot \rhd \lambda x.\, x : \alpha \to \alpha} \to \text{I}}{\cdot \rhd \lambda x.\, x : \forall \alpha.\alpha \to \alpha} \text{ Gen}$$

If α is already in use as a free type variable, however, such a generalization results in assigning a wrong type to $\lambda x.\, x$:

$$\dfrac{\dfrac{\dfrac{x : \alpha, y : \alpha \rhd x : \alpha}{x : \alpha \rhd \lambda y.\, x : \alpha \to \alpha} \; \text{Var}}{\rule{0pt}{0pt}} \; \to\!\text{I}}{x : \alpha \rhd \lambda y.\, x : \forall \alpha.\alpha \to \alpha} \; \text{Gen}$$

Here variable y is unrelated to variable x, yet is assigned the same type in the premise of the rule $\to\!\text{I}$. A correct typing derivation assigns a fresh type variable to y to reflect the fact that x and y are unrelated:

$$\dfrac{\dfrac{\dfrac{x : \alpha, y : \beta \rhd x : \alpha}{x : \alpha \rhd \lambda y.\, x : \beta \to \alpha} \; \text{Var}}{\rule{0pt}{0pt}} \; \to\!\text{I}}{x : \alpha \rhd \lambda y.\, x : \forall \beta.\beta \to \alpha} \; \text{Gen}$$

As an example of applying the rule Spec, here is a typing derivation assigning a monotype $\text{bool} \to \text{bool}$ to $\lambda x.\, x$ by instantiating a type variable:

$$\dfrac{\cdot \rhd \lambda x.\, x : \forall \alpha.\alpha \to \alpha}{\cdot \rhd \lambda x.\, x : \text{bool} \to \text{bool}} \; \text{Spec}$$

For the above example, the rule Spec is unnecessary because we can directly assign bool to variable x:

$$\dfrac{\dfrac{x : \text{bool} \rhd x : \text{bool}}{\rule{0pt}{0pt}} \; \text{Var}}{\cdot \rhd \lambda x.\, x : \text{bool} \to \text{bool}} \; \to\!\text{I}$$

As we have seen in Section 13.6, however, the rule Spec is indispensable for specializing the type of a variable introduced in a let-binding. In the following example, the same type variable α in $\forall \alpha.\alpha \to \alpha$ is instantiated to two different types $\beta \to \beta$ and β by the rule Spec:

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \alpha \rhd x : \alpha}{\cdot \rhd \lambda x.\, x : \alpha \to \alpha} \; \text{Var}}{\cdot \rhd \lambda x.\, x : \forall \alpha.\alpha \to \alpha} \; \text{Gen} \quad \dfrac{\dfrac{\dfrac{f : \forall \alpha.\alpha \to \alpha \rhd f : \forall \alpha.\alpha \to \alpha}{f : \forall \alpha.\alpha \to \alpha \rhd f : (\beta \to \beta) \to (\beta \to \beta)} \; \text{Spec} \quad \dfrac{\dfrac{f : \forall \alpha.\alpha \to \alpha \rhd f : \forall \alpha.\alpha \to \alpha}{f : \forall \alpha.\alpha \to \alpha \rhd f : \beta \to \beta} \; \text{Spec}}{\rule{0pt}{0pt}}}{f : \forall \alpha.\alpha \to \alpha \rhd f\, f : \beta \to \beta} \; \to\!\text{E}}{\cdot \rhd \text{let } f = \lambda x.\, x \text{ in } f\, f : \beta \to \beta}} \; \text{Let}$$

Exercise 13.12. What is wrong with the following typing derivation?

