Introduction to Programming Languages

Jaemin Hong and Sukyoung Ryu

August 27, 2021

Acknowledgement

The contents of this book are based on the KAIST *Programming Languages* course. We thank PLT[1] since the course referred to many materials from PLT in the beginning. We also thank every student who took the course before. We have learned many things from the interaction with the students, and those lessons have affected various parts of the book. In addition, we thank all the previous and current teaching assistants of the course. They gave opinions to the course and wrote some of the exercises in the book. Especially, Jihyeok Park highly contributed to the course, and Jihee Park helped us edit the exercises.

We would be delighted to receive comments and corrections, which may be sent to `jaemin.hong@kaist.ac.kr`. We thank in advance everyone who will contribute to the book in the future.

1: https://racket-lang.org/people.html

Contents

1 Introduction

What is a programming language?

The simplest answer is "it is a language used for programming." However, this answer does not help us understand programming languages. We need a better question to get a better answer.

What does a programming language consist of?

There is a good answer for this question: "in a narrow sense, a programming language consists of syntax and semantics, and in a broad sense, it additionally has a standard library and an ecosystem."

Syntax and semantics are principal concepts to understand programming languages. Syntax determines how a language looks like, and semantics fills the inside. If we consider a programming language as a human, we can say that syntax is one's appearance, and semantics is one's thoughts. Programmers write programs according to syntax. Syntax decides characters used in source code. Once programs are written, semantics decides what each program does. Without semantics, all the programs are useless. Programs can work as being expected only after semantics determines the meaning of them. A programming language with syntax and semantics is complete. Programmers using that language can write programs with the syntax and execute the programs with the semantics. From a theoretical perspective, syntax and semantics are all of a programming language.

For programmers, syntax and semantics are not the only elements of a programming language. First, the standard library of a language is another element. The standard library provides various utilities required by applications: data structures like lists and maps, functions handling file and network IO, and so on. The standard library is like clothes for humans. A human without clothes is a human; a programming language without a standard library is a programming language. At the same time, clothes are important to humans as they make bodies warm and protect bodies from dangerous objects. Similarly, a standard library is important to a programming language as it supplies diverse functionalities for applications. Each person wears clothes different from others, and each language puts different things from other languages in its standard library. Some languages include many utilities in their standard libraries, while others include much less. Some languages treat lists and maps as built-in concepts in their semantics, while others define them with other primitives in their standard libraries. Programmers avoid using a language without a standard library because such a language increases the effort to write programs.

Another important element to programmers is the ecosystem of a programming language. The ecosystem includes everything related to the language: developers and companies using the language, third-party libraries written in the language, and so on. It is like a society for humans.

If many programmers and companies use a programming language, one can easily get help and find complementary materials by using the same language. There will be more chances of cooperative work and employment, too. Third-party libraries also take important roles in software development. The standard library offers only general facilities and often lacks domain-specific features. When a required functionality cannot be found in the standard library, a third-party library can provide the exact functionality. For these reasons, the ecosystem of a programming language is important to programmers.

Practically, the standard library and the ecosystem of a language are important elements. Unlike syntax and semantics, they are not essential. A programming language can exist even without its standard library and ecosystem. However, developers take standard libraries and ecosystems into account as well as syntax and semantics to choose languages they use. From a practical perspective, a programming language consists of syntax, semantics, a standard library, and an ecosystem.

This book is not for helping readers use a specific programming language. It does not recommend a specific programming language, either. This book helps readers learn new programming languages easily. You can acquaint any programming languages once you completely read and understand this book. Obviously, this goal cannot be achieved if the book discusses various languages separately. It is possible only by discussing the underlying principles of every programming language.

The principles of programming languages can be found from their semantics. Each language seems very different from the others, but it is actually not the case. Precisely speaking, their insides are quite the same, while their appearances look different. They look different because their syntax and standard libraries, which determine the appearances, are different. However, their insides, the semantics, fundamentally share the same mathematical principles. If you understand essential concepts residing in the semantics of multiple languages, it is easy to understand and learn new languages.

People who know the key principles and can separate the elements of a language can easily learn programming languages. As an analogy, consider a man learning how to use a computer. It is a big problem if he cannot distinguish a keyboard from a computer. For example, he thinks “to say hello, my right index finger presses the keyboard, my left middle finger presses the keyboard, my right ring finger presses the keyboard three times.” If the layout of the keyboard changes, he should learn the whole computer again. On the other hand, if he knows that a keyboard is just a tool to input text, he will less suffer from the change of the keyboard layout. As he thinks “to say hello, I press H, E, L, L, and O,” he does not need to learn the whole computer again. Of course, he should learn the new keyboard layout, but it will be much easier. In addition, it is straightforward to apply his knowledge to do new things. For example, he will easily figure out “to say lol, I press L, O, and L.” If he does not distinguish a keyboard from a computer, he cannot find any common principles between saying hello and saying lol. Learning programming languages is the same. People who cannot distinguish syntax and semantics believe that they should learn the whole language again when the syntax changes. On the other hand, people who can

distinguish syntax and semantics know that semantics remains the same even if syntax may vary. They know that understanding the principles of semantics is important to learn languages. Becoming familiar with the new syntax is all they need to use a new language fluently.

This book explains the semantics of principal concepts in programming languages. Chapters 2, 3, 4, and 5 introduce the Scala programming language. This book uses Scala to implement interpreters and type checkers of languages introduced in the book. Chapter 6 explains syntax and semantics. Then, the book finally introduces various features of programming languages.

- Chapter 7 introduces identifiers.
- Chapters 8, 9, and 10 introduce functions.
- Chapters 11 and 12 introduce mutation.
- Chapter 13 introduces lazy evaluation.
- Chapters 14, 15, and 16 introduce continuations.
- Chapter 17 introduces De Bruijn indices.
- Chapters 18 and 19 introduce basic type systems.
- Chapter 20 introduces algebraic data types.
- Chapter 21 introduces parametric polymorphism.
- Chapter 22 introduces subtype polymorphism.

Each chapter explains a feature by defining a small language providing the feature. Those languages may seem inconvenient in practice because they are too small. However, the simplicity will allow us to focus on the topic of each chapter.

1.1 Exercises

1. Write the name of a programming language that you have used. What are the pros and cons of the language?
2. Write the names of two programming languages you know and compare them.

Scala

Introduction to Scala 2

This book uses Scala as an implementation language, and this chapter thus introduces the Scala programming language. Scala stands for a **sca**lable **la**nguage [OSV16]. It is a multi-paradigm language that allows both functional and object-oriented styles. This book focuses on the functional nature of Scala. In this chapter, we will see what functional programming is and why this book uses functional programming. In addition, we will install Scala and write simple programs in Scala.

2.1 Functional Programming

What is *functional programming*? According to Wikipedia,

> Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

According to the book Functional Programming in Scala [CB14],

> Functional programming (FP) is based on a simple premise with far-reaching implications: we construct our programs using only pure functions—in other words, functions that have no side effects.

The above two sentences are enough to describe functional programming.

First, consider the phrase "the evaluation of mathematical functions." From the perspective of functional programming, a program is a single mathematical expression and the execution of the program is finding a value denoted by the expression. Each expression consists of zero or more subexpressions and evaluates to a value.

Let us discuss how functional programming is different from imperative programming with code examples.

```
int x = 1;
int y = 2;
if (y < 3)
    x = x + 4;
else
    x = x - 5;
```

The above code is written in C, which represents imperative languages. Imperative programming mimics a way in which computers operate. During the execution of a program, a state, which can be interpreted as the memory of a computer, exists and the execution modifies the state. The execution of the above C program has the following steps:

1. A state that both `x` and `y` are uninitialized
2. A state that `x` is `1` and `y` is uninitialized
3. A state that `x` is `1` and `y` is `2`
4. Since `y < 3` is `true` under the state of the third step, go to the next line.
5. A state that `x` is `5` and `y` is `2`

The state keeps changes throughout the execution of the program. Each line modifies the current state rather than resulting in some value.

```
let x = 1 in
let y = 2 in
if y < 3 then x + 4 else x - 5
```

The above code is written in OCaml, which represents functional languages. A program is an expression and the result of the execution is the result of evaluating the expression. The execution does not require the notion of a state. The execution of the above OCaml program has the following steps:

1. Given the fact that `x` equals `1`, evaluate `let y = 2 in if y < 3 then x + 4 else x - 5`.
2. Given the fact that `x` equals `1` and `y` equals `2`, evaluate `if y < 3 then x + 4 else x - 5`.
3. Given the fact that `x` equals `1` and `y` equals `2`, evaluating `y < 3` yields `true`, and the next step is to evaluate `x + 4`.
4. Given the fact that `x` equals `1` and `y` equals `2`, evaluate `x + 4`.
5. The result is `5`.

There is no state. Each expression consists of subexpressions. The result of an expression is determined by the results of its subexpressions.

Since the programs are simple, two programs look similar, but it is important to understand two different perspectives of what a program is.

Now, look at the phrases "avoids changing-state and mutable data" and "using only pure functions." Functional programming avoids mutable variables, mutable data structures, and mutable objects. The term *mutable* means being able to change. Its opposite is *immutable*, which means not being able to change. States change throughout the execution of programs. In functional programming, states do not exist since things never change. Due to the lack of states, a function always does the same stuff and always returns the same value for the same arguments. Such functions are called pure functions.

In practice, especially for large-scale projects, using only immutable things in the whole code is often inefficient. Most real-world functional languages provide mutation via language constructs like `var` of Scala, `ref` of OCaml, and `set!` and `box` of Racket. However, functional programming uses immutable things in most cases. Even without mutation, we can still express most programs without difficulties.

As we have seen so far, immutability is the most important concept of functional programming. Immutability allows modular programming and eases the reasoning of programs. Because of immutability, programs

that have to be trustworthy or require parallel computing are good applications of functional programming. Chapter 3 will discuss the advantages of immutability in detail and how to write interesting programs without mutation.

There are other important characteristics of functional programming as well as immutability. Use of first-class functions and pattern matching also take the key roles in functional programming. Both first-class functions and pattern matching are valuable as they help abstraction. First-class functions allow programmers to abstact computation; pattern matching allows programmers to abstract data. Because of the ability of abstraction, programs whose input has complex and abstract structures like source code are typically written in functional languages. Chapter 4 and Chapter 5 will respectively discuss first-class functions and pattern matching in Scala.

This book implements interpreters and type checkers. They take source code as input and process the input according to the mathematical semantics of programming languages. It is important to reason about the correctness of interpreters and type checkers. These properties exactly match the strengths of functional programming. It is why this book uses functional programming and Scala.

Before moving on to the next section, let us see how people use functional programming in industry.

Akka[1] is a concurrent, distributed computing library written in Scala. Many companies have been using Akka. Apache Spark,[2] a well-known library for data processing, also is written in Scala. Play[3] is a widely-used web framework based on Akka.

1: https://akka.io/

2: https://spark.apache.org/

3: https://www.playframework.com/

Facebook has developed Infer,[4] a static analyzer for Java, C, C++, and Objective-C, in OCaml. Facebook and other companies including Amazon and Mozila use Infer to find bugs statically in their programs. Facebook has developed also Flow,[5] a static type checker for JavaScript. Jane Street[6] is a financial company well-known in the programming language community and has developed its own software in OCaml. According to the OCaml website,[7] various companies including Docker use OCaml.

4: https://fbinfer.com/

5: https://flow.org/

6: https://www.janestreet.com/

7: http://ocaml.org/learn/companies.html

Haskell Wiki[8] describes that Google, Facebook, Microsoft, Nvidia, and many other companies use Haskell.

8: http://wiki.haskell.org/Haskell_in_industry

Erlang is a functional language for concurrent and parallel computing. Elixir operates on the Erlang virtual machine and is used for the same purpose as Erlang. An article from Code Sync[9] said that various companies including WhatsApp, Pinterest, and Goldman Sachs use Erlang and Elixir.

9: https://codesync.global/media/successful-companies-using-elixir-and-erlang/

2.2 Installation

As Scala programs are compiled to Java bytecode, which runs on the Java Virtual Machine (JVM), you must install Java before installing Scala. Java has various versions. Scala 2.13, which is used in this book, needs JDK 8 or higher. JDK 8 is the most recommended one. The Scala website[10] discusses compatibility issues regarding the other versions.

10: https://docs.scala-lang.org/overviews/jdk-compatibility/overview.html

The Oracle website[11] provides an installation file for JDK 8.

You can donwload an installation file for Scala 2.13 from the Scala website.[12] Note that you need a file in the "Other resources" section at the bottom of the page. On macOS, you may use Homebrew instead. By installing Scala, you can use the Scala REPL, interpreter, and compiler. Section 2.3, Section 2.4, and Section 2.5 will discuss their usages respectively.

Another thing to install is SBT. SBT is a build tool for Scala. An installation file for SBT is available at the SBT website.[13] Section 2.6 will discuss the usage of SBT.

11: https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html

12: https://www.scala-lang.org/download/

13: https://www.scala-sbt.org/download.html

2.3 REPL

Once you install Scala, you can launch Scala REPL by typing `scala` in your command line.

```
$ scala
Welcome to Scala 2.13.5.
Type in expressions for evaluation. Or try :help.

scala>
```

The term REPL stands for **r**ead, **e**val, **p**rint, and **l**oop. It is a program that iterativley reads code from a user, evaluates the code, and prints the result. REPL is not a place to write a program but is a good place to write short code and see how it works.

If you input an integer to REPL, it will evaluate the integer and show the result.

```
scala> 0
val res0: Int = 0
```

It means that the expression `0` evaluates to the value `0` and the type of `0` is `Int`. You can try some arithmetic expressions as well.

```
scala> 1 + 2
val res1: Int = 3
```

A boolean is `true` or `false` in Scala.

```
scala> true
val res2: Boolean = true
```

You can also use basic logical operators.

```
scala> true && false
val res3: Boolean = false
```

String literals require double quotation marks.

```
scala> "hello"
val res4: String = hello
```

Operations regarding strings can be done by calling methods.

```
scala> "hello".length
val res5: Int = 5

scala> "hello".substring(0, 4)
val res6: String = hell
```

Strings in Scala provide the same methods as those in Java.[14]

14: https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

The `println` function prints a given message into the console.

```
scala> println("Hello world!")
Hello world!
```

Note that there is no result of `println("Hello world!")`. Actually, `println("Hello world!")` evaluates to `()`, which is called unit. Unit implies that the result does not have any meaningful information. It is similar to `None` in Python and `undefined` in JavaScript. At the same time, functions returning unit are similar to functions whose return types are `void` in C or Java. Since unit does not have meaningful information, REPL does not show the result when it is unit.

The remainder of this section introduces basic features of Scala, such as variables and functions, with REPL.

Variables

The syntax of a variable definition is as follows:

```
val [name]: [type] = [expression]
```

It defines a variable whose name is `[name]`. The result of the expression becomes the value denoted by the variable and must belong to the type.

```
scala> val x: Int = 1
val x: Int = 1
```

If the type of the result does not match a given type, the variable will not be defined due to a type mismatch.

```
scala> val y: Boolean = 2
                        ^
       error: type mismatch;
        found   : Int(2)
        required: Boolean
```

You can omit the `: [type]` part and use the following syntax instead:

```
val [name] = [expression]
```

In this case, a type mimatch never happens, and the type of the variable becomes the same as the type of its value. People usually omit the type annotations of local variables.

```
scala> val x = 3
val x: Int = 3
```

Variables defined by `val` cannot be mutated, i.e. their values never change. Reassignment will incur an error. We call such variables immutable variables.

```
scala> x = 4
         ^
       error: reassignment to val
```

Sometimes, mutable variables, i.e. variables whose values can change, are useful. Scala provides mutable variables as well as immutable variables. You need to use `var` instead of `val` to define mutable variables. You may or may not write the type of a variable.

```
scala> var z = 5
var z: Int = 5

scala> z = 6
// mutated z

scala> z
val res8: Int = 6
```

To assign a new value to a mutable variable, the value must conform to the type of the variable. Otherwise, a type mismatch will happen.

```
scala> z = true
           ^
       error: type mismatch;
        found   : Boolean(true)
        required: Int
```

Functions

The syntax of a function definition is as follows:

```
def [name]([name]: [type], ...): [type] = [expression]
```

Many programming languages require `return` to specify the return value of a function. On the other hand, functions in Scala are like functions in mathematics: `return` is unnecessary. The return value of a function is the result of the body expression, which is the expression at the right side of `=` in the definition. The type annotation after each parameter specifies the type of the parameter. The type after the parentheses is the return type, which must be the same as the type of the return value.

```
scala> def add(x: Int, y: Int): Int = x + y
def add(x: Int, y: Int): Int

scala> add(3, 7)
val res9: Int = 10
```

The return types of functions can be omitted.

```
scala> def add(x: Int, y: Int) = x + y
def add(x: Int, y: Int): Int
```

However, parameter types cannot be omitted.

```
scala> def add(x, y) = x + y
                ^
       error: ':' expected but ',' found.
```

To write multiple expressions including variable and functions definitions in the body of a function, we put expressions separated by line breaks inside curly braces. Each line will be evaluated in the order, and the result of the last line will be the return value.

```
scala> def quadruple(x: Int): Int = {
     |   val y = x + x
     |   y + y
     | }
def quadruple(x: Int): Int
```

Inside `quadruple`, the variable `y` is defined and used for the computation of the return value.[15]

Multiple expressions inside curly braces are collectively treated as a single expression. We call such an expression a sequenced expression. Like any other expressions, a sequenced expression can occur anywhere an expression is needed. For example, it can be used to define a variable.