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \forall \alpha.\alpha \to \alpha \rhd x : \forall \alpha.\alpha \to \alpha}{x : \forall \alpha.\alpha \to \alpha \rhd x : (\forall \alpha.\alpha \to \alpha) \to (\forall \alpha.\alpha \to \alpha)} \; \text{Var}}{\rule{0pt}{0pt}} \; \text{Spec} \quad \dfrac{x : \forall \alpha.\alpha \to \alpha \rhd x : \forall \alpha.\alpha \to \alpha}{\rule{0pt}{0pt}} \; \text{Var}}{x : \forall \alpha.\alpha \to \alpha \rhd x\, x : \forall \alpha.\alpha \to \alpha} \; \to\!\text{E}}{\cdot \rhd \lambda x.\, x\, x : (\forall \alpha.\alpha \to \alpha) \to (\forall \alpha.\alpha \to \alpha)} \; \to\!\text{I}$$

The type reconstruction algorithm, called $\mathcal{W}$, takes a typing context Γ and an expression e as input, and returns a pair of a *type substitution* S and a monotype A as output:

$$\mathcal{W}(\Gamma, e) \;=\; (S, A)$$

A type substitution is a mapping from type variables to monotypes. Note that it is not a mapping to polytypes because type variables range only over monotypes:

$$\text{type substitution} \quad S \quad ::= \quad \text{id} \mid \{A/\alpha\} \mid S \circ S$$

id is an identity type substitution which changes no type variable. $\{A/\alpha\}$ is a singleton type substitution which maps α to A. $S_1 \circ S_2$ is a composition of S_1 and S_2 which applies first S_2 and then S_1. id is the identity for the composition operator $\circ$, *i.e.*, $\text{id} \circ S = S \circ \text{id} = S$. As $\circ$ is associative, we write $S_1 \circ S_2 \circ S_3$ for $S_1 \circ (S_2 \circ S_3) = (S_1 \circ S_2) \circ S_3$.

An application of a type substitution to a polytype U is formally defined as follows:

$$
\begin{aligned}
\mathrm{id} \cdot U &= U \\
\{A/\alpha\} \cdot \alpha &= A \\
\{A/\alpha\} \cdot \beta &= \beta && \text{where } \alpha \neq \beta \\
\{A/\alpha\} \cdot B_1 \rightarrow B_2 &= \{A/\alpha\} \cdot B_1 \rightarrow \{A/\alpha\} \cdot B_2 \\
\{A/\alpha\} \cdot \forall\alpha.U &= \forall\alpha.U \\
\{A/\alpha\} \cdot \forall\beta.U &= \forall\beta.\{A/\alpha\} \cdot U && \text{where } \alpha \neq \beta \text{ and } \beta \notin \mathit{ftv}(A) \\
S_1 \circ S_2 \cdot U &= S_1 \cdot (S_2 \cdot U)
\end{aligned}
$$

Note that if β is a free type variable in A, the case $\{A/\alpha\} \cdot \forall\beta.U$ needs to rename the bound type variable β in order to avoid type variable captures. When applied to a typing context, a type substitution is applied to the type in each type binding:

$$
S \cdot (\Gamma, x : U) \;\; = \;\; S \cdot \Gamma, x : (S \cdot U)
$$

The specification of the algorithm $\mathcal{W}$ is concisely stated in its soundness theorem:

Theorem 13.13 (Soundness of $\mathcal{W}$). *If $\mathcal{W}(\Gamma, e) = (S, A)$, then $S \cdot \Gamma \triangleright e : A$.*

Given a typing context Γ and an expression e, the algorithm analyzes e to build a type substitution S mapping free type variables in Γ so that e typechecks with a monotype A. An invariant is that S has no effect on A, i.e., $S \cdot A = A$, since A is obtained *after* applying S to free type variables in Γ. Here are a few examples:

- $\mathcal{W}(x : \alpha, x + 0) = (\{\mathsf{int}/\alpha\}, \mathsf{int})$ where we assume a base type int.
  When the algorithm starts, x has been assigned a yet unknown monotype α. In the course of analyzing $x + 0$, the algorithm discovers that x must be assigned type int, in which case $x + 0$ is also assigned type int. Thus the algorithm returns a type substitution $\{\mathsf{int}/\alpha\}$ along with int as the type of $x + 0$.

- $\mathcal{W}(\cdot, \lambda x.\, x + 0) = (\{\mathsf{int}/\alpha\}, \mathsf{int} \rightarrow \mathsf{int})$ where we assume a base type int.
  When it starts to analyze the λ-abstraction, the algorithm creates a fresh type variable, say α, for variable x because nothing is known about x yet. In the course of analyzing the body $x + 0$, the algorithm discovers that α must be identified with int, in which case the type of the λ-abstraction becomes $\mathsf{int} \rightarrow \mathsf{int}$. Hence the algorithm returns a type substitution $\{\mathsf{int}/\alpha\}$ (which is not used afterwards) with $\mathsf{int} \rightarrow \mathsf{int}$ as the type of $\lambda x.\, x + 0$.