15: Vertical bars (|) at the beginning of lines are not part of code. They have been automatically inserted by REPL.

```
scala> val a = {
     |   val x = 1 + 1
     |   x + x
     | }
val a: Int = 4
```

There are many other things related to functions: recursion, first-class functions, closures, and anonymous functions. Chapter 3 will discuss recursion, and Chapter 4 will discuss the other topics.

Conditionals

A conditional expression performs computation depending on a certain condition, i.e. a boolean value. The syntax of a conditional expression is as follows:

```
if ([expression]) [expression] else [expression]
```

The first expression is the condition; the second expression is the true branch; the last expression is the false branch.

```
scala> if (true) 1 else 2
val res10: Int = 1
```

A conditional expression evaluates to a value. It is more similar to the ternary operator `? :` in C than a if statement. We do not need to make a variable mutable to initialize the variable with a conditional value.

```
scala> val x = if (true) 1 else 2
val x: Int = 1
```

On the other hand, people write code like below in languages like C.

```
int x;
if (true)
    x = 1;
else
    x = 2;
```

Conditional expressions in Scala are more expressive than the ternary operator in C because we can make complex computation a single expression with expression sequencing, which is impossible in C.

```
scala> if (true) {
     |   val x = 2
     |   x + x
     | } else {
     |   val x = 3
     |   x * x
     | }
val res11: Int = 4
```

Lists

A list is a collection of zero or more elements. A list maintains the order between its elements. Lists in Scala are immutable. Once a list is created, its elements never change. There are two ways to create a new list in Scala:

- `List([expression], ..., [expression])`
- `[expression] :: ... :: [expression] :: Nil`

The type of a list whose elements have type `T` is `List[T]`.

```
scala> List(1, 2, 3)
val res12: List[Int] = List(1, 2, 3)

scala> 1 :: 2 :: 3 :: Nil
val res13: List[Int] = List(1, 2, 3)
```

`List(...)` is more convenient than `::` for creating a new list from scratch. However, `::` is more flexible since it can prepend a new element in front of an existing list.[16]

16: It does not mutate the existing list to prepend the new element. It creates a new list with the element and the list.

```
scala> val l = List(1, 2, 3)
val l: List[Int] = List(1, 2, 3)

scala> 0 :: l
val res14: List[Int] = List(0, 1, 2, 3)
```

The `length` method computes the length of a list; parentheses are used to fetch the element at a specific index.[17]

17: The first index is `0`.

```
scala> l.length
val res15: Int = 3

scala> l(0)
val res16: Int = 1
```

In functional programming, accessing an arbitrary element of a list by an index is rare. We use pattern matching in most cases. The syntax of pattern matching for a list is as follows:[18]

18: The order between the cases can vary, which means that the `::` case may come first.

```
[expression] match {
  case Nil => [expression]
  case [name] :: [name] => [expression]
}
```

The expression in front of `match` is the target of pattern matching. If it is an empty list, it matches `case Nil`. The expression of the `Nil` case will be evaluated. Otherwise, it is a nonempty list and matches `case [name] :: [name]`. The first name denotes the head[19] of the list, and the second name denotes the tail[20] of the list. The expression of the `::` case will be evaluated.

19: the first element

20: a list consisting of all the elements except the head

The following function takes a list of integers as an argument and returns the head. The return value is zero when the list is empty.

```
scala> def headOrZero(l: List[Int]): Int = l match {
     |   case Nil => 0
     |   case h :: t => h
     | }
def headOrZero(l: List[Int]): Int
```

```
scala> headOrZero(List(1, 2, 3))
val res17: Int = 1

scala> headOrZero(List())
val res18: Int = 0
```

Chapter 3 will show use of pattern matching for lists in recursive functions, and Chapter 5 will discuss pattern matching in detail.

Tuples

A tuple contains two or more elements and maintains the order between its elements. We use parentheses to create a new tuple:

```
([expression], ..., [expression])
```

The type of a tuple whose elements have types from `T1` to `Tn` respectively is `(T1, ..., Tn)`. For example, the type of a tuple whose first element is `Int` and second element is `Boolean` is `(Int, Boolean)`.

```
scala> (1, true)
val res19: (Int, Boolean) = (1,true)
```

To fetch the `i`-th element of a tuple, we can use `._i`.[21]

21: The first index is 1.

```
scala> (1, true)._1
val res20: Int = 1
```

Tuples look similar to lists but have important differences from lists. First, a tuple's elements can have different types, while a list's elements cannot. For example, a tuple of the type `(Int, Boolean)` has one integer and one boolean, while a list of the type `List[Int]` can have only integers. We say that tuples are heterogenous, while lists are homogeneous. Second, a list allows accessing an arbitrary index of a list, while a tuple does not. For example, `l(f())` is possible where `l` is a list and `f` returns an integer, while there is no way to access the `f()`-th element of a tuple since the return value of `f` is unknown before execution.

We use lists and tuples for different purposes. Lists are appropriate when the number of elements can vary and an arbitrary index should be accessible. For instance, a list should be used to represent a collection of the heights of students in a certain class.

```
List(189, 167, 156, 170, 183)
```

It allows us to fetch the height of the `i`-th student.

On the other hand, tuples are appropriate when the number of elements are fixed and each index has a specific meaning. For instance, a tuple can represent the information of a single student, where the information consists of one's name, one's height, and whether one has payed the school expense or not.

```
("John Doe", 173, true)
```

We can use `._1` to find the name, `._2` to find the height, and `._3` to check whether one has payed.

We call a length-2 tuple a pair and a length-3 tuple a triple. Also, we can consider unit as a length-0 tuple.

Maps

A map is a collection of pairs, where each pair consists of a key and a value. Its provides the corresponding value when a key is given. Maps in Scala are immutable as well. Below is the syntax to create a new map:

```
Map([expression] -> [expression], ...)
```

The type of a map whose keys have type `T` and values have type `S` is `Map[T, S]`.

```
scala> val m = Map(1 -> "one", 2 -> "two", 3 -> "three")
val m: Map[Int,String] = Map(1 -> one, 2 -> two, 3 -> three)
```

To find the value corresponding to a certain key, we use parentheses.

```
scala> m(2)
val res21: String = two
```

Maps provide various methods.[22]

22: https://www.scala-lang.org/api/current/scala/collection/immutable/Map.html

Classes and Objects

An object is a value with fields and methods. Fields store values, and methods are operations related to the object. A class is a blueprint of objects. We can easily create multiple objects of the same structure by defining a single class. This book uses only "case" classes of Scala. Case classes are similar to classes but more convenient, e.g. automatic support for pretty printing and pattern matching.

The syntax of a class definition is as follows:

```
case class [name]([name]: [type], ...)
```

The first name is the name of a new class. The names inside the parentheses are the names of the fields of the class. A class definition must specify the types of its fields.

```
scala> case class Student(name: String, height: Int)
class Student
```

Creating new objects is similar to a function call.

```
scala> val s = Student("John Doe", 173)
val s: Student = Student(John Doe,173)
```

Fields can be accessed by `.[name]`.

```
scala> s.name
val res22: String = John Doe
```

Objects in Scala are immutable by default. If we add `var` to a field when defining a class, the field becomes mutable.

```
scala> case class Student(name: String, var height: Int)
class Student

scala> val s = Student("John Doe", 173)
val s: Student = Student(John Doe,173)

scala> s.height = 180
// mutated s.height

scala> s.height
val res23: Int = 180
```

2.4 Interpreter

An *interpreter* is a program that takes source code as input and runs the code. The Scala interpreter takes Scala source code as input. To use the interpreter, we need to save source code into a file. Make a file with the following code, and save it as `Hello.scala`.

```
println("Hello world!")
```

You can excute the interpreter by typing `scala` with the name of a file in your command line. Here, we need to say `scala Hello.scala`.

```
$ scala Hello.scala
Hello world!
```

You can write multiple lines in a single file. Fix `Hello.scala` like below.

```
val x = 2
println(x)
val y = x * x
println(y)
```

Then, excute the interpreter again.

```
$ scala Hello.scala
2
4
```

2.5 Compiler

A *compiler* is a program that takes source code as input and translates it into another language. Usually, the target language is a low-level language like machine code or bytecode of a particular virtual machine. The Scala compiler takes Scala source code as input and translates it into Java bytecode. Once code is compiled, we can run the generated bytecode with the JVM.

For compilation, we need to define the main method of a program. The main method is the entrypoint of every program running on the JVM. Make a file with the following code, and save it as `Hello.scala`.

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello world!")
  }
}
```

You can make the compiler compile the code by typing `scalac` with the name of the file in your command line.

```
$ scalac Hello.scala
```

After compilation, you will be able to find the `Hello.class` file in the same directory. The file contains Java bytecode.

You can run the bytecode with the JVM by the `scala` command. In this time, you should write only the class name.

```
$ scala Hello
Hello world!
```

You can change the behavior of a program by modifying the main method. Each time you modify, you need to re-compile the program to re-generate the bytecode.

Running bytecode is much more efficient than interpreting Scala source code. You can easily notice that `scala Hello` takes much less than `scala Hello.scala` even though their results are the same.

Scala has two sorts of errors: compile-time errors and run-time errors. Compile-time errors occur during compilation, i.e. while running `scalac`. If the compiler finds things that might go wrong at run time, it raises errors and aborts the compilation. For example, an expression adding an integer to a boolean results in a compile-time error because such an addition cannot succeed at run time.

```
true + 1
```

```
error: type mismatch;
 found   : Int(1)
 required: String
```

```
true + 1
     ^
```
Compile-time error

Unfortunately, some bad behaviors cannot be detected by the compiler. The compiler does not generate any errors for those behaviors. Such problems will incur run-time errors during execution, i.e. while running `scala`, and terminate the execution abnormally. Division by zero is one example of run time errors.

```
1 / 0
```

```
java.lang.ArithmeticException: / by zero
```
Run-time error

2.6 SBT

SBT is a build tool for Scala. Build tools help programmers work on large projects with many files and libraries by tracking dependencies between files and managing libraries. There are various build tools in the world, and SBT is the most popular one for Scala.

You can create a new Scala project by the `sbt new` command.

```
$ sbt new scala/scala-seed.g8
[info] welcome to sbt 1.4.7
[info] loading global plugins from ~/.sbt/1.0/plugins
[info] set current project to ~/ (in build file:~/)
[info] set current project to ~/ (in build file:~/)


A minimal Scala project.

name [Scala Seed Project]: hello

Template applied in ~/hello
```

After the creation, the directory structure is as follows:

```
hello
├── build.sbt
├── project
│   ├── Dependencies.scala
│   └── build.properties
└── src
    ├── main
    │   └── scala
    │       └── example
    │           └── Hello.scala
    └── test
        └── scala
            └── example
                └── HelloSpec.scala
```

The `build.sbt` file configures the project. It manages the version of Scala used for the project, third-party libraries used in the project, and many

other things. Source files are in the `src` directory. Files in `main` are main source files, while files in `test` are only for testing. You can add files into the `src/main/scala` directory and edit them to write code.

An SBT console can be started by the `sbt` command. The current working directory of your shell should be the base directory of the project.

```
$ sbt
[info] welcome to sbt 1.4.7
[info] loading global plugins from ~/.sbt/1.0/plugins
[info] loading project definition from ~/hello/project
[info] loading settings for project root from build.sbt ...
[info] set current project to hello (in build file:~/hello/)

[info] sbt server started at
local:///~/.sbt/1.0/server/d4cd702f998423203dfe/sock
[info] started sbt server
sbt:hello>
```

You can compile, run, and test the project by executing SBT commands in the console.

- `compile`: compile the project.
- `run`: run the project (re-compile if necessary).
- `test`: test the project (re-compile if necessary).
- `exit`: terminate the console.

```
sbt:hello> compile
[info] compiling 1 Scala source to ~/hello/target/scala-2.13

| => root / Compile / compileIncremental 0s
[success] Total time: 4 s
sbt:hello> test
[info] compiling 1 Scala source to ~/hello/target/scala-2.13
[info] HelloSpec:
[info] The Hello object
[info] - should say hello
[info] Run completed in 455 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0
[info] All tests passed.
[success] Total time: 2 s
sbt:hello> run
[info] running example.Hello
hello
[success] Total time: 0 s
sbt:hello> exit
[info] shutting down sbt server
```

To learn SBT more, refer to the SBT website.[23]

23: https://www.scala-sbt.org/learn.html

Immutability 3

Immutability means not changing. Immutable variables never change their values after initialization; immutable data structures never change their elements once created. The opposite of immutability is mutability. While imperative programming uses numerous mutable variables, data structures, and objects, functional programming leverages the power of immutable varibles, data structures, and objects. This chapter explains why immutability is important and valuable. Also, we will see how to program without mutation.

3.1 Advantages

The book Programming in Scala [OSV16] discusses four strengths of immutability:

> First, immutable objects are often easier to reason about than mutable ones, because they do not have complex state spaces that change over time. Second, you can pass immutable objects around quite freely, whereas you may need to make defensive copies of mutable objects before passing them to other code. Third, there is no way for two threads concurrently accessing an immutable to corrupt its state once it has been properly constructed, because no thread can change the state of an immutable. Fourth, immutable objects make safe hash table keys. If a mutable object is mutated after it is placed into a `HashSet`, for example, that object may not be found the next time you look into the `HashSet`.

We will focus on the first two advantages: easier reasoning and no need for defensive copies.

First, let us see why immutability makes things easy to reason about.

```
val x = 1
...
f(x)
```

At the first line of the code, `x` is `1`. Since `x` is immutable, there is no doubt that `x` is still `1` when `x` is passed as an argument for `f` at the last line of the code.

```
var x = 1
...
f(x)
```

On the other hand, if `x` is a mutable variable, one should read every line of code in the middle to find the value of `x` at the time when the function call happens.

When `x` is mutable, without tracking every modification of `x` throughout the code, the value of `x` at the last line is unknown. It hampers programmers from understanding the code and possibly leads to more bugs. The program with immutable `x` does not suffer from such problems. Remembering only one line of the code is enough to track the value of `x`.

Mutable data structures cause similar problems.

```
val x = List(1, 2)
...
f(x)
...
x
```

As `List` is immutable, `x` is a list always containing `1` and `2`.

```
import scala.collection.mutable.ListBuffer
val x = ListBuffer(1, 2)
...
f(x)
...
x
```

On the other hand, `ListBuffer` is a mutable data structure in the Scala standard library. It is possible to add an item to or remove an item from the list referred by `x`. Programmers cannot be certain about the content of `x` unless they read all the lines in between. Besides, a function `f` also is able to change the content of `x`. If one writes a program with a wrong assumption that `f` does not modify `x`, then the program might be buggy.

Mutable global variables make code much harder to understand than mutable local variables.

```
def f(x: Int) = g(x, y)
```

The return value of function `f` depends on the value of a global variable `y`. If `y` is mutable, `f` is not a pure function and expecting the behavior of `f` is nontrivial. `y` can be declared in any arbitrary file and all files are able to change the value of `y`. In the worst case, an external library defines `y` and source code modifying `y` is not available for reading.

The examples are small and seem artificial, but immutability greatly improves maintainability and readability of code in practice, especially for large projects.

Now, let us see why immutability free us from making defensive copies.

```
val x = ListBuffer(1, 2)
...
f(x)
...
x
```

Since `ListBuffer` creates mutable lists, there is no guarantee that the content of `x` is not changed by `f`. If it is necessary to prevent modification, copying `x` is essential.

```
val x = ListBuffer(1, 2)
val y = x.clone
...
f(y)
...
x
```

In cases that `x` has many elements and the code is executed multiple times, copying `x` increases the execution time significantly.

In the code, using the `clone` method is enough to copy the list because the list contains only integers. However, to pass lists containing mutable objects safely to functions, defining additional methods for deep copy is inevitable.

Immutability has several clear advantages. Immutability is an important concept in functional programming. Functional programs use immutable variables and data structures in most cases. If you write a large program whose logic is complex and correctness is important, you should adopt the functional paradigm. However, mind that immutability is not the silver bullet for every program. For example, implementing algorithms in a functional style is usually inefficient. It would be better to use mutable data structures like arrays, mutable variables, and loops to implement algorithms. They make programs much more efficient and faster. Choosing a programming proper paradigm for the purpose of a program is the key to write good code.

3.2 Recursion

Repeating the same computation multiple times is a common pattern in programming. Loops allow concise code expressing such cases. However, if everything is immutable, going back to the beginnings of loops does not change any states. Therefore, it is impossible to apply the same operation on different values for each iteration or to terminate the loops. As a consequence, loops are useless in functional programming. Functional programs use recursive functions instead of loops to rerun computation. A *recursive* function is a function that calls itself.[1] To do more computation, the function calls itself with proper arguments. Otherwise, it terminates the computation by returning some value.

1: In general, a definition that refers to itself is a recursive definition. There can be recursive variables, recursive types, and so on.

The below `factorial` function calculates the factorial of a given integer. For simplicity, we do not consider when the input is negative. The following implementation uses an imperative style:

```
def factorial(n: Int) = {
  var i = 1, res = 1
  while (i <= n) {
    res *= i
    i += 1
```

```
  }
  res
}
```

We can implement the same function in a functional style with recursion.

```
def factorial(n: Int): Int =
  if (n <= 0)
    1
  else
    n * factorial(n - 1)
```

Note that recursive functions always require explicit return types in Scala, unlike non-recursive functions, whose return types can be omitted.

The recursive version is preferred over the imperative version since its correctness is easily verified.

To check the correctness of the imperative `factorial` function, one should find a *loop invariant*, which is a proposition that is always true at the loop head. The loop invariant of this case is $((\texttt{i} - 1)! = \texttt{res}) \wedge (\texttt{i} \leq \texttt{n} + 1)$. By using this invariant, we can conclude that $\texttt{i} = \texttt{n} + 1$ and, therefore, $\texttt{res} = (\texttt{i} - 1)! = \texttt{n}!$ at the last line of the function, which implies that it correctly implements factorial. It is nontrival to find a proper loop invariant and show that the loop invariant holds at the beginning of each iteration.

On the other hand, recursive functions usually reveal their mathematical definitions more clearly than functions using loops. Consider the following mathematical definition of factorial:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

You can see that the implementation of the `factorial` function using recursion is identical to the mathematical definition of factorial. It is almost trivial to show that the recursive `factorial` function is correct. Recursion allows concise and intuitive descriptions of mathematical functions. In many cases, functions with recursion is much easier to be verified formally or informally than functions with loops.

Recursive functions are also good at treating recursive data structures like lists. A list is recursive since a nonempty list consists of a head element and a tail list, which means that a nonempty list has another list as its component. Writing some functions regarding lists helps understanding and practicing recursion.

The following function takes a list as an argument and returns a list whose elements are one larger than the elements of the given list.

```
def inc1(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h + 1 :: inc1(t)
}
```

When a given list is empty, the function returns the empty list. Otherwise, the return value is a list whose head is one larger than the head of the given list and tail has elements that are one larger than the elements of the tail of the given list.

Similarly, `square` takes a list of integers as an argument and returns a list whose elements are the squares of the elements of the given list.

```
def square(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h * h :: square(t)
}
```

The following function takes a list of integers as an argument and returns a list whose elements are odd integers.

```
def odd(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t =>
    if (h % 2 != 0)
      h :: odd(t)
    else
      odd(t)
}
```

For a nonempty list, the function checks whether the head is odd or not. If the head is odd, the resulting list contains the head, and its tail has only odd integers. Otherwise, the head is removed.

Similarly, `positive` takes a list of integers as an argument and returns a list whose elements are positive.

```
def positive(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t =>
    if (h > 0)
      h :: positive(t)
    else
      positive(t)
}
```

The following function calculates the sum of the elements of a given list.

```
def sum(l: List[Int]): Int = l match {
  case Nil => 0
  case h :: t => h + sum(t)
}
```

The sum of elements in the empty list is zero as there are no elements. When a list is nonempty, the sum of its elements can be calculated by adding the value of the head to the sum of its tail's elements.

Similarly, `product` calculates the product of the elements of a given list.

```
def product(l: List[Int]): Int = l match {
  case Nil => 1
  case h :: t => h * product(t)
}
```

Recursion has some disadvantages: overheads of function calls and stack overflow. Most modern CPUs have enough computing power to ignore function call overheads. However, loops are still more ideal than recursive functions in performance-critical programs. Stack overflow happens when a stack lacks space due to repetitive function calls. It is a critical problem since it causes immediate termination of execution without yielding meaningful output. Moreover, programs like web servers do not finish their execution, and their stacks will eventually overflow. To prevent stack overflow, many functional languages provide tail call optimization. The following section explains tail call optimization in detail.

3.3 Tail Call Optimization

If the last action of a function is a function call, then the call is a tail call. When a tail call happens, the callee does every computation, and thus the local variables of the caller have no need to remain after the call. The stack frame of the caller can be destroyed. Most functional languages exploit this fact to optimize tail calls. This optimization is called *tail call optimization*. At compile time, compilers check whether calls are tail calls. If a call is a tail call, the compilers generate code that eliminates the stack frame of the caller before the call. They do not optimize non-tail function calls because the local variables of the callers can be used after the callees return. If every function call in a program is a tail call, the stack never grows so that the program is safe from stack overflow.

```
def factorial(n: Int): Int =
  if (n <= 0)
    1
  else
    n * factorial(n - 1)
```

The previous `factorial` function multiplies `n` and the return value of the recursive `factorial(n - 1)` call. The multiplication is the last action. The recursive call is not a tail call. The stack frame of the caller must remain. The following process computes `factorial(3)`:

- `factorial(3)`
- `3 * factorial(2)`
- `3 * (2 * factorial(1))`
- `3 * (2 * (1 * factorial(0)))`
- `3 * (2 * (1 * 1))`
- `3 * (2 * 1)`
- `3 * 2`
- `6`

At most four stack frames coexist. For a large argument, a stack grows again and again and finally overflows.

```
factorial(10000)
```

```
java.lang.StackOverflowError
  at .factorial
```
Run-time error

To implement the function with a tail call, instead of multiplying `n` and `factorial(n - 1)`, the function has to pass both `n` and `n - 1` as arguments and make the callee multiply `n` and `(n - 1)!`. This strategy can be interpreted as passing an intermediate result.

- `factorial(3)`
- `factorial(2, intermediate result = 3)`
- `factorial(1, intermediate result = 3 * 2)`
- `factorial(1, intermediate result = 6)`
- `factorial(0, intermediate result = 6 * 1)`
- `factorial(0, intermediate result = 6)`
- `6`

There is no need to return to the caller. The below code shows the `factorial` function with a tail call. The function needs one more parameter that takes an intermediate result as an argument. `factorial(n, i)` computes `n!` × `i`.

```
def factorial(n: Int, inter: Int): Int =
  if (n <= 0)
    inter
  else
    factorial(n - 1, inter * n)
```

The function uses a tail call. More precisely, the function is tail-recursive. Its last action is calling itself. Unlike most functional languages, Scala cannot optimize general tail calls. Scala optimizes only tail-recursive calls. The Scala compiler generates Java bytecode, which is excuted by the JVM. The JVM does not allow bytecode to jump to the beginning of another function. In the JVM, functions can only either return or call functions. Therefore, the Scala compiler cannot generate optimized code by removing the stack frame of a caller. Instead, they transform tail-recursive calls into loops. The `factorial` function is compiled to the following bytecode:

```
public int factorial(int, int);
  Code:
     0: iload_1
     1: iconst_0
     2: if_icmpgt     9
     5: iload_2
     6: goto          20
     9: iload_1
    10: iconst_1
    11: isub
    12: iload_2
    13: iload_1
    14: imul
    15: istore_2
```

```
    16: istore_1
    17: goto          0
    20: ireturn
```

We can check that there is no function call at all.[2] The function just jumps to instructions inside the function. Due to the tail call optimization, the function never incurs stack overflow.

2: `invokevirtual` is a function call instruction.

Even with tail recursion, the result is still incorrect because of integer overflow.

```
assert(factorial(10000, 1) == 0)  // weird result
```

The `BigInt` type resolves integer overflow.

```
def factorial(n: BigInt, inter: BigInt): BigInt =
  if (n <= 0)
    inter
  else
    factorial(n - 1, inter * n)

assert(factorial(10000, 1) > 0)
```

The optimization of the Scala compiler not only prevents stack overflow but also removes the overheads of function calls. The downside is that mutually recursive functions using tail calls lie beyond the scope of the optimization. Mutual recursion is recursion involving two or more definitions. The following functions can cause stack overflow in Scala even though they use tail calls because they are not tail-recursive:

```
def even(n: Int): Boolean = if (n <= 0) true else odd(n - 1)
def odd(n: Int): Boolean = if (n == 1) true else even(n - 1)
```

In Scala, programmers can ask the compiler to check whether functions are tail-recursive with annotations. The annotations prevent programmers from making functions non-tail-recursive by mistakes.

```
import scala.annotation.tailrec
@tailrec def factorial(n: BigInt, inter: BigInt): BigInt =
  if (n <= 0)
    inter
  else
    factorial(n - 1, inter * n)
```

A non-tail-recursive function with the `tailrec` annotation results in a compile-time error.

```
@tailrec def factorial(n: Int): Int =
  if (n <= 0)
    1
  else
    n * factorial(n - 1)
```

```
      ^
error:
could not optimize @tailrec annotated method factorial:
it contains a recursive call not in tail position
```
Compile-time error

The annotation does not affect the behavior of the resulting bytecode. Regardless of the existence of the annotation, the compiler always optimizes tail-recursive functions. Still, using the annotations is desirable to prevent mistakes.

Calling the tail-recursive version of `factorial` needs the unnecessary second argument. The below code defines a new `factorial` function with one parameter and uses the tail-recursive one as a local function inside the function.

```
def factorial(n: BigInt): BigInt = {
  @tailrec def aux(n: BigInt, inter: BigInt): BigInt =
    if (n <= 0)
      inter
    else
      aux(n - 1, inter * n)
  aux(n, 1)
}
```

Some functions treating lists also can be rewritten in a tail-recursive way. Below is a tail-recursive version of `sum`.

```
def sum(l: List[Int]): Int = {
  @tailrec def aux(l: List[Int], inter: Int): Int = l match {
    case Nil => inter
    case h :: t => aux(t, inter + h)
  }
  aux(l, 0)
}
```

`aux(l, n)` calculates `n` plus the sum of `l`'s elements.

Similarly, `product` can be implemented in a tail-recursive way.

```
def product(l: List[Int]): Int = l match {
  @tailrec def aux(l: List[Int], inter: Int): Int = l match {
    case Nil => inter
    case h :: t => aux(t, inter * h)
  }
  aux(l, 1)
}
```

3.4 Exercises

1. Consider the following definition of `Student`:
   ```
   case class Student(name: String, height: Int)
   ```
   Implement a function `names`:

`def names(l: List[Student]): List[String] = ???`
that takes a list of students as an argument and returns a list containing the names of the students.

2. Consider the same definition of `Student`. Implement a function `tall`:
   `def tall(l: List[Student]): List[Student] = ???`
   that takes a list of students as an argument and returns a list of students whose heights are greater than 170.
3. Implement a function `length`:
   `def length(l: List[Int]): Int = ???`
   that takes a list of integers as an argument and returns the length of the list.
   Remark that there is a built-in method `l.length`, but try to implement yourself using recursive function.
4. Implement a function `append`:
   `def append(l: List[Int], n: Int): List[Int] = ???`
   that takes a list of integers and an integer as arguments and returns a list obtained by appending the integer at the end of the list. Then, compare the time complexity of appending a new element to that of prepending a new element by `::`, which is $O(1)$.
   Remark that there is a built-in method `l.appended(n)`, but try to implement yourself using recursive function.

Functions 4

This section focuses on use of functions in functional programming. In functional programming, functions are first-class. First-class functions allow programmers to abstract complex computation easily. This section explains what first-class functions are. In addition, anonymous functions and closures, which are related to first-class functions, will be introduced. To show the power of first-class functions, we will re-implement the functions in Chapter 3 (`inc1`, `square`, . . .) with first-class functions.

4.1 First-Class Functions

An entity in a programming language is *first-class* if it satisfies the following conditions:

- It can be an argument of a function call.
- It can be a return value of a function.
- A variable can refer to it.

Anything that is first-class can be used as a value. Functions are highly important and treated as values in functional languages. Functions that are first-class are called *first-class functions*.

Some people use the term higher-order functions. *Higher-order functions* are functions that are not first-order, where first-order functions neither take functions as arguments nor return functions. Therefore, higher-order functions can take functions as arguments and return functions. Strictly speaking, they are different from first-class functions because first-class functions are functions that can be passed as arguments or returned from functions. However, any languages that support first-class functions support higher-order functions and vice versa. The reason is obvious: to pass first-class functions as arguments, there should be higher-order functions, and to pass functions to higher-order functions, there should be first-class functions. Consequently, in most contexts, people do not distinguish first-class functions and higher-order functions, and you can consider first-class functions and higher-order functions as exchangeable terms.

Now, let us see how we can use first-class functions in Scala with some code examples.

```
def f(x: Int): Int = x
def g(h: Int => Int): Int = h(0)

assert(g(f) == 0)
```

The function `g` has one parameter `h`. The type of `h` is `Int => Int`. An argument passed to `g` is a function that receives one integer and returns an integer. In Scala, `=>` expresses the types of functions. Functions without

parameters have types of the form `() => [return type]`. `[parameter type] => [return type]` is the type of a function with a single parameter. Parentheses are required to express the types of functions with two or more parameters: `([parameter type], ... ) => [return type]`. The function `f` has one integer parameter and returns an integer, i.e. its type is `Int => Int`. Thus, it can be an argument for `g`. Evaluating `g(f)` equals evaluating `f(0)`, which results in `0`.

```
def f(y: Int): Int => Int = {
  def g(x: Int): Int = x
  g
}

assert(f(0)(0) == 0)
```

The function `f` returns the function `g`. Since the return type of `f` is `Int => Int`, its return value must be a function that takes an integer as an argument and returns an integer. `g` satisfies the condition. `f(0)` is the same as `g` and therefore is a function. `f(0)(0)` equals `g(0)`, which returns `0`.

```
val h0 = f(0)

assert(h0(0) == 0)
```

A variable can refer to `f(0)`. `h0` refers to the return value of `f(0)` and has type `Int => Int`. Calling variables referring to function values is possible. `h0(0)` is a valid expression and results in `0`.

```
val h1 = f
         ^
error: missing argument list for method f

Unapplied methods are only converted to functions
when a function type is expected.

You can make this conversion explicit
by writing ‘f _‘ or ‘f(_)‘ instead of ‘f‘.
```

Compile-time error

On the other hand, defining a variable referring to `f` results in a compile error. In Scala, a function defined by `def` is not a value per se. Since `f` is the name of a function but not a variable referring to a value, `h1` cannot refer to the value of `f`. As the above error message implies, underscores convert function names into function values.

```
val h1 = f _

assert(h1(0)(0) == 0)
```

Compiling the above code succeeds. The type of `h1` is `Int => (Int => Int)`. `Int => Int => Int` denotes the same type because `=>` is a right-associative type operator. `h1(0)(0)` is valid and yields `0`.

Actually, above expressions except `val h1 = f` use function names as values successfully. The Scala compiler transforms function names into function values when they occur where function types are expected. Therefore, enforcing the type of `h1` to be a function type corrects the code without the underscore. The following code works well:

```
val h1: Int => Int => Int = f
```

When programmers use function names as values, they usually place the names where function types are expected. In these cases, underscores and explicit type annotations are unnecessary. Code rarely becomes problematic and needs underscores or type annotations like the above to enforce the transformations.

How does the compiler create function values from function names? If the parameter type of function `f` is `Int`, the corresponding function value is `(x: Int) => f(x)`. The transformation is called *eta expansion*. `(x: Int) => f(x)` is a function value without a name and does the same thing as `f`. The following section covers functions without names.

4.2 Anonymous Functions

In functional programming, functions often appear only once as an argument or a return value. Naming functions used only once is unnecessary. The meaning of a function value is how it behave. While the parameters and body of a function decide its behavior, its name does not affect the behavior. Naturally, functional languages provide syntax to define functions without giving them names. Such functions are *anonymous functions*.

The syntax of an anonymous function in Scala is as follows:

```
([parameter name]: [parameter type], ...) => [expression]
```

Like functions declared by `def`, anonymous functions can be arguments, return values, or values referred by variables. Directly calling them is possible as well.

```
def g(h: Int => Int): Int = h(0)
g((x: Int) => x)

def f(): Int => Int = (x: Int) => x
f()(0)

val h = (x: Int) => x
h(0)

((x: Int) => x)(0)
```

The code does similar things to the previous code but uses anonymous functions.

Anonymous functions need explicit parameter types as named functions do. However, annotating every parameter type is verbose and inconvenient. The Scala compiler infers the types of parameters when anonymous functions occur where the compiler expects function types.

```
def g(h: Int => Int): Int = h(0)
g(x => x)
```

Since `g` has a parameter of type `Int => Int`, the compiler expects `x => x` to have the type `Int => Int`. It infers the type of `x` as `Int`.

```
val h: Int => Int = x => x
```

`h` has an explicit type annotation. `Int => Int` is the expected type of `x => x`. The compiler infers the type of `x` as `Int`.

```
val h = x => x
```

```
         ^
error: missing parameter type
```

Compile-time error

Unlike previous one, this code is problematic. Since there is no information to infer the type of `x` in `x => x`, the compiler generates an error.

Most cases using anonymous functions are arguments for function calls. Those functions do not require explicit parameter types. However, beginners might not be sure about whether parameter types can be omitted or not. Specifying parameter types is safe when you are not sure.

Scala provides one more syntax for anonymous functions: syntax using underscores. Underscores help programmers to create anonymous functions in a concise and intuitive way. Underscores can be used only when certain conditions are satisfied. Every parameter must occur exactly once in the body of a function in the order. Moreover, the function must not be an identity function like `(x: Int) => x`. In functions satisfying the conditions, underscores can replace parameters in the body. Otherwise, it is impossible to use underscores to create anonymous functions.

```
def g0(h: Int => Int): Int = h(0)
g0(_ + 1)

def g1(h: (Int, Int) => Int): Int = h(0, 0)
g1(_ + _)
```

The compiler transforms `_ + 1` into `x => x + 1`. Similarly, `_ + _` becomes `(x, y) => x + y`. The compiler automatically creates parameters as many as underscores and substitutes the underscores with the parameters. The mechanism clearly shows why the aforementioned conditions exist.

```
val h0 = (_: Int) + 1
val h1 = (_: Int) + (_: Int)
```

Underscores can have explicit types. Programmers should supply parameter types to succeed compiling when the compiler cannot infer them.

The transformation happens for the shortest expression containing underscores. Expressing anonymous functions with underscores is sometimes tricky.

```
def f(x: Int): Int = x
def g1(h: Int => Int): Int = h(0)
g1(f(_))
```

As intended, `f(_)` becomes `x => f(x)`, whose type is `Int => Int`.[1]

1: Actually, there is no need to write `g(f(_))` because it is equal to `g(f)`.