- $\mathcal{W}(\cdot, \lambda x.\, x) = (\mathrm{id}, \alpha \rightarrow \alpha)$
  When it starts to analyze the λ-abstraction, the algorithm creates a fresh type variable, say α, for variable x because nothing is known about x yet. The body x, however, provides no information on the type of x, either, and the algorithm ends up returning $\alpha \rightarrow \alpha$ as a possible type of $\lambda x.\, x$.

Exercise 13.14. What is the result of $\mathcal{W}(y : \beta, (\lambda x.\, x)\, y)$ if the algorithm $\mathcal{W}$ creates a temporary type variable α for variable x? Is the result unique?

Figure 13.1 shows the pseudocode of the algorithm $\mathcal{W}$. We write $\vec{\alpha}$ for a sequence of distinct type variables $\alpha_1, \alpha_2, \cdots, \alpha_n$. Then $\forall\vec{\alpha}.A$ stands for $\forall\alpha_1.\forall\alpha_2.\cdots\forall\alpha_n.A$, and $\{\vec{\beta}/\vec{\alpha}\}$ stands for $\{\beta_n/\alpha_n\} \circ \cdots \circ \{\beta_2/\alpha_2\} \circ \{\beta_1/\alpha_1\}$. We write $\Gamma + x : U$ for $\Gamma - \{x : U'\}, x : U$ if $x : U' \in \Gamma$, and for $\Gamma, x : U$ if Γ contains no type binding for variable x.

The first case $\mathcal{W}(\Gamma, x)$ summarizes the result of applying the rule Spec to $\forall\vec{\alpha}.A$ as many times as the

$$
\begin{aligned}
\mathcal{W}(\Gamma, x) &= (\mathrm{id}, \{\vec{\beta}/\vec{\alpha}\} \cdot A) && x : \forall \vec{\alpha}.A \in \Gamma \text{ and fresh } \vec{\beta} \\
\mathcal{W}(\Gamma, \lambda x.\, e) &= \text{let } (S, A) = \mathcal{W}(\Gamma + x : \alpha, e) \text{ in} && \text{fresh } \alpha \\
&\quad\; (S, (S \cdot \alpha) \to A) \\
\mathcal{W}(\Gamma, e_1\, e_2) &= \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in} \\
&\quad\; \text{let } (S_2, A_2) = \mathcal{W}(S_1 \cdot \Gamma, e_2) \text{ in} \\
&\quad\; \text{let } S_3 = \mathsf{Unify}(S_2 \cdot A_1 = A_2 \to \alpha) \text{ in} && \text{fresh } \alpha \\
&\quad\; (S_3 \circ S_2 \circ S_1, S_3 \cdot \alpha) \\
\mathcal{W}(\Gamma, \text{let } x = e_1 \text{ in } e_2) &= \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in} \\
&\quad\; \text{let } (S_2, A_2) = \mathcal{W}(S_1 \cdot \Gamma + x : \mathsf{Gen}_{S_1 \cdot \Gamma}(A_1), e_2) \text{ in} \\
&\quad\; (S_2 \circ S_1, A_2)
\end{aligned}
$$

Figure 13.1: Algorithm $\mathcal{W}$

$$
\begin{aligned}
\mathsf{Unify}(\cdot) &= \mathrm{id} \\
\mathsf{Unify}(E, \alpha = A) = \mathsf{Unify}(E, A = \alpha) &= \text{if } \alpha = A \text{ then } \mathsf{Unify}(E) \\
&\qquad \text{else if } \alpha \in \mathit{ftv}(A) \text{ then } \textit{fail} \\
&\qquad \text{else } \mathsf{Unify}(\{A/\alpha\} \cdot E) \circ \{A/\alpha\} \\
\mathsf{Unify}(E, A_1 \to A_2 = B_1 \to B_2) &= \mathsf{Unify}(E, A_1 = B_1, A_2 = B_2)
\end{aligned}
$$