```
g1(f(_ + 1))
```

```
     ^
error: missing parameter type for expanded function
((<x$1: error>) => x$1.$plus(1))
```
Compile-time error

On the other hand, `f(_ + 1)` becomes `f(x => x + 1)` but not `x => f(x + 1)`. As `f` takes an integer, not a function, it results in a compile-time error.

```
def g2(h: (Int, Int) => Int): Int = h(0, 0)
g2(f(_) + _)
```

`f(_) + _` becomes `(x, y) => f(x) + y`, whose type is `(Int, Int) => Int`, and the compilation succeeds.

```
g2(f(_ + 1) + _)
```

```
             ^
error: missing parameter type for expanded function
((<x$2: error>) => f(((<x$1: error>) => x$1.$plus(1)))
  .<$plus: error>(x$2))
```
Compile-time error

`f(_ + 1) + _` becomes `y => f(x => x + 1) + y` but not `(x, y) => f(x + 1) + y`.

Like type inference of parameter types, novices may not be sure about how anonymous functions with underscores are transformed. It is recommended to use normal anonymous functions without underscores for those who are not confident about the mechanism of underscores.

4.3 Closures

Closures are function values that capture environments, which store the values of existing variables, when they are defined. The bodies of closures may have variables not defined in themselves, and the environments store the values of those variables.

```
def makeAdder(x: Int): Int => Int = {
  def adder(y: Int): Int = x + y
  adder
}
```

The definition of `adder`, `def adder(y: Int): Int = x + y`, does not define but uses `x`. However, the code is correct.

```
val add1 = makeAdder(1)
assert(add1(2), 3)

val add2 = makeAdder(2)
assert(add2(2), 4)
```

`add1` and `add2` refer to the same `adder` function, but the former returns an integer one larger than an argument, and the latter returns an integer two larger than an argument. The results of `add1(2)` and `add2(2)` are `3` and `4`, respectively. It is possible because the closures capture the environments when they are created. `add1` refers to a thing like `(adder, x = 1)` instead of just `adder`. Similarly, `add2` is actually `(adder, x = 2)`. Since the environment of `add1` stores the fact that `x` is `1`, `add1(2)` results in `3`. Under the environment of `add2`, `x` denotes `2`, and thus `x + y` is `4` when `y` is `2`.

4.4 First-Class Functions and Lists

This section shows how first-class functions allow generalization of the functions defined in Chapter 3.

```
def inc1(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h + 1 :: inc1(t)
}

def square(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h * h :: square(t)
}
```

`inc1` increases every element of a given list by one, and `square` squares every element. The two functions are remarkably similar. To make the similarity clearer, let us rename the functions to `g`.

```
def g(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h + 1 :: g(t)
}

def g(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h * h :: g(t)
}
```

The only difference is the left operand of `::` in the third line: `h + 1` versus `h * h`. By adding one parameter, the functions become entirely identical.

```
def g(l: List[Int], f: Int => Int): List[Int] = l match {
  case Nil => Nil
  case h :: t => f(h) :: g(t, f)
}
g(l, h => h + 1)

def g(l: List[Int], f: Int => Int): List[Int] = l match {
  case Nil => Nil
  case h :: t => f(h) :: g(t, f)
}
g(l, h => h * h)
```

This function is called `map`. The returned list has elements obtained by **map**ping a given function to the elements of a given list.

```
def map(l: List[Int], f: Int => Int): List[Int] = l match {
  case Nil => Nil
  case h :: t => f(h) :: map(t, f)
}
```

`inc1` and `square` can be redefined with `map`.

```
def inc1(l: List[Int]): List[Int] = map(l, h => h + 1)
def square(l: List[Int]): List[Int] = map(l, h => h * h)
```

An underscore makes `inc1` conciser.

```
def inc1(l: List[Int]): List[Int] = map(l, _ + 1)
```

Let us compare `odd` and `positive`.

```
def odd(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t =>
    if (h % 2 != 0)
      h :: odd(t)
    else
      odd(t)
}

def positive(l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t =>
    if (h > 0)
      h :: positive(t)
    else
      positive(t)
}
```

They look similar. They can become identical by renaming and adding parameters.

```
def filter(l: List[Int], f: Int => Boolean): List[Int] = l match {
  case Nil => Nil
  case h :: t =>
    if (f(h))
      h :: filter(t, f)
    else
      filter(t, f)
}
```

The function is called `filter` because it **filter**s unwanted elements out from a given list.

`odd` and `positive` can be redefined with `filter`.

```
def odd(l: List[Int]): List[Int] =
  filter(l, h => h % 2 != 0)
def positive(l: List[Int]): List[Int] =
  filter(l, h => h > 0)
```

Underscores make the functions conciser.

```
def odd(l: List[Int]): List[Int] = filter(l, _ % 2 != 0)
def positive(l: List[Int]): List[Int] = filter(l, _ > 0)
```

Let us compare `sum` and `product` without tail recursion.

```
def sum(l: List[Int]): Int = l match {
  case Nil => 0
  case h :: t => h + sum(t)
}

def product(l: List[Int]): Int = l match {
  case Nil => 1
  case h :: t => h * product(t)
}
```

After renaming the names to `g`, two differences exist: `0` versus `1` and `h + g(t)` versus `h * g(t)`. By adding two parameters, an initial value and a function taking `h` and `g(t)` as arguments, the functions become identical.

```
def foldRight(
  l: List[Int],
  n: Int,
  f: (Int, Int) => Int
): Int = l match {
  case Nil => n
  case h :: t => f(h, foldRight(t, n, f))
}
```

This function is called `foldRight` since it appends an initial value at the right side of a list and **fold**s the list from the **right** side with a given function.

`sum` and `product` can be redefined with `foldRight`.

```
def sum(l: List[Int]): Int =
  foldRight(l, 0, (h, gt) => h + gt)
def product(l: List[Int]): Int =
  foldRight(l, 1, (h, gt) => h * gt)
```

They may use underscores for conciseness.

```
def sum(l: List[Int]): Int = foldRight(l, 0, _ + _)
def product(l: List[Int]): Int = foldRight(l, 1, _ * _)
```

The following equations give an intuitive interpretation of `foldRight`:

```
  foldRight(List(a, b, .., y, z), n, f)
= f(a, f(b, .. f(y, f(z, n)) .. ))

  foldRight(List(1, 2, 3), 0, add)
= add(1, add(2, add(3, 0)))

  foldRight(List(1, 2, 3), 1, mul)
= mul(1, mul(2, mul(3, 1)))
```

Let us compare tail-recursive `sum` and `product`.

```
def sum(l: List[Int]): Int = {
  def aux(l: List[Int], inter: Int): Int = l match {
    case Nil => inter
    case h :: t => aux(t, inter + h)
  }
  aux(l, 0)
}

def product(l: List[Int]): Int = l match {
  def aux(l: List[Int], inter: Int): Int = l match {
    case Nil => inter
    case h :: t => aux(t, inter * h)
  }
  aux(l, 1)
}
```

After renaming, there are two differences: `inter + h` versus `inter * h` and `0` versus `1`. Similarly, adding two parameters makes the functions identical.

```
def foldLeft(
  l: List[Int],
  n: Int,
```

```
  f: (Int, Int) => Int
): Int = {
  def aux(l: List[Int], inter: Int): Int = l match {
    case Nil => inter
    case h :: t => aux(t, f(inter, h))
  }
  aux(l, n)
}
```

This function is called `foldLeft`. Its semantics is different from `foldRight`. While `foldRight` appends an initial value at the right side and folds a list from the right side, `foldLeft` prepends an initial value at the left side and **fold**s a list from the **left** side. The following equations give an intuitive interpretation:

```
  foldLeft(List(a, b, .., y, z), n, f)
= f(f( .. f(f(n, a), b), .. , y), z)

  foldLeft(List(1, 2, 3), 0, add)
= add(add(add(0, 1), 2), 3)

  foldLeft(List(1, 2, 3), 1, mul)
= mul(mul(mul(1, 1), 2), 3)
```

The order traversing a list does not affect the results of `sum` and `product`. Both `foldRight` and `foldLeft` can express the functions.

```
def sum(l: List[Int]): Int = foldLeft(l, 0, _ + _)
def product(l: List[Int]): Int = foldLeft(l, 1, _ * _)
```

On the other hand, the order is important for some functions. Consider a function that takes a list of digits as arguments and returns the decimal number obtained by concatenating the digits. `foldLeft` is the easiest way to implement this function.

```
def digitToDecimal(l: List[Int]) =
  foldLeft(l, 0, _ * 10 + _)

  foldLeft(List(1, 2, 3), 0, f)
= f(f(f(0, 1), 2), 3)
= ((0 * 10 + 1) * 10 + 2) * 10 + 3
= (1 * 10 + 2) * 10 + 3
= 12 * 10 + 3
= 123
```

Using `foldRight` with the same arguments will yield completely different result.

```
def digitToDecimal(l: List[Int]) =
  foldRight(l, 0, _ * 10 + _)

  foldRight(List(1, 2, 3), 0, f)
```

```
= f(1, f(2, f(3, 0)))
= 1 * 10 + (2 * 10 + (3 * 10 + 0))
= 1 * 10 + (2 * 10 + 30)
= 1 * 10 + 50
= 60
```

`map`, `filter`, `foldRight`, and `foldLeft` are powerful functions. The four functions offer concise implementation for many procedures dealing with lists. Since they are so useful, the Scala standard library provides `map`, `filter`, `foldRight`, and `foldLeft` as the methods of the `List` class. You do not need to implement `map`, `filter`, `foldRight`, and `foldLeft` by yourself.

`map(l, f)` can be rewritten to `l.map(f)` by using the `map` method instead.

```
def inc1(l: List[Int]): List[Int] = l.map(_ + 1)
def square(l: List[Int]): List[Int] = l.map(h => h * h)
```

`filter(l, f)` can be rewritten to `l.filter(f)` by using the `filter` method instead.

```
def odd(l: List[Int]): List[Int] = l.filter(_ % 2 != 0)
def positive(l: List[Int]): List[Int] = l.filter(_ > 0)
```

`foldRight(l, n, f)` can be rewritten to `l.foldRight(n)(f)` by using the `foldRight` method instead.

```
def sum(l: List[Int]): Int = l.foldRight(0)(_ + _)
def product(l: List[Int]): Int = l.foldRight(1)(_ * _)
```

`foldLeft(l, n, f)` can be rewritten to `l.foldLeft(n)(f)` by using the `foldLeft` method instead.

```
def sum(l: List[Int]): Int = l.foldLeft(0)(_ + _)
def product(l: List[Int]): Int = l.foldLeft(1)(_ * _)
def digitToDecimal(l: List[Int]) = l.foldLeft(0)(_ * 10 + _)
```

The methods in the standard library are polymorphic, i.e. they can take arguments of various types. For example, our `map` function takes only a list of integers. To use `map` for a list of students, we need to define a new version of `map`. However, the `map` method in the standard library can take lists of any types as arguments.

```
case class Student(name: String, height: Int)

def heights(l: List[Student]): List[Int] = l.map(_.height)
```

The standard library provides many other useful methods for lists.[2]

2: https://www.scala-lang.org/api/current/scala/collection/immutable/List.html

4.5 For Loops

Scala has for loops. In Scala, a for loop is an expression, which evalautes to a value. For expressions are highly expressive. Unlike `while`, which work with mutable variables or objects, `for` of Scala helps programmers to write code in a functional and readable way.

The syntax of a for expression is as follows:

```
for ([name] <- [expression])
  yield [expression]
```

The first expression should result in a collection. Its result is a collection containing the result of evaluating the second expression at each iteration. Therefore, for expressions can replace use of the `map` method.

```
val l = for (n <- List(0, 1, 2)) yield n * n
assert(l == List(0, 1, 4))
```

For expressions can appear at any places expecting expressions.

```
def square(l: List[Int]): List[Int] =
  for (n <- l)
    yield n * n
```

In Scala, `for` is just syntactic sugar. Instead of giving specific semantics to `for`, syntactic rules transform code using `for` into the code using methods of collections and anonymous functions. The above function becomes the following function, which uses `map`, by the transformation:

```
def square(l: List[Int]): List[Int] =
  l.map(n => n * n)
```

For this reason, for expressions are powerful. Any user-defined types can appear in for expressions if the types define `map`.

For expressions can replace use of the `filter` method as well.

```
def positive(l: List[Int]): List[Int] =
  for (n <- l if n > 0)
    yield n
```

Elements not satisfying a given condition will be omitted during iteration.

In addition, combination of `map` and `filter` can be expressed with a for loop concisely. Consider a function that takes a list of students and returns a list of the names of students whose height is greater than 170. The function can be implemented with `map` and `filter` like below.

```
def tall(l: List[Student]): List[String] =
  l.filter(_.height > 170).map(_.name)
```

We can use a for expression instead.

```
def tall(l: List[Student]): List[String] =
  for (s <- l if s.height > 170)
    yield s.name
```

4.6 Exercises

1. Implement a function `incBy`:
   `def incBy(l: List[Int], n: Int): List[Int] = ???`
   that takes a list of integers and an integer as arguments and increases every element of the list by the given integer. Use the `map` method.
2. Implement a function `gt`:
   `def gt(l: List[Int], n: Int): List[Int] = ???`
   that takes a list of integers and an integer as arguments and filters elements less than or equal to the given integer out from the list. Use the `filter` method.
3. Implement a function `append`:
   `def append(l: List[Int], n: Int]): List[Int] = ???`
   that takes a list of integers and an integer as arguments and returns a list obtained by appending the integer at the end of the list. Use the `foldRight` method.
4. Implement a function `reverse`:
   `def reverse(l: List[Int]): List[Int] = ???`
   that takes a list of integers and returns a list obtained by reversing the order between the elements. Use the `foldLeft` method.

Pattern Matching 5

This section explains pattern matching of Scala. Pattern matching is one of the key features of functional programming. It helps programmers handle complex, but structured data. We have already used a simple form of pattern matching for lists. This section discusses the benefits of pattern matching and various patterns available in Scala. In addition, it will introduce the option type, which is widely-used in functional programming.

5.1 Algebraic Data Types

It is common to include values of various shapes in a single type.

A natural number is

- zero or
- the successor of a natural number.

A list is

- the empty list or
- a pair of an element and a list.

A binary tree is

- the empty tree or
- a tree containing a root element and two child trees.

An arithmetic expression is

- a number,
- the sum of two arithmetic expressions, or
- the difference of two arithmetic expressions.

An expression of the lambda calculus is

- a variable,
- a function, which is a pair of a variable and an expression, or
- a function application, which is a pair of two expressions.

As the examples show, in computer science, a single type often includes values of various shapes. *Algebraic data types* (ADT) express such types. An ADT is the sum type of product types. That is why it is called "algebraic." A *product type* is a type whose every element is an enumeration of values of types in the same specific order. Tuple types are typical product types. A *sum type*, whose another name is a *tagged union type*, has values of multiple types as its values. Unlike a union type, each component of a sum type has a tag to be distinguished from other components. In an ADT, one form of values that can be distinguished from the other forms is called a *variant*.

For example, an arithmetic expression, which has three variants, is

- a number,
- the sum of two arithmetic expressions, or
- the difference of two arithmetic expressions.

Therefore, we can define the `AE` type, which is the type of an arithmetic expression, as the sum type of

- the `Int` type tagged with `Num`,
- the `AE * AE` type tagged with `Add`, and
- the `AE * AE` type tagged with `Sub`,

where `AE * AE` denotes the product type of `AE` and `AE`.

ADTs are common in functional languages. Most functional languages allow users to define new ADTs. The following OCaml code defines arithmetic expressions:

```
type ae =
| Num of int
| Add of ae * ae
| Sub of ae * ae
```

Scala does not provide a direct way to define ADTs. Instead, Scala provides traits and classes, which are more general mechanisms to define new types, and programmers can express ADTs with traits and classes.

A new type can be defined as a trait. The syntax of a trait definition is as follows:

```
trait [name]
```

It defines a type whose name is `[name]`. The following code defines the `AE` type, which is the type of arithmetic expressions:

```
sealed trait AE
```

The `sealed` modifier prevents `AE` being extended outside the file that defines `AE`. We will get back to this point when we discuss the exhaustivity checking of pattern matching.

Once a type is defined as a trait, the type can be used just like any other types. For example, we can define an identity function for arithmetic expressions.

```
def identity(ae: AE): AE = ae
```

However, traits do not have ability to construct new values. It means that there is no way to create a value of the type `AE` yet. We need to define the variants of `AE` as case classes by extending `AE`.

```
case class Num(value: Int) extends AE
case class Add(left: AE, right: AE) extends AE
case class Sub(left: AE, right: AE) extends AE
```

As you have seen in Chapter 2, we can easily create values of case classes.

```
val n = Num(10)
val m = Num(5)
val e1 = Add(n, m)
val e2 = Sub(e1, Num(3))
```

Like traits, case classes also define types. The name of each class is the name of the defined type. Every instance of a class belongs to the type corresponding to the class.

```
val n: Num = Num(10)
val m: Num = Num(5)
val e1: Add = Add(n, m)
val e2: Sub = Sub(e1, Num(3))
```

In addition, becuase of the `extends` keyword, `Num`, `Add`, and `Sub` are subtypes of `AE`. It means that any value of the types `Num`, `Add`, or `Sub` is also a value of the type `AE`.

```
val n: AE = Num(10)
val m: AE = Num(5)
val e1: AE = Add(n, m)
val e2: AE = Sub(e1, Num(3))
```

We know that we can access the fields of objects with their names.

```
val n: Num = Num(10)
n.value
```

However, we cannot access the fields of an object when its type becomes `AE`.

```
val m: AE = Num(10)
m.value
```

```
  ^
error: value value is not a member of AE
```

Compile-time error

The reason is that `m` can be `Add` or `Sub`, which do not have the field `value`, as `AE` consists of not only `Num` but also `Add` and `Sub`. The compiler thinks that `m` may not have the field `value` and considers `m.value` as an unsafe expression, which should be rejected.

The best way to use ADTs is pattern matching. The following function evaluates a given arithmetic expression and returns the number denoted by the arithmetic expression.

```
def eval(e: AE): Int = e match {
  case Num(n) => n
  case Add(l, r) => eval(l) + eval(r)
```

```
  case Sub(l, r) => eval(l) - eval(r)
}

assert(eval(Sub(Add(Num(3), Num(7)), Num(5))) == 5)
```

When `e` is `Num(n)`, `eval` simply returns `n`. When `e` is `Add(l, r)`, `eval` respectively evaluates `l` and `r`, which are arithmetic expressions. `eval` returns the sum of the results of `l` and `r`. The `Sub(l, r)` case is similar. `eval` returns the difference of the results of `l` and `r`.

The list type is another good example of an ADT. The Scala standard library defines lists similar to the following code:

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class ::[A](head: A, tail: List[A]) extends List[A]
```

This code omits some details but clearly shows the high-level idea to define lists.[1] A list is either the empty list or a nonempty list, which is a pair of its head and tail. `Nil` is defined as a case object, not a case class, since there is only one empty list. Every empty list is identical. We use a case object to express this idea. `Nil` is created only once during entire execution, and every `Nil` is identitcal. The name `::` looks a bit weird, but it is for readability of pattern matching. Scala allows writing class names as infix operators in patterns. It means that both `case ::(h, t) =>` and `case h :: t =>` are allowed. Due to the class name `::`, we can write `case h :: t =>` in pattern matching.

1: We will not see what `[+A]` and `Nothing` are here. You can understand the overall ADT structure without knowing those concepts.

5.2 Advantages

Conciseness

Without pattern matching, handling ADTs becomes a complicated job. We need to use dynamic type tests to distinguish variants and type casting to access the fields of values.

Below is `eval` without pattern matching:

```
def eval(e: AE): Int =
  if (e.isInstanceOf[Num])
    e.asInstanceOf[Num].value
  else if (e.isInstanceOf[Add]) {
    val e0 = e.asInstanceOf[Add]
    eval(e0.left) + eval(e0.right)
  } else {
    val e0 = e.asInstanceOf[Sub]
    eval(e0.left) - eval(e0.right)
  }
```

`e.isInstanceOf[Num]` tests whether `e` is an instance of class `Num`. If it is true, `eval` should return the value of the field `value` of `e`. However, `value` cannot be directly accessed since `e`'s type is `AE`. Because

`e.isInstanceOf[Num]` is true, we are sure that `e`'s actual type is `Num`. In this case, we can inform this knowledge to the compiler with type casting. `e.asInstanceOf[Num]` does not change the value denoted by `e` but lets the compiler know that the programmer guarantees the type of `e` to be `Num`. Therefore, the compiler considers `e.asInstanceOf[Num]` to belong `Num` and allows accessing the field `value`. These type tests and casting processes should be done for the other variants, `Add` and `Sub`, too.

The code is long and complicated despite its simple functionality. Dynamic type tests and explicit type casting occupy most of the code, while real computation requires short code. Besides, such code is error-prone. For example, programmers may write code like below by mistake:

```
else if (e.isInstanceOf[Add]) {
  val e0 = e.asInstanceOf[Sub]
  eval(e0.left) + eval(e0.right)
}
```

While the condition checks whether `e` is an instance of `Add`, `e` becomes casted to `Sub`. Such code will trigger an error at run time and terminate the execution abnormally. It is easy to check whether `eval` is correct because it is short. However, complex types and computation will increase the possibility of mistakes.

Pattern matching gives us a much better solution. Pattern matching hides type tests and casting and makes code concise. At the same time, pattern matching removes the possibility of mistakes.

Exhaustivity Checking

Pattern matching checks the exhaustivity of patterns. At run time, a match error occurs when a given value matches none of given patterns.

```
def eval(e: AE): Int = e match {
  case Add(l, r) => eval(l) + eval(r)
  case Sub(l, r) => eval(l) - eval(r)
}
```

The function lacks the `Num` pattern.

```
eval(Num(3))
```

```
scala.MatchError: Num(3) (of class Num)
```

Run-time error

An argument of type `Num` results in a match error at run time.

Fortunately, we can easily avoid such mistakes. The Scala compiler checks whether patterns are exhaustive and warns if they are not.

```
def eval(e: AE): Int = e match {
  case Add(l, r) => eval(l) + eval(r)
  case Sub(l, r) => eval(l) - eval(r)
}
```

```
                        ^
warning: match may not be exhaustive.
It would fail on the following input: Num(_)
```
Compile-time warning

The compiler warns programmers about that the patterns are not exhaustive. Moreover, it precisely informs which patterns are missing to help debugging. Exhaustivity checking is beneficial for complex programs. It helps programmers make error-free programs and thus is a crucial strength of pattern matching.

For exhaustivity checking, the `sealed` modifier of traits is necessary. Wihtout `sealed`, a trait can be extended outside the file that defines it. The unit of compilation is a single file, so it is impossible to find all the variants by scanning a single file when a trait is not sealed. Exhaustivity checking during pattern matching will be impossible. The `sealed` keyword resolves the problem. Since sealed traits cannot be extended further, it is enough to check only the file that defines a sealed trait to find every variant of the trait. It is why we use sealed traits to define ADTs.

Reachability Checking

Like `switch-case`, pattern matching compares a value to patterns sequentially from top to bottom and selects the first matching pattern. If there are duplicated patterns, the latter will not be reachable. The compiler warns programmers when they find unreachable patterns to prevent such code.

```
def eval(e: AE): Int = e match {
  case Num(n) => 0
  case Add(l, r) => eval(l) + eval(r)
  case Num(n) => n
  case Sub(l, r) => eval(l) - eval(r)
}
```

```
  case Num(n) => n
                 ^
warning: unreachable code
```
Compile-time warning

When code is simple and short, it is easy to check whether there are unreachable patterns. However, in complex code, programmers often insert unreachable patterns by mistake and make critical bugs. Reachability checking of the compiler is an important feature to prevent such bugs.

5.3 Patterns in Scala

Constant and Wildcard Patterns

`switch-case` statements divide a given value into multiple cases in imperative languages. Pattern matching is a general form of `switch-case`.

The following code is an example using a `switch-case` statement in Java:

```
String grade(int score) {
  switch (score / 10) {
    case 10: return "A";
    case 9: return "A";
    case 8: return "B";
    case 7: return "C";
    case 6: return "D";
    default: return "F";
  }
}
```

Constant and wildcard patterns exist in Scala. Constant patterns are literals like integers and strings. A constant pattern matches a given value if a value denoted by the pattern equals the given value. The underscore denotes the wildcard pattern, which matches every value, and is equivalent to `default` of `switch-case`. The following function rewrites the previous function with pattern matching:

```
def grade(score: Int): String =
  (score / 10) match {
    case 10 => "A"
    case 9 => "A"
    case 8 => "B"
    case 7 => "C"
    case 6 => "D"
    case _ => "F"
  }

assert(grade(85) == "B")
```

Or Patterns

`switch-case` statements use the fall-through semantics; if `break` does not exist, after executing code corresponding to a case, the flow of the execution moves to code corresponding to the next case. Since the results of cases `10` and `9` are identical, the function can use fall-through.

```
String grade(int score) {
  switch (score / 10) {
    case 10:
    case 9: return "A";
    case 8: return "B";
    case 7: return "C";
    case 6: return "D";
    default: return "F";
  }
}
```

In contrast, pattern matching disallows fall-through. Instead, or patterns give a way to write the same expression only once for multiple patterns.

The syntax of an or pattern is `[pattern] | [pattern] | ...`, which is a sequence of multiple patterns with vertical bars in between. `A | B` matches values that match `A` or `B`.

```
def grade(score: Int): String =
  (score / 10) match {
    case 10 | 9 => "A"
    case 8 => "B"
    case 7 => "C"
    case 6 => "D"
    case _ => "F"
  }

assert(grade(100) == "A")
```

Nested Patterns

Nested patterns are patterns containing patterns. The `optimizeAdd` function optimizes a given arithmetic expression by eliminating additions of zeros.

```
def optimizeAdd(e: AE): AE = e match {
  case Num(_) => e
  case Add(Num(0), r) => optimizeAdd(r)
  case Add(l, Num(0)) => optimizeAdd(l)
  case Add(l, r) => Add(optimizeAdd(l), optimizeAdd(r))
  case Sub(l, r) => Sub(optimizeAdd(l), optimizeAdd(r))
}
```

Nested patterns help programmers treat values with complex structures easily.

Patterns with Binders

Assume that we have one more variant of `AE`:

```
case class Abs(e: AE) extends AE
```

It denotes the absolute value of an operand. Optimizing an arithmetic expression decorated by two consecutive `Abs` operators results in the arithmetic expression with only one `Abs` operator.

```
def optimizeAbs(e: AE): AE = e match {
  case Num(_) => e
  case Add(l, r) => Add(optimizeAbs(l), optimizeAbs(r))
  case Sub(l, r) => Sub(optimizeAbs(l), optimizeAbs(r))
  case Abs(Abs(e0)) => optimizeAbs(Abs(e0))
  case Abs(e0) => Abs(optimizeAbs(e0))
}
```

A flaw of the implementation is that a value matching `Abs(e0)` cannot be an argument of `optimizeAbs` directly, and constructing a new `Abs` instance containing a value matching `e0` is essential. The `@` symbol makes code efficient by binding a value matching to a pattern to a variable. Pattern `[variable] @ [pattern]` makes the variable refer to a value matching the pattern.

```
def optimizeAbs(e: AE): AE = e match {
  case Num(_) => e
  case Add(l, r) => Add(optimizeAbs(l), optimizeAbs(r))
  case Sub(l, r) => Sub(optimizeAbs(l), optimizeAbs(r))
  case Abs(e0 @ Abs(_)) => optimizeAbs(e0)
  case Abs(e0) => Abs(optimizeAbs(e0))
}
```

Type Patterns

In `optimizeAbs`, the first `Num(_)` pattern does no more than checking whether a value belongs to type `Num`. A type pattern helps to rewrite the function. Type patterns are in the form of `[name]: [type]`. If a value belongs to the type, it matches the pattern, and the variable refers to the value. The wildcard pattern can substitute the name if the variable is unnecessary.

```
def optimizeAbs(e: AE): AE = e match {
  case _: Num => e
  case Add(l, r) => Add(optimizeAbs(l), optimizeAbs(r))
  case Sub(l, r) => Sub(optimizeAbs(l), optimizeAbs(r))
  case Abs(e0 @ Abs(_)) => optimizeAbs(e0)
  case Abs(e0) => Abs(optimizeAbs(e0))
}
```

Type patterns are useful for dynamic type checking. The following function takes any value as an argument and check whether it is a string or not.[2]

2: Every type is a subtype of `Any`, i.e. every value belongs to `Any`.

```
def show(x: Any): String = x match {
  case s: String => s + " is a string"
  case _ => "not a string"
}

assert(show("1") == "1 is a string")
assert(show(1) == "not a string")
```

Note that type patterns cannot check type arguments of polymorphic types. Using type patterns against polymorphic types is dangerous.

```
def show(x: Any): String = x match {
  case _: List[String] => "a list of strings"
  case _ => "not a list of strings"
}
```

```
          ^
warning: non-variable type argument String in type pattern
List[String] is unchecked since it is eliminated by erasure
```
Compile-time warning

```
val l: List[Int] = List(1, 2, 3)
assert(show(l) == "a list of strings")  // weird result
```

Although the type of the argument is `List[Int]`, it matches the first pattern. As the warnings imply, the JVM uses type erasure semantics, and type arguments are unavailable at run time.

Tuple Patterns

The syntax of a tuple pattern is `([pattern], ..., [pattern])`. It matches a tuple whose elements respectively match the internal patterns.

The following function uses tuple patterns to check whether two lists are identical:

```
def equal(l0: List[Int], l1: List[Int]): Boolean =
  (l0, l1) match {
    case (h0 :: t0, h1 :: t1) =>
      h0 == h1 && equal(t0, t1)
    case (Nil, Nil) => true
    case _ => false
  }
```

Pattern Guards

A binary search tree is

- the empty tree or
- a tree containing an integral root element and two child trees.

```
sealed trait Tree
case object Empty extends Tree
case class Node(root: Int, left: Tree, right: Tree) extends Tree
```

The function `add` takes a tree and an integer as arguments and returns a tree obtained by adding the integer to the tree. If the integer is an element of the given tree, the tree itself is the return value.

```
def add(t: Tree, n: Int): Tree =
  t match {
    case Empty => Node(n, Empty, Empty)
    case Node(m, t0, t1) =>
      if (n < m)
        Node(m, add(t0, n), t1)
      else if (n > m)
```

```
        Node(m, t0, add(t1, n))
      else
        t
  }
```

An expression corresponding to the second pattern uses `if-else`. Pattern guards allow adding constraints to patterns. A pattern in the form of `[pattern] if [expression]` matches a value if the value matches the pattern, and the expression results in `true`. The following version of `add` uses pattern guards:

```
def add(t: Tree, n: Int): Tree =
  t match {
    case Empty => Node(n, Empty, Empty)
    case Node(m, t0, t1) if n < m =>
      Node(m, add(t0, n), t1)
    case Node(m, t0, t1) if n > m =>
      Node(m, t0, add(t1, n))
    case _ => t
  }
```

Guarded patterns may be inexhaustive and need care.

```
def add(t: Tree, n: Int): Tree =
  t match {
    case Empty => Node(n, Empty, Empty)
    case Node(m, t0, t1) if n < m =>
      Node(m, add(t0, n), t1)
    case Node(m, t0, t1) if n > m =>
      Node(m, t0, add(t1, n))
  }
```

The patterns in the above code is not exhaustive, but the compiler does not warn programmers about the inexhaustivity.

Patterns with Backticks

The function `remove` takes a tree and an integer as arguments and returns a tree obtained by removing the integer from the tree. If the integer is not an element of the tree, the given tree itself is the return value. `removeMin` is a helper function used by `remove`. It returns the pair of the smallest element of a given tree and a tree obtained by removing the element from the tree.

```
def removeMin(t: Tree): (Int, Tree) = {
  t match {
    case Node(n, Empty, t1) =>
      (n, t1)
    case Node(n, t0: Node, t1) =>
      val (min, t2) = removeMin(t0)
      (min, Node(n, t2, t1))
  }
```

```
}

def remove(t: Tree, n: Int): Tree = {
  t match {
    case Empty =>
      Empty
    case Node(m, t0, Empty) if n == m =>
      t0
    case Node(m, t0, t1: Node) if n == m =>
      val res = removeMin(t1)
      val min = res._1
      val t2 = res._2
      Node(min, t0, t2)
    case Node(m, t0, t1) if n < m =>
      Node(m, remove(t0, n), t1)
    case Node(m, t0, t1) if n > m =>
      Node(m, t0, remove(t1, n))
  }
}
```

`Node(‘n‘, t0, Empty)` can replace `case None(m, t0, Empty) if n == m`. The pattern `Node(n, t0, Empty)` defines a new variable `n` and makes `n` refer to the root element; it does not check whether the root element equals `n`. However, backticks prohibit defining a new variable and allow to compare the root element to `n` in the scope.

```
def remove(t: BinTree, n: Int): BinTree = {
  t match {
    case Empty =>
      Empty
    case Node(‘n‘, t0, Empty) =>
      t0
    case Node(‘n‘, t0, t1: Node) =>
      val res = removeMin(t1)
      val min = res._1
      val t2 = res._2
      Node(min, t0, t2)
    case Node(m, t0, t1) if n < m =>
      Node(m, remove(t0, n), t1)
    case Node(m, t0, t1) if n > m =>
      Node(m, t0, remove(t1, n))
  }
}
```

5.4 Applications of Pattern Matching

Variable Definitions

It is possible to define variables with pattern matching.

```
val (n, m) = (1, 2)
```

```
assert(n == 1 && m == 2)

val (a, b, c) = ("a", "b", "c")
assert(a == "a" && b == "b" && c == "c")

val h :: t = List(1, 2, 3, 4)
assert(h == 1 && t == List(2, 3, 4))

val Add(l, r) = Add(Num(1), Num(2))
assert(l == Num(1) && r == Num(2))
```

Pattern matching helps programmers declare variables concisely, but a match error occurs when the pattern does not match the right-hand-side value. It is desirable to use pattern matching only when there is a guarantee that the match succeeds. Since a tuple pattern always matches a tuple value of the same length, tuple patterns are widely used for variable definitions.

Anonymous Functions

The function `toSum` takes a list of pairs of two integers as arguments and returns a list whose elements are the sums of the integers in the pairs.

```
def toSum(l: List[(Int, Int)]): List[Int] =
  l.map(p => p match {
    case (n, m) => n + m
  })

val l = List((0, 1), (2, 3), (3, 4))
assert(toSum(l), List(1, 5, 7))
```

The anonymous function directly uses parameter `p` as the target of the pattern matching. Scala allows simplification of `x => x match { ... }` to `{ ... }`. Therefore, we can use an enumeration of patterns as an anonymous function.

```
def toSum(l: List[(Int, Int)]): List[Int] =
  l.map({ case (n, m) => n + m })
```

For Loops

`toSum` can use a for expression instead of `map`.

```
def toSum(l: List[(Int, Int)]): List[Int] =
  for (p <- l)
    yield p match { case (n, m) => n + m }
```

For expressions directly support pattern matching.

```
def toSum(l: List[(Int, Int)]): List[Int] =
  for ((n, m) <- l)
    yield n + m
```

The code is readable and concise.