Figure 13.2: Algorithm Unify

length of $\vec{\alpha}$. Note that $[A/\alpha]U$ is written as $\{A/\alpha\} \cdot U$ in the following typing derivation.

$$
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{
\cfrac{x : \forall \alpha_1.\forall \alpha_2. \cdots \forall \alpha_n.A \in \Gamma}
{\Gamma \triangleright x : \forall \alpha_1.\forall \alpha_2. \cdots \forall \alpha_n.A} \; \mathsf{Var}}
{\Gamma \triangleright x : \{\beta_1/\alpha_1\} \cdot \forall \alpha_2. \cdots \forall \alpha_n.A} \; \mathsf{Spec}}
{\Gamma \triangleright x : \{\beta_2/\alpha_2\} \circ \{\beta_1/\alpha_1\} \cdot \cdots \forall \alpha_n.A} \; \mathsf{Spec}}
{\vdots} \; \mathsf{Spec}}
{\Gamma \triangleright x : \cdots \circ \{\beta_2/\alpha_2\} \circ \{\beta_1/\alpha_1\} \cdot \forall \alpha_n.A} \; \mathsf{Spec}}
{\Gamma \triangleright x : \{\beta_n/\alpha_n\} \circ \cdots \circ \{\beta_2/\alpha_2\} \circ \{\beta_1/\alpha_1\} \cdot A} \; \mathsf{Spec}
$$

The second case $\mathcal{W}(\Gamma, \lambda x.\, e)$ creates a fresh type variable α to be assigned to variable x.

The case $\mathcal{W}(\Gamma, e_1\, e_2)$ uses an auxiliary function $\mathsf{Unify}(E)$ where E is a set of *type equations* between monotypes:

$$
\text{type equations} \qquad E \quad ::= \quad \cdot \mid E, A = A
$$

$\mathsf{Unify}(E)$ attempts to calculate a type substitution that unifies two types A and A' in each type equation $A = A'$ in E. If no such type substitution exists, $\mathsf{Unify}(E)$ fails. Figure 13.2 shows the definition of $\mathsf{Unify}(E)$. We write $S \cdot E$ for the result of applying type substitution S to every type in E:

$$
S \cdot (E, A = A') = S \cdot E, S \cdot A = S \cdot A'
$$

The specification of the function Unify is stated as follows:

Proposition 13.15. *If* $\mathsf{Unify}(A_1 = A_1', \cdots, A_n = A_n') = S$, *then* $S \cdot A_i = S \cdot A_i'$ *for* $i = 1, \cdots, n$.

Here are a few examples of $\mathsf{Unify}(E)$ where we assume a base type int:

$$
\begin{array}{rrcl}
(1) & \mathsf{Unify}(\alpha = \mathsf{int} \to \alpha) &=& \textit{fail} \\
(2) & \mathsf{Unify}(\alpha = \alpha \to \alpha) &=& \textit{fail} \\
(3) & \mathsf{Unify}(\alpha \to \alpha = \mathsf{int} \to \mathsf{int}) &=& \{\mathsf{int}/\alpha\} \\
(4) & \mathsf{Unify}(\alpha \to \beta = \alpha \to \mathsf{int}) &=& \{\mathsf{int}/\beta\} \\
(5) & \mathsf{Unify}(\alpha \to \beta = \beta \to \alpha) &=& \{\beta/\alpha\} \text{ or } \{\alpha/\beta\} \\
(6) & \mathsf{Unify}(\alpha \to \beta = \beta \to \alpha, \alpha = \mathsf{int}) &=& \{\mathsf{int}/\beta\} \circ \{\mathsf{int}/\alpha\}
\end{array}
$$

In cases (1) and (2), the unification fails because both $\mathsf{int}\to\alpha$ and $\alpha\to\alpha$ contain α as a free type variable, but are strictly larger than α. In case (5), either $\{\beta/\alpha\}$ or $\{\alpha/\beta\}$ successfully unifies $\alpha\to\beta$ and $\beta\to\alpha$. Case (6) uses an additional assumption $\mathsf{Unify}(E, \mathsf{int} = \mathsf{int}) = \mathsf{Unify}(E)$:

$$
\begin{aligned}
\mathsf{Unify}(\alpha\to\beta = \beta\to\alpha, \alpha = \mathsf{int}) &= \mathsf{Unify}(\{\mathsf{int}/\alpha\} \cdot (\alpha\to\beta = \beta\to\alpha)) \circ \{\mathsf{int}/\alpha\} \\
&= \mathsf{Unify}(\mathsf{int}\to\beta = \beta\to\mathsf{int}) \circ \{\mathsf{int}/\alpha\} \\
&= \mathsf{Unify}(\mathsf{int} = \beta, \beta = \mathsf{int}) \circ \{\mathsf{int}/\alpha\} \\
&= \mathsf{Unify}(\{\mathsf{int}/\beta\} \cdot \mathsf{int} = \beta) \circ \{\mathsf{int}/\beta\} \circ \{\mathsf{int}/\alpha\} \\
&= \mathsf{Unify}(\mathsf{int} = \mathsf{int}) \circ \{\mathsf{int}/\beta\} \circ \{\mathsf{int}/\alpha\} \\
&= \mathsf{Unify}(\cdot) \circ \{\mathsf{int}/\beta\} \circ \{\mathsf{int}/\alpha\} \\
&= \mathsf{id} \circ \{\mathsf{int}/\beta\} \circ \{\mathsf{int}/\alpha\} \\
&= \{\mathsf{int}/\beta\} \circ \{\mathsf{int}/\alpha\}
\end{aligned}
$$

The case $\mathcal{W}(\Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$ uses another auxiliary function $\mathsf{Gen}_\Gamma(A)$ which generalizes monotype A to a polytype after taking into account free type variables in typing context Γ:

$$\mathsf{Gen}_\Gamma(A) = \forall\alpha_1.\forall\alpha_2.\cdots\forall\alpha_n.A \text{ where } \alpha_i \notin \mathit{ftv}(\Gamma) \text{ and } \alpha_i \in \mathit{ftv}(A) \text{ for } i = 1, \cdots, n.$$

That is, if $\alpha \in \mathit{ftv}(A)$ is in $\mathit{ftv}(\Gamma)$, $\alpha \in A$ is not interpreted as "any type" with respect to Γ. Note that $\mathsf{Gen}_\Gamma(A) = \forall\alpha_1.\forall\alpha_2.\cdots\forall\alpha_n.A$ is equivalent to applying the rule Gen exactly n times as follows:

$$
\cfrac{\cfrac{\cfrac{\Gamma \rhd e : A \quad \alpha_n \notin \mathit{ftv}(\Gamma)}{\Gamma \rhd e : \forall\alpha_n.A}\ \mathsf{Gen} \quad \alpha_{n-1} \notin \mathit{ftv}(\Gamma)}{\Gamma \rhd e : \forall\alpha_{n-1}.A}\ \mathsf{Gen}}{\cfrac{\vdots}{\Gamma \rhd e : \forall\alpha_1.\forall\alpha_2.\cdots\forall\alpha_n.A}\ \mathsf{Gen}}
$$

Here are a few examples of $\mathsf{Gen}_\Gamma(A)$:

$$
\begin{aligned}
\mathsf{Gen}_{\cdot}(\alpha\to\alpha) &= \forall\alpha.\alpha\to\alpha \\
\mathsf{Gen}_{x:\alpha}(\alpha\to\alpha) &= \alpha\to\alpha \\
\mathsf{Gen}_{x:\alpha}(\alpha\to\beta) &= \forall\beta.\alpha\to\beta \\
\mathsf{Gen}_{x:\alpha,y:\beta}(\alpha\to\beta) &= \alpha\to\beta
\end{aligned}
$$

Given an expression e, the algorithm $\mathcal{W}$ returns a monotype A which may contain free type variables. If we wish to obtain the most general polytype for e, it suffices to generalize A with respect to the given typing context. Specifically, if $\mathcal{W}(\Gamma, e) = (S, A)$ holds, Theorem 13.13 justifies $S \cdot \Gamma \rhd e : A$, which in turn justifies $S \cdot \Gamma \rhd e : \mathsf{Gen}_{S\cdot\Gamma}(A)$. Hence we may take $\mathsf{Gen}_{S\cdot\Gamma}(A)$ as the most general type for e under typing context Γ, although we do not formally prove this property (called the completeness of $\mathcal{W}$) here.