5.5 Options

The option type is a widely-used ADT. It represents a value whose existence is optional. This section introduces the option type and explains the usage of options.

Consider the function `get`, which takes a list and integer `n` as arguments and returns the `n`th element of the list. It is problematic when `n` is negative or exceeds the length of the list. Throwing exceptions is a widely used solution in imperative languages. In Scala, `throw [expression]` throws an exception. For convenience, we define the function `error`, which throws an exception, like below and use it throughout the book.

```
def error(msg: String) = throw new Exception(msg)

def get(l: List[Int], n: Int): Int =
  if (n < 0)
    error("index out of bounds")
  else l match {
    case Nil =>
      error("index out of bounds")
    case h :: t =>
      if (n == 0)
        h
      else
        get(t, n - 1)
  }
```

Throwing an excpeion is a simple and effective solution. However, exceptions have two problems. First, exceptions should be handled by exception handlers.

```
try {
  get(List(1, 2), 2)
} catch {
  case e: Exception =>
    // prints "index out of bounds"
    println(e.getMessage)
}
```

If an exception is not handled properly, it will eventually cause a run-time error and terminate the execution.

```
get(List(1, 2), 2)
```

```
java.lang.Exception: index out of bounds
```
Run-time error

The Scala compiler does not check whether exceptions are handled properly. It means that there will not be any compile-time error even if there is a possibility of unhandled exceptions.

Another problem of exceptions is that exception handling is not local. When an exception is thrown, the control flow suddenly jumps to the position of the nearest exception handler. Non-local transition of the control flow usually hinders programmers from understanding code. Therefore, implementing `get` without exceptions is desirable.

The first attempt is to use `null`. `null` is a value that denotes that it does not refer to any existing object. We can try to make `get` return `null` when a given index is invalid.

```
def get(l: List[Int], n: Int): Int =
  if (n < 0)
    null
  else l match {
    case Nil => null
    case h :: t =>
      if (n == 0)
        h
      else
        get(t, n - 1)
  }
```

```
    null
    ^
error: an expression of type Null is ineligible
for implicit conversion

    case Nil => null
                ^
error: an expression of type Null is ineligible
for implicit conversion
```

Run-time error

Unfortunately, `null` is not an element of `Int` in Scala. The compiler rejects the code. Even with the assumption that we can treat `null` as `Int`, using `null` is a bad solution. Dereferencing `null` causes a run-time error, which is the well-known `NullPointerException`. The compiler does not check whether `null` is dereferenced. Therefore, using `null` is nothing better than using exceptions. Use of `null` has been criticized enormously because `null` is extremely error-prone. Even Tony Hoare, the inventor of `null`, said that inventing `null` was a terrible mistake:

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965.[3]

3: https://en.wikipedia.org/wiki/Null_pointer#History

The second attempt is to use a particular error-indicating value, e.g. `-1`.

```
def get(l: List[Int], n: Int): Int =
  if (n < 0)
    -1
  else l match {
    case Nil =>
      -1
    case h :: t =>
      if (n == 0)
        h
```

```
      else
        get(t, n - 1)
  }
```

The strategy has an obvious problem. The caller cannot distinguish two situations:

- The list contains `-1`.
- The index is invalid.

Default values can be successful solutions for certain purposes but do not fit `get`.

Instead of using a fixed particular value in `get`, the caller can specify the default value.

```
def getOrElse(l: List[Int], n: Int, default: Int): Int =
  if (n < 0)
    default
  else l match {
    case Nil =>
      default
    case h :: t =>
      if (n == 0)
        h
      else
        getOrElse(t, n - 1, default)
}
```

It works well when an appropriate default value exists. However, when checking failures is per se important, the new strategy is as bad as the previous strategy. There is no way to distinguish an element and the default value.

Functional languages provide the option type to handle erroneous situations safely. As the name implies, it represents an optional existence of a value. The Scala standard library defines the option type like below.[4]

4: We will not see what `[+A]` and `Nothing` are here. You can understand the overall ADT structure without knowing those concepts.

```
sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some[A](value: A) extends Option[A]
```

An option that may have a value of type `T` has type `Option[T]`. An option is either `None` or `Some`. `None` is a value that does not denote any value and similar to `null`. It indicates a problematic situation. Like `Nil`, it is defined as a case object because every `None` is identical. `Some` constructs a value that denotes that a value exists. It is similar to a reference to a real object and indicates that computation has succeeded.

The following code defines `getOption`, which returns an option.

```
def getOption(l: List[Int], n: Int): Option[Int] =
  if (n < 0)
    None
  else l match {
```

```
    case Nil =>
      None
    case h :: t =>
      if (n == 0)
        Some(h)
      else
        getOption(t, n - 1)
  }

assert(getOption(List(1, 2), 0) == Some(1))
assert(getOption(List(1, 2), 2) == None)
```

For an invalid index, the return value is `None`. The caller can notice that the operation has failed by `None`. Otherwise, the function packs an element inside `Some` to make the return value.

The Scala standard library uses options in many places. Various methods return options. For example, `headOption` of a list returns `None` when the list is empty. Otherwise, `Some` containing the head of the list is returned.

```
assert(List().headOption == None)
assert(List(1).headOption == Some(1))
```

Also, `get` of a map returns `None` when the map does not have a given key. Otherwise, `Some` containing the value corresponding to the key is returned.

```
val m = Map(1 -> "one", 2 -> "two")
assert(m.get(0) == None)
assert(m.get(1) == Some("one"))
```

Pattern matching allows programmers to deal with options by distinguishing the `None` and `Some` cases. In addition, like the methods of lists, options also provide methods to abstract common patterns. We are going to see two methods: `map` and `flatMap`.

`map` can be used when we want to apply some computation only when the previous computation has succeeded. `map` takes a single argument, which must be a function. `opt.map(f)` results in `None` when `opt` is `None`. If `opt` is `Some(v)`, then `opt.map(f)` evaluates to `Some(f(v))`.

As an example, let us consider a map containing students. Names are the keys, and students are the values. We want to find a student by a name and get one's height only when the student exists. It can be implemented with `map`.

```
def getHeight(
  m: Map[String, Student],
  name: String
): Option[Int] =
  m.get(name).map(_.height)
```

If `m.get(name)` is `None`, then `m.get(name).map(_.height)` also is `None`. Otherwise, `m.get(name)` should be `Some(student)`, and `m.get(name).map(_.height)` will result in `Some(student.height)`.

In summary, `map` is useful when the computation consists of two steps, and the first step can fail.

`flatMap` is similar to `map` but a bit different. It is useful when the computation consists of two steps, and both steps can fail. `flatMap` takes a single argument, which must be a function that returns an option. `opt.flatMap(f)` results in `None` when `opt` is `None`. If `opt` is `Some(v)`, then `opt.map(f)` evaluates to `f(v)`.

Let us consider a list of names and a map like before. When the list is nonempty, we will find a student with the first name in the list from the map. It is a typical application of `flatMap`.

```
def getStudent(
  l: List[String],
  m: Map[String, Student]
): Option[Student] =
  l.headOption.flatMap(m.get)
```

The standard library provides many other useful methods for options.[5]

5: https://www.scala-lang.org/api/current/scala/Option.html

Untyped Languages

Syntax and Semantics 6

This chapter is about syntax and semantics.

Syntax of a programming language decides the appearance of the language. Syntax consists of concrete syntax and abstract syntax. While concrete syntax describes programs as strings, abstract syntax describes the structures of programs as trees. Parsing is the process bridging the gap between concrete syntax and abstract syntax. A string is transformed to a tree by parsing. This chapter explains concrete syntax, abstract syntax, and parsing.

Semantics of a programming langauge determines the behavior of each program. This chapter explains how we can define the semantics of a language. In addition, we will see what syntactic sugar is.

6.1 Concrete Syntax

People write programs with strings. Some strings are valid programs, while other strings are not. For example, consider the following code:

```
println()
```

It is a valid Scala program. On the other hand, the following code is not a valid Scala program.

```
println(
```

Concrete syntax determines whether a certain string is a program or not. According to the concrete syntax of Scala, `println()` is a program, but `println(` is not because the closing parenthesis is missing. Without concrete syntax, programmers cannot know whether a given string is a program or not. Concrete syntax is one of the essential elements of a programming language. This section defines concrete syntax formally and explains how concrete syntax can be specified.

The first thing to do is to define strings since programs are represented as strings. A string is a finite sequence of characters. The definition of a character varies in programming languages. In some languages, a character is limited to those expressed by the ASCII code. In other cases, every Unicode symbol is considered as a character. As we want to deal with general programming languages, we do not fix the characters to be a specific set. Instead, we assume that a set of characters is given and do not care about which characters exactly exist in the set. From now on, C is the set of every character:

$$C = \text{the set of every character} = \{c \mid c \text{ is a character}\}$$

Now, a character is an element of C.

Once characters are defined, we can define strings with the definition of characters. S is the set of every string:

$$S = \text{the set of every string} = \{\texttt{"}c_1 \cdots c_n\texttt{"} \mid c_1, \ldots, c_n \in C\}$$

A string is an element of S, which is a sequence of zero or more characters. For example, if `'a'`, `'b'` $\in C$, then `"aba"` $\in S$.

The definition of strings differs from the definition of programs because some strings are not programs. As we have seen already, `"println()"` is a program, but `"println("` is not although both are strings.

Defining concrete syntax is to define which strings are programs. Therefore, we can say that concrete syntax determines the set of every program. Let P be the set of every program.

$$P = \text{the set of every program} = \{p \mid p \text{ is a program}\}$$

A program is an element of P. Then, P has only one restriction: P should be a subset of S as every program is a string.

$$P \subseteq S$$

The concrete syntax of each language defines its own P as a subset of S. Each language has different P from each other. For instance, `"println()"` $\in P$ and `"println("` $\notin P$, where P is defined by the concrete syntax of Scala. At the same time, there can be a language whose P does not have `"println()"` as an element.

To define the concrete syntax of a language, we need to define the set P. The problem is that P is usually an infinite set. There are infinitely many programs in each language. Defining an infinite set is difficult because we cannot enumerate every element of an infinite set. We need a way to define infinite sets to define concrete syntax.

The most popular way to define concrete syntax is *Backus-Naur form* (BNF). It defines a set of strings in an intuitive way. It provides a constructive definition, i.e. it lets us know how we can construct an element of the set. To define concrete syntax with BNF, we need to discuss the concepts used in BNF first.

BNF has three concepts: terminals, nonterminals, and expressions. A *terminal* is a string. For example, `"0"` and `"ab"` are terminals. A *nonterminal* is a name between a pair of angle brackets and denotes a set of strings. For instance, `<digit>` is a nonterminal and may denote the set {`"0"`, `"1"`, `"2"`, `"3"`, `"4"`, `"5"`, `"6"`, `"7"`, `"8"`, `"9"`}. An *expression* is an enumeration of one or more terminals/nonterminals. Therefore, all of the following are expressions:

- `"abc"` (a single terminal)
- `"0"` `"1"` (multiple terminals)
- `<digit>` (a single nonterminal)
- `<digit>` `<number>` (multiple nonterminals)

- `"-"` `<number>` (a single terminal and a single nonterminal)

An expression also denotes a set of strings. The set denoted by an expression is the concatenation of strings denoted by its components. For example, in the expression `"0" "1"`, `"0"` denotes `"0"`, and `"1"` denotes `"1"`. Therefore, `"0" "1"` denotes the set {`"01"`}. If `<digit>` denotes the set {`"0"`, `"1"`, `"2"`, `"3"`, `"4"`, `"5"`, `"6"`, `"7"`, `"8"`, `"9"`}, then `"0"` `<digit>` denotes {`"00"`, `"01"`, `"02"`, `"03"`, `"04"`, `"05"`, `"06"`, `"07"`, `"08"`, `"09"`}.

Now, let us see how we can define a new set from scratch. BNF allows us to define the set denoted by a nonterminal with the following form:

```
[nonterminal] ::= [expression] | [expression] | ...
```

The vertical bars in the right hand side separate distinct expressions. The expressions define the set denoted by the nonterminal. The union of the sets denoted by the expressions equals the set denoted by the nonterminal. For example, the following definition makes `<digit>` denote {`"0"`, `"1"`, `"2"`, `"3"`, `"4"`, `"5"`, `"6"`, `"7"`, `"8"`, `"9"`}:

```
<digit> ::= "0" | "1" | "2" | "3" | "4"
          | "5" | "6" | "7" | "8" | "9"
```

From now on, we are going to define the concrete syntax of a tiny language named `AE` to show example usage of BNF. `AE` stands for arithmetic expressions. Its features is limited to addition and subtraction of decimal integers.

`AE` programs should be able to express decimal integers. Thus, the following strings should be programs of `AE`:

- `"0"`
- `"1"`
- `"-10"`
- `"42"`

At the same time, programs should be able to express addition and subtraction. Thus, the following strings also should be programs:

- `"0+1"`
- `"-2-1"`
- `"1+-3+42"`
- `"4-3+2-1"`

First, we can define the set of every string that represents a decimal integer in BNF. It can be done with the following definitions:

```
<digit>  ::= "0" | "1" | "2" | "3" | "4"
           | "5" | "6" | "7" | "8" | "9"
<nat>    ::= <digit> | <digit> <nat>
<number> ::= <nat> | "-" <nat>
```

We know that `<digit>` denotes {`"0"`, `"1"`, `"2"`, `"3"`, `"4"`, `"5"`, `"6"`, `"7"`, `"8"`, `"9"`}. Since `<digit>` is one way to construct an element of `<nat>`, every string denoted by `<digit>` is also a string of `<nat>`. Hence, {`"0"`,

"1", "2", "3", "4", "5", "6", "7", "8", "9"} is a subset of the set denoted by <nat>. At the same time, <digit> <nat> is the other way to construct an element. We can make a new element of <nat> by selecting strings from <digit> and <nat>, respectively. For example, <digit> can denote "1", and <nat> can denote "0". Therefore, "10" is an element of <nat>. By repeating this process, we can construct infinitely many strings, e.g. "110" by concatenating "1" and "10", "1110" by concatenating "1" and "110", and so on. In the end, we can conclude that <nat> denotes the set of every string that consists of the characters from '0' to '9', i.e. every string that represents a decimal natural number.[1]

1: This book considers zero as a natural number.

Finding the set denoted by <number> is easier. Since <nat> is one way to construct an element of <number>, the set denoted by <nat> is a subset of the set denoted by <number>. The expression "-" <nat> is the other way to construct an element. It implies that if we concatenate "-" and a string denoted by <nat>, we can get a new element of <number>. In conclusion, <number> denotes the set of every string that represents a decimal integer.

It is enough to add only addition and subtraction to complete the definition of the concrete syntax.

```
<expr> ::= <number> | <expr> "+" <expr> | <expr> "-" <expr>
```

In a similar way, we can figure out which set is denoted by <expr>. The set includes every string that represents arithmetic expression consisting of decimal integers, addition, and subtraction. We can say that <expr> defines *P* of AE, and the concrete syntax of AE is defined now.

6.2 Abstract Syntax

Defining syntax solely with concrete syntax has problems from both language users' and language designers' perspectives.

Programmers usually learn multiple languages. Languages considerably vary in their concrete syntax. Consider a function that takes two integers as arguments and returns the sum of the integers. We can implement the function in four different languages like below.

- Python

```
def add(n, m):
    return n + m
```

- JavaScript

```
function add(n, m) {
    return n + m;
}
```

- Racket

```
(define (add n m) (+ n m))
```

- OCaml

```
let add n m = n + m
```

They look so different even though they define the same function. The keyword to define a function is `def` in Python, `function` in JavaScript, `define` in Racket, and `let` in OCaml. It is not the only difference. Python and JavaScript need parentheses and commas for parameters, while Racket and OCaml do not. JavaScript puts function bodies inside curly braces. Racket treats `+` as a prefix operator, while the others treat `+` as an infix operator. These differences are the differences between the concrete syntax of each language. Various forms of concrete syntax hinder programmers from learning multiple languages easily.

However, their structures are quite the same. In every language, a function definition consists of a name, parameters, and a body expression. The above example defines a function whose name is `add`, parameters are `n` and `m`, and body is an expression that adds `n` and `m`. In every language in the example, an addition expression consists of two operands. The body expression uses `n` and `m` as the operands of the addition.

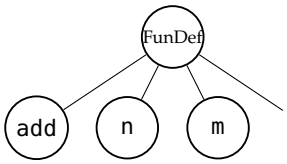
Thus, programmers should focus on the structures of programs, rather than strings per se, to learn multiple languages easily. The structures remain the same even when the strings vary.

At the same time, concrete syntax cares about tedious details that language designers want to ignore. For example, both `"2+1"` and `"02+001"` are `AE` programs. They are different strings but represent the same arithmetic expression: 2 + 1. When the designers of `AE` define logic to evaluate arithmetic expressions, distinction between 2 + 1 and 2 − 1 is important, but distinction between `"2+1"` and `"02+001"` is completely unnecessary. The designers want to focus only on the structures of programs but not strings.

For both programmers and designers, concrete syntax is problematic because it describes only strings and does not give good abstraction of structures even though people want to focus on the structures. Of course, we cannot discard the notion of concrete syntax. Everyone write programs as strings, and concrete syntax is essential for that step. At the same time, we need a way to describe the structures of programs without being affected by differences in strings. To meet the need, we introduce another notion of syntax: abstract syntax. Concrete syntax and abstract syntax are complementary. They collectively construct the syntax of a language.

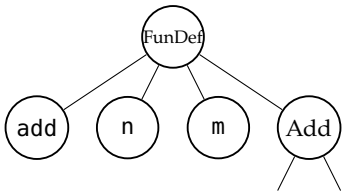
Abstract syntax describes the structure of a program as a tree. A program consists of multiple components. Each component consists of subcomponents again. Trees formally express such recursive structures. A component can be represented as a tree whose root describes the sort of a component and children are the trees representing the subcomponents.

As an example, let us express the function `add` as a tree. The function definition has four components: the name `add`, the first parameter `n`, the second parameter `m`, and the body expression. The following tree represents the function definition:



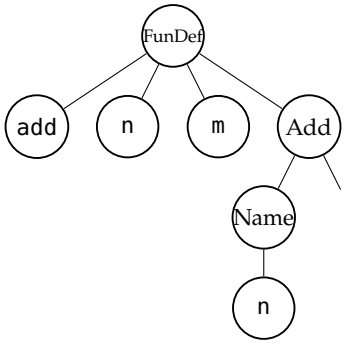
The root of the tree is the symbol FunDef, which explains that this tree represents a function definition. The tree has four children: `add`, `n`, `m`, and the body expression. We do not know how to draw the tree representing the body expression yet.

The body expression is an addition expression. It has two components: the operands of the addition.



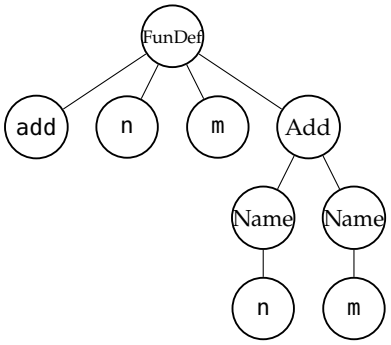
The root of the tree is Add as the expression is addition. It has two children: the operand expressions.

The first operand expression consists of a single component: `n`.



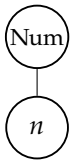
The root of the tree is Name, since the expression is just a name. The only child is `n`.

The second operand expression can be similarly represented as a tree.



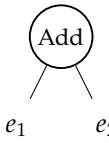
The above tree represents the structure of the function definition. It is independent of its underlying programming language. The tree can be a Python function definition and a JavaScript function definition at the same time. By expressing programs with trees, we can ignore unnecessary details in strings and focus on the structures of programs.

As abstract syntax treats programs as trees, defining the abstract syntax of a language is to define the set of every tree that represents a program. Let us define the abstract syntax of `AE`. In `AE`, every natural number is a program. A natural number program has only one component: the natural number itself. Therefore, the following fact is true, where E denotes the set of every program:

If $n \in \mathbb{Z}$, then 

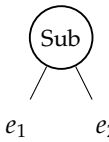
$\in E$.

Addition of two arithmetic expressions is also a program. Such a program has two components: the left and right operands. Each operand is an arithmetic expression and thus a program. Therefore, the following fact is true:

If $e_1, e_2 \in E$, then 

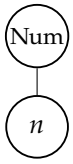
$\in E$.

Subtraction is similar.

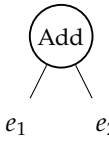
If $e_1, e_2 \in E$, then 

$\in E$.

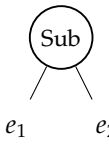
By collecting all the above facts, we can define the abstract syntax of `AE` as the smallest set E satisfying the following conditions.

If $n \in \mathbb{Z}$, then 

$\in E$.

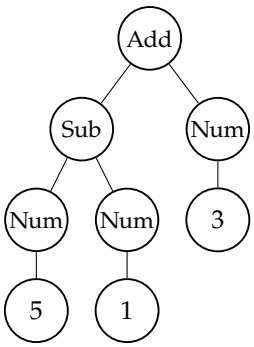
If $e_1, e_2 \in E$, then 

$\in E$.

If $e_1, e_2 \in E$, then 

$\in E$.

For example, the following tree represents an `AE` program:

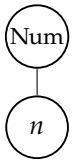


We call a tree that is an element of the set defined by abstract syntax an *abstract syntax tree* (AST).

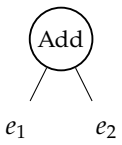
Abstract syntax can be easily implemented with ADTs in Scala. The following code implements the abstract syntax of `AE`:

```
sealed trait AE
case class Num(value: Int) extends AE
case class Add(left: AE, right: AE) extends AE
case class Sub(left: AE, right: AE) extends AE
```

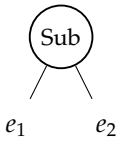
`Num(`n`)` corresponds to



`Add(`e_1`, `e_2`)` corresponds to



`Sub(`e_1`, `e_2`)` corresponds to



The previous AST can be written as `Add(Sub(Num(5), Num(1)), Num(3))` in Scala.

It is inconvenient to draw a tree every time we need to express a program. For this reason, people usually use notations that look like code to represent trees. For example, we can simply write n instead of drawing a tree whose root is Num and only child is n when it is clear that n denotes an AST from the context. Similarly, we can use $e_1 + e_2$ and $e_1 - e_2$ instead of trees that represent addition and subtraction, respectively. Note that + and − in the notations are not the mathematical addition and subtraction operators. They are just parts of the notations and do not have any meaning.

We can define the abstract syntax of `AE` again by using the above notations. E is the smallest set satisfying the following conditions:

- If $n \in \mathbb{Z}$, then $n \in E$.
- If $e_1, e_2 \in E$, then $e_1 + e_2 \in E$.
- If $e_1, e_2 \in E$, then $e_1 - e_2 \in E$.

Even though the notations themselves do not look like trees at all, they still represent ASTs. Also, symbols like + and − do not have any meaning. It is extremely important to keep these points in your mind. Otherwise, you will mix abstract syntax using notations up with concrete syntax in the end.

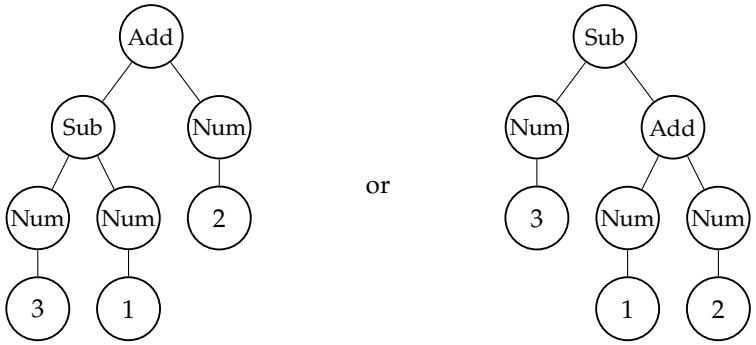
Notations are just notations. You can define different notations and use them. For example, one may use $ADD\ e_1\ e_2$ instead of $e_1 + e_2$ to represent addition. You can freely choose notations, but once you define them, you should consistently use them not to make other people confused.

To make the definition of abstract syntax more concise, we adopt BNF to the definition of abstract syntax. We can re-define the abstract syntax of `AE` with BNF:

$$e ::= n \mid e + e \mid e - e$$

We call each symbol that denotes a particular element in abstract syntax a *metavariable*. It is called **meta**variable because it is a variable at a meta-level, not the level of the defined programming language. For example, e is a metavariable that ranges over programs, and n is a metavariable that ranges over integers.

We often use parentheses to express elements of abstract syntax without ambiguity. For instance, $3 - 1 + 2$ can be interpreted in two ways:



If we write $(3 - 1) + 2$, it is clear that it denotes the former. Otherwise, we write $3 - (1 + 2)$ to denote the latter.

6.3 Parsing

Concrete syntax considers programs as strings, while abstract syntax considers programs as trees. Parsing bridges this gap. *Parsing* is a process of transforming a string following concrete syntax into an AST. A parser is a program that parses input. We can consider a parser as a partial function from S (the set of every string) to E (the set of every AST).

$$parse : S \twoheadrightarrow E$$

Partial functions

A partial function from a set A to a set B is a function from a subset S of A to B. S is called the domain of definition, or just domain in short, of the partial function. While $A \to B$ is a set of functions from A to B, $A \twoheadrightarrow B$ is a set of partial functions from A to B.

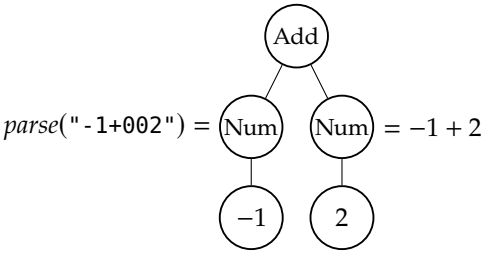
Let f be a partial function from A to B. Then, there can be $a \in A$ such that $f(a)$ is undefined. From a programmers' perspective, f can be interpreted as a function from A to `Option[`B`]`, where `None` means that the image is undefined and `Some(`b`)` means that the image is b.

The result of *parse* is undefined when an input does not belong to P (the set of every program). That is why *parse* is a partial function. When an input belongs to P, *parse* results in its corresponding AST.

Consider the parser of `AE`. The results of *parse* are undefined for the following strings as they are not `AE` programs:

- `1+`
- `2*4`
- `0++3`

On the other hand, below is an example of when *parse* succeeds.



This book does not discuss implementation of parsers.

6.4 Semantics

Syntax is an essential element of a programming language. It allows us to know which strings are programs and what the structures of programs are. However, syntax does not explain execution of programs. Programmers write programs to execute them. They should know what will happen when their programs are executed. Therefore, we need semantics in addition to syntax. Semantics is the other essential element of a programming language. It defines the behaviors of programs.

Let us define the semantics of AE. Semantics is defined based on abstract syntax. The structure of a program determines its behavior. Since abstract syntax represents the structure, it is natural to use abstract syntax for semantics. The semantics of AE defines the semantics of each AE program, where the semantics of a program means things that happen when the program is executed. When an AE program is executed, it does one thing: outputs the result of the evaluation of the arithmetic expression. For example, 0 + 1 should result in 1 if the semantics is defined correctly. Note that + in 0 + 1 does not mean addition, and we cannot say anything about the result of 0 + 1 until the semantics is defined. To make AE a reasonable language, we must define the semantics of AE so that 0 + 1 results in 1.

There are infinitely many programs. We cannot define the semantics of each program separately. We need to utilize the structures of programs defined by the abstract syntax. According to the abstract syntax, programs can split into three groups: n, $e_1 + e_2$, and $e_1 - e_2$. By defining the semantics of each group once, we can complete the semantics of infinitely many programs in a finite method.

The simplest case is n. n is an expression consisting of an integer. An integer evaluates to itself. We represent this semantics as the following rule:

Rule Num
n evaulates to n.

We can conclude the following facts by using Rule Num.

- 1 evaulates to 1.
- 5 evaulates to 5.

The next case is $e_1 + e_2$. As e_1 is an arithmetic expression, it results in some integer. Let n_1 be the integer. Similarly, e_2 also results in some integer. Let n_2 be the integer. Then, the result of $e_1 + e_2$ is the sum of n_1 and n_2. In this chapter, we use $+_{\mathbb{Z}}$ instead of + to denote mathematical addition. It will help you distinguish mathematical addition from + used for the abstract syntax. Once you become familiar with syntax and semantics, you can easily distinguish them by checking the context even if both are denoted by +. From the next chapter, we will use + for both abstract syntax and mathematical addition. The following rule defines the semantics of $e_1 + e_2$.

Rule Add
If e_1 evaluates to n_1, and e_2 evaluates to n_2,

then $e_1 + e_2$ evaluates to $n_1 +_{\mathbb{Z}} n_2$.

We can define the semantics of $e_1 - e_2$ in a similar way. Like $+_{\mathbb{Z}}$, we use $-_{\mathbb{Z}}$ for mathematical subtraction in this chapter. The following rule defines the semantics of $e_1 - e_2$.

Rule Sub
If e_1 evaluates to n_1, and e_2 evaluates to n_2,
then $e_1 - e_2$ evaluates to $n_1 -_{\mathbb{Z}} n_2$.

These three rules are all of the semantics of AE. We now know the behavior of every AE program. For example, consider $(3 - 1) + 2$. The following steps show why $(3 - 1) + 2$ evaluates to 4.

1. (By Rule Num) 3 evaluates to 3.
2. (By Rule Num) 1 evaluates to 1.
3. (By Rule Sub) If 3 evaluates to 3 and 1 evaluates to 1, then $3 - 1$ evaluates to 2.
4. (By 1, 2, and 3) $3 - 1$ evaluates to 2.
5. (By Rule Num) 2 evaluates to 2.
6. (By Rule Add) If $3 - 1$ evaluates to 2 and 2 evaluates to 2, then $(3 - 1) + 2$ evaluates to 4.
7. (By 4, 5, and 6) $(3 - 1) + 2$ evaluates to 4.

Now, let us define the semantics of AE in a more mathematical way. The semantics defines the result of the execution of each program. Here, the result is an integer. We can say that semantics outputs an integer when a program is given. Thus, the semantics can be considered as a function from a program to an integer.

$$eval : E \to \mathbb{Z}$$

For each $e \in E$, there should exist a unique integer $eval(e)$. It is obviously true in AE. Every arithmetic expression evaluates to a unique integer.

However, defining semantics as a function is a bad choice in other languages. Some programs do not produce any results. Nonterminating programs are such examples. Programs that incur run-time errors also belong to this category. You will see programs with run-time errors in the next chapter. Moreover, there is a program whose result is not unique. We call such programs nondeterministic programs. For example, the behavior of a concurrent program with multiple threads depends on how the threads are interleaved during execution. If the threads are interleaved differently, the result may change. Programs without results and nondeterministic programs prevent us from defining semantics as a function. We should define semantics as a relation. Even though the semantics of AE can be defined as a function, we define the semantics as a relation to make the discussion of this chapter easily extendable to other languages.

We define the semantics of AE as $\Rightarrow$, a binary relation over E and $\mathbb{Z}$.

Binary relations

A binary relation over sets A and B is a subset of $A \times B$, where $A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$.

Let R be a binary relation over A and B. Then, $R \subseteq A \times B$. For $a \in A$ and $b \in B$, we write $a\ R\ b$ when $(a, b) \in R$. For example, $<$ is a binary relation over $\mathbb{Z}$ and $\mathbb{Z}$, and we can write $1 < 2$ instead of $(1, 2) \in <$.

$$\Rightarrow \subseteq E \times \mathbb{Z}$$

$(e, n) \in \Rightarrow$, i.e. $e \Rightarrow n$ implies that e evaluates to n.

Let us define the semantics again with mathematical concepts.

Rule Num
$n \Rightarrow n$.

Rule Add
If $e_1 \Rightarrow n_1$ and $e_2 \Rightarrow n_2$,
then $e_1 + e_2 \Rightarrow n_1 +_{\mathbb{Z}} n_2$.

Rule Sub
If $e_1 \Rightarrow n_1$ and $e_2 \Rightarrow n_2$,
then $e_1 - e_2 \Rightarrow n_1 -_{\mathbb{Z}} n_2$.

We use one more mathematical concept: inference rules. An *inference rule* is a rule to prove a new proposition from given propositions. An inference rule has the following form:

$$\frac{premise_1 \qquad premise_2 \qquad \cdots \qquad premise_n}{conclusion}$$

It consists of a horizontal line, propositions above the line, and a proposition below the line. We call the propositions above the line *premises* and the proposition below the line a *conclusion*. The rule means that if every premise is true, then also the conclusion is true. A single inference rule can have zero or more premises. A rule without premises implies that its conclusion is always true. When a rule does not have any premises, we can omit the horizontal line.

Let us define the semantics of `AE` with inference rules.

$$n \Rightarrow n \quad [\textsc{Num}]$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 + e_2 \Rightarrow n_1 +_{\mathbb{Z}} n_2} \quad [\textsc{Add}]$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 - e_2 \Rightarrow n_1 -_{\mathbb{Z}} n_2} \quad [\textsc{Sub}]$$

As you can see, the rules are much clearer and more concise than the rules written in a natural language.

We can prove $(3 - 1) + 2 \Rightarrow 4$ with the rules. We usually draw a proof tree when we prove a proposition with inference rules. A *proof tree* is a tree whose root is the proposition to be proven. Each node of the tree is a proposition, and the children nodes of a node are evidences supporting that the proposition of the node is true. Unlike most trees in computer science, we place the root of a proof tree at the bottom. Every node is placed below its children.

The following proof tree proves $3 \Rightarrow 3$.

$$3 \Rightarrow 3$$

The tree has only the root node because Rule Num does not have any premises.

Similarly, the following proof tree proves $1 \Rightarrow 1$.

$$1 \Rightarrow 1$$

We draw the following proof tree with Rule Sub and the above trees to prove $3 - 1 \Rightarrow 2$.

$$\frac{3 \Rightarrow 3 \qquad 1 \Rightarrow 1}{3 - 1 \Rightarrow 2}$$

By using Rule Num again, we prove $2 \Rightarrow 2$.

$$2 \Rightarrow 2$$

Finally, we get the proof tree of $(3 - 1) + 2 \Rightarrow 4$.

$$\frac{\dfrac{3 \Rightarrow 3 \qquad 1 \Rightarrow 1}{3 - 1 \Rightarrow 2} \qquad 2 \Rightarrow 2}{(3 - 1) + 2 \Rightarrow 4}$$

To explain what proof trees are, we have drawn the proof tree from its leaf nodes. However, we usually draw a proof tree from the root node. We start by drawing a horizontal line and writing the program we want to evaluate.

$$\frac{}{(3 - 1) + 2 \Rightarrow}$$

Then, we find which inference rule can be applied. In this case, we can use Rule Add since the program is addition.

$$\dfrac{\dfrac{}{3-1\Rightarrow} \qquad 2\Rightarrow}{(3-1)+2\Rightarrow}$$

We need to evalaute $3-1$ and 2 respectively. Let us focus on $3-1$ first. Since $3-1$ is subtraction, we use Rule Sub.

$$\dfrac{\dfrac{3\Rightarrow \qquad 1\Rightarrow}{3-1\Rightarrow} \qquad 2\Rightarrow}{(3-1)+2\Rightarrow}$$

We can conclude that $3\Rightarrow 3$ from Rule Num.

$$\dfrac{\dfrac{3\Rightarrow 3 \qquad 1\Rightarrow}{3-1\Rightarrow} \qquad 2\Rightarrow}{(3-1)+2\Rightarrow}$$

Similarly, $1\Rightarrow 1$.

$$\dfrac{\dfrac{3\Rightarrow 3 \qquad 1\Rightarrow 1}{3-1\Rightarrow} \qquad 2\Rightarrow}{(3-1)+2\Rightarrow}$$

By subtracting 1 from 3, we get 2.

$$\dfrac{\dfrac{3\Rightarrow 3 \qquad 1\Rightarrow 1}{3-1\Rightarrow 2} \qquad 2\Rightarrow}{(3-1)+2\Rightarrow}$$

We use Rule Num agian and get $2\Rightarrow 2$.

$$\dfrac{\dfrac{3\Rightarrow 3 \qquad 1\Rightarrow 1}{3-1\Rightarrow 2} \qquad 2\Rightarrow 2}{(3-1)+2\Rightarrow}$$

Finally, we can complete the proof tree and prove $(3-1)+2\Rightarrow 4$.

$$\dfrac{\dfrac{3\Rightarrow 3 \qquad 1\Rightarrow 1}{3-1\Rightarrow 2} \qquad 2\Rightarrow 2}{(3-1)+2\Rightarrow 4}$$

Sometimes, we call a proof tree proving the result of a program a *evaluation derivation* since the tree explains how the result of the program is derived.

The way of defining semantics we have seen so far is *big-step operational semantics*. It is called "operational" because it focuses on which **operation**s

happen during execution and "big-step" because it finds the result of a program by taking a single **big** step. There are other ways to define semantics: denotational semantics and small-step operational semantics. Most part of this book uses big-step operational semantics. However, it will use small-step operational semantics to deal with continuations later.

An *interpreter* is a program that takes a program as input and evaluates the program. We can easily implement an interpreter of `AE` according to its semantics. The interpreter consists of a single function that takes an AST as an argument and returns an integer.

```
def interp(e: AE): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l) + interp(r)
  case Sub(l, r) => interp(l) - interp(r)
}
```

6.5 Syntactic Sugar

Syntactic sugar adds a new feature to a language by defining syntactic transformation rules instead of changing the semantics. Syntactic sugar is widely used in real-world programming languages because it allows languages to provide useful features without increasing the burden of the language designers too much.

Suppose that we want to add integer negation to `AE`. It can be done by modifying both syntax and semantics of `AE`. First, we fix the concrete syntax to add integer negation.

```
<expr> ::= <number> | <expr> "+" <expr>
         | <expr> "-" <expr> | "-" "(" expr ")"
```

Similarly, we fix the abstract syntax, too.

$$e ::= n \mid e + e \mid e - e \mid -e$$

The parser should be changed accordingly. For example, `-(03+4)` is parsed to $-(3 + 4)$.

Finally, we add a new rule to the semantics to handle the $-e$ case.

Rule Neg
If e evaluates to n,
then $-e$ evaluates to $-_{\mathbb{Z}}\, n$.

Note that $-_{\mathbb{Z}}$ denotes mathematical negation.

We can express the same thing as an inference rule.

$$\frac{e \Rightarrow n}{-e \Rightarrow -_{\mathbb{Z}}\, n}\ \textsc{Neg}$$

It requires considerable amount of work as we need to fix every component of the language.

Another way to add integer negation to `AE` is to add it as syntactic sugar. It is enough to modify the concrete syntax and the parser. The change in the concrete syntax is the same as before. Now, we fix the parser to parse `"-" "(" expr ")"` to $0 - e$ when `expr` is parsed to e. For example, `-(03+4)` is parsed to $0 - (3 + 4)$. Since $0 -_{\mathbb{Z}} n = -_{\mathbb{Z}} n$ for any integer n, we are done. It is the power of syntactic sugar. Language designers can easily add new features by syntactically transforming them to existing features. The procedure removing syntactic sugar by transformation is called *desugaring*.

We can find various examples of syntactic sugar in real-world languages. For instance, for loops in Scala are supported as syntactic sugar.

> A for comprehension `for (`p `<-` e`) yield` e' is translated to e`.map { case` p `=>` e' `}`.[2]

2: https://www.scala-lang.org/files/archive/spec/2.13/06-expressions.html#for-comprehensions-and-for-loops

In addition, macros in languages like C, Scala, LISP, and Rust can be considered as user-defined syntactic sugar.

6.6 Exercises

1. Consider the following concrete syntax:

```
<expr> ::= <num>
         | "{" "+" <expr> <expr> "}"
         | "{" "*" <expr> <expr> "}"
         | "{" "let" "{" <id> <expr> "}" <expr> "}"
         | <id>
```

   Describe whether each of the following is `expr` and why. Assume that it is allowed to add whitespaces among terminals freely.

   a) `{let {x 5} {+ 8 {* x 2 3}}}`
   b) `{with {x 0} {with {x 7}}}`
   c) `{let {3 5} {+ 8 {- x 2}}}`
   d) `{let {3 y} {+ 8 {* x 2}}}`
   e) `{let {x y} {+ 8 {* x 2}}}`

2. Consider the following concrete syntax:

$$\texttt{espresso} \in \texttt{<coffee>} \qquad \frac{e_1 \in \texttt{<milk>} \quad e_2 \in \texttt{<coffee>}}{e_1\ \texttt{"on"}\ e_2 \in \texttt{<coffee>}}$$

$$\frac{e_1 \in \texttt{<coffee>} \quad e_2 \in \texttt{<milk>}}{e_1\ \texttt{"on"}\ e_2 \in \texttt{<coffee>}} \qquad \frac{e_1 \in \texttt{<flavor>} \quad e_2 \in \texttt{<coffee>}}{e_1\ \texttt{"on"}\ e_2 \in \texttt{<coffee>}}$$

$$\texttt{"milk-foam"} \in \texttt{<milk>} \qquad \texttt{"steamed-milk"} \in \texttt{<milk>}$$

$$\texttt{"caramel"} \in \texttt{<flavor>} \qquad \texttt{"cinnamon"} \in \texttt{<flavor>}$$

$$\texttt{"cocoa-powder"} \in \texttt{<flavor>} \qquad \texttt{"chocolate-syrup"} \in \texttt{<flavor>}$$

Assume that it is allowed to add whitespaces among terminals freely.

a) Which of the following are elements of `<coffee>`?
   i. `caramel latte macchiato`
   ii. `espresso`
   iii. `steamed-milk on caramel on milk-foam on espresso`
   iv. `chocolate-syrup on cocoa-powder on cinnamon on milk-foam on steamed-milk on espresso`
   v. `steamed-milk on espresso on chocolate-syrup`

b) Explain whether the following is `<coffee>` or not:
   `cocoa-powder on milk-foam on steamed-milk on espresso`

3. Consider the following concrete syntax:

```
<ice-cream> ::= "sprinkles" "on" <ice-cream>
              | "cherry" "on" <ice-cream>
              | "scoop" "of" <flavor> "on" <ice-cream>
              | "sugar-cone"
              | "waffle-cone"
<flavor>    ::= "vanilla"
              | "lettuce"
```

Assume that it is allowed to add whitespaces among terminals freely.

a) Which of the following are elements of `<ice-cream>`?
   i. `sprinkles`
   ii. `sugar-cone`
   iii. `vanilla`
   iv. `scoop of vanilla on waffle-cone`
   v. `sprinkles on lettuce on waffle-cone`
   vi. `scoop of vanilla on sprinkles on waffle-cone`

b) Explain why the following is an element of `<ice-cream>`:
   `cherry on scoop of lettuce on scoop of vanilla on sugar-cone`

7 Identifiers

Variables are one of the basic concepts of programming languages. A *variable* relates a name to a value. We use the value of a variable by writing the name of the variable. For example, the following Scala program prints `3`.

```
val x = 3
println(x)
```

The program defines a variable whose name is `x` and value is `3`. At the second line, the name `x` denotes the value `3`.

We call the names of variables identifiers. An *identifier* is a name related to a certain entity in a program. Not only the names of variables are identifiers; there are various kinds of identifiers:

- Function names, which are related to functions
- Parameter names, which are related to the values of arguments
- Field names, which are related to values of fields
- Method names, which are related to methods
- Class names, which are related to classes

This chapter introduces identifiers. Identifiers in programs can split into three groups: binding occurrences, bound occurrences, and free identifiers. We will see what they are. This chapter discusses identifiers based on the use of variables in programs. We will define VAE by extending AE of Chapter 6 with variables. Variables of VAE are immutable. We will deal with mutable variables in Chapter 12. In VAE, the names of variables are the only identifiers. However, as you have seen already, real-world programming languages have many kinds of identifiers.

7.1 Identifiers

Identifiers name entities like variables and functions. Let us discuss notions related to identifiers with the following Scala program:

```
f(0)
def f(x: Int): Int = {
  val y = 2
  x + y
}
f(1)
x - z
```

In this program, `f`, `x`, `y`, and `z` are identifiers. Strictly speaking, `Int` also is an identifier, but we ignore it because we do not want to take types into account here.

A single identifier can occur multiple times in a program. For instance, `f` occurs three times in the program: line 1, line 2, and line 6. We can classify occurrences of identifiers into three categories: binding occurrences, bound occurrences, and free identifiers.

An occurrence of an identifier is called a *binding occurrence* if the identifier occurs to be defined. A binding occurrence relates the identifier to a particular entity. The program has three binding occurrences:

- `f` at line 2
  It relates `f` to a function.
- `x` at line 2
  It relates `x` to the value of an argument given to `f`.
- `y` at line 3
  It relates `y` to the value `2`.

Every binding occurrence has its own scope. The *scope* of a binding occurrence means a code region where the indentifier defined by the binding occurrence is alive, i.e. usable. The scope of each identifier in the program is as follows:

- `f`
  A function can be used in its body (as Scala allows recursive function definitions) and at the lines below its definition. The scope of `f` is from line 3 to line 7.
- `x`
  A paramter of a function can be used only in the function body. The scope of `x` is line 3 and line 4.
- `y`
  A variable can be used at the lines below its definition. The scope of `y` is line 4.

An occurrence of an identifier is called a *bound occurrence* if the identifier occurs to use the entity related to itself. Since an identifier becomes related to an entity by its binding occurrence, any bound occurrences must reside in the scope of the binding occurrence. The program has three bound occurrences:

- `f` at line 6
  It denotes the function defined at line 2.
- `x` at line 4
  It denotes the value of an argument passed to `f`.
- `y` at line 4
  It denotes the value `2`.

An occurrence of an identifier is called a *free identifier* if it is neither binding nor bound. A free identifier neither introduces a new name nor uses a name defined already. It is not in the scope of any binding occurrence of the same identifier. The program has three free identifiers:

- `f` at line 1
  It is outside the scope of `f`.
- `x` at line 7
  It is outside the scope of `x`.
- `z` at line 7
  The program never defines `z`.

We call a free identifier a *free variable* when it is the name of a variable. Therefore, both `x` and `z` at line 7 are free variables.

Now, consider a binding occurrence that resides in the scope of a binding occurrence of the same identifier. For example, the following program has two binding occurrences of `x`, and the second binding occurrence is in the scope of the first binding occurrence.

```
def f(x: Int): Int = {
  def g(x: Int): Int =
    x
  g(x)
}
```

In this case, shadowing happens. *Shadowing* means that the innermost binding occurrence **shadow**s, i.e. temporarily invalidates, the outer binding occurrences of the same name. Therefore, `x` at line 2 shadows `x` at line 1. `x` at line 3 belongs to the scope of both binding occurrences simultaenously. It denotes the value of an argument given to `g`, not `f`, because of shadowing. On the other hand, `x` at line 4 denotes the value of an argument given to `f` since it belongs to the scope of only `x` at line 1.

7.2 Syntax

Let us define the abstract syntax of VAE. We do not consider concrete syntax anymore. Therefore, the term syntax will be used to mean abstract syntax. Also, from now on, we use the term *expressions* rather than programs when we discuss languages like VAE. For example, we say that $1 + 2$ is an expression of AE, and 1 and 2 are the subexpressions of $1 + 2$.

Recall the example at the beginning of the chapter:

```
val x = 3
println(x)
```

To add variables to AE, we need two kinds of expressions. The first kind is expressions defining a variable, i.e. binding an identifier. In the example, `val x = 3; println(x)` is such an expression. It defines the variable `x` and starts the scope of `x` so that `x` can be used in `println(x)`. We can conclude that an expression defining a variable consists of three parts: the name of the variable, an expression determining the value of the variable, and an expression that can use the variable. These parts are `x`, `3`, and `println(x)`, respectively, in the example. The second kind is expressions using a variable, i.e. a bound occurrence. In the example, `x` at the second line is such an expression. It uses the variable `x` to denote the value `3`. Based on this observation, we can define the syntax of VAE.

First, we need to add a new syntactic element: identifiers. The metavariable x ranges over identifiers. Let *Id* be the set of every identifier.

$$x \in Id$$

We do not care what *Id* really is.

The syntax of VAE is as follows:[1]

1: We omit the common part to AE.

$$e ::= \cdots \mid \mathsf{val}\ x\mathsf{=}e\ \mathsf{in}\ e \mid x$$

- $\mathsf{val}\ x\mathsf{=}e_1\ \mathsf{in}\ e_2$
  It defines a new variable whose name is x. Therefore, the occurrence of x is a binding occurrence. e_1 decides the value denoted by the variable. The scope of the variable includes e_2 but excludes e_1.
- x
  It uses a variable; it is either a bound occurrence of x or a free identifier. If it belongs to the scope of a binding occurrence of the same name, then it is a bound occurrence and denotes the value associated with the identifier. Otherwise, it is a free identifier, which denotes nothing.

7.3 Semantics

To define the semantics of VAE, we need an additional semantic element that stores the values denoted by variables. Without such an element, we cannot know the value of each variable. We call the element an *environment*. An environment is a finite partial function.[2] The metavariable σ ranges over environments.

2: A finite partial function is a partial function whose domain is a finite set.

$$Env = Id \xrightarrow{\text{fin}} \mathbb{Z}$$

$$\sigma \in Env$$

For example, consider an environment σ. If $\sigma(\mathsf{x}) = 1$, the value of a variable named x is 1. An environment is a partial function because it does not have the values related to free identifiers. If a variable named y is free in σ, then $\sigma(\mathsf{y})$ is undefined. In addition, it is finite since every program defines only finitely many identifiers.

Every expression in VAE can evaluate to an integer only under some environment. The reason is obvious: without environments, there is no way to find the values of variables, and thus environments are essential to evaluation.

The following rule defines the semantics of x:

Rule Id
If x is in the domain of σ,
then x evaluates to $\sigma(x)$ under σ.

If x is an element of the domain of σ, x is a bound occurrence. The environment gives us the value denoted by x, which is $\sigma(x)$. Then, the result is $\sigma(x)$. Otherwise, x is not in the domain and is a free identifier. In that case, we cannot evaluate x. The evaluation terminates immediately. It can be interpreted as a run-time error.

Formally, the semantics of VAE is a ternary relation over Env, E, and $\mathbb{Z}$ since it must take environments into account.

$$\Rightarrow \subseteq Env \times E \times \mathbb{Z}$$

$(\sigma, e, n) \in \Rightarrow$ is true if and only if e evaluates to n under σ. We write $\sigma \vdash e \Rightarrow n$ instead of $(\sigma, e, n) \in \Rightarrow$. Intuitively, σ and e are inputs, and n is the corresponding output.

Rule Id can be formulated as the following inference rule: [3]

3: $Domain(\sigma)$ denotes the domain of σ.

$$\frac{x \in Domain(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)} \quad \text{[Id]}$$

When a variable is defined, the value of the variable is added to the environment. We write $\sigma[x \mapsto n]$ to denote an environment obtained by adding the fact that x denotes v to σ. Then, the following property holds:

$$\sigma[x \mapsto n](x') = \begin{cases} n & \text{if } x = x' \\ \sigma(x') & \text{if } x \neq x' \end{cases}$$

The following rule defines the semantics of val x=e_1 in e_2:

Rule Val
If e_1 evaluates to n_1 under σ, and e_2 evaluates to n_2 under $\sigma[x \mapsto n_1]$, then val x=e_1 in e_2 evaluates to n_2 under σ.

To evaluate val x=e_1 in e_2, we need to determine the value of x first. It can be done by evaluating e_1. Since the scope of x excludes e_1, the evaluation proceeds under σ, which is a given environment. The result of e_1 is the value of x, and this information must be added to the environment. By adding the fact to σ, we get $\sigma[x \mapsto n_1]$. As e_2 is the scope of x, e_2 is evaluated under $\sigma[x \mapsto n_1]$. The result of e_2 is the final result.

This semantics naturally explains shadowing. Let x already be in the domain of σ. Suppose that $\sigma(x) = n$. However, e_2 is evaluated under $\sigma[x \mapsto n_1]$, and $\sigma[x \mapsto n_1](x) = n_1$. When x is used in e_2, its value is n_1, not n. Therefore, we can say that the innermost definition of x is used for the evaluation of e_2. This behavior exactly matches the concept of shadowing explained before.

Rule Val can be expressed as the following inference rule:

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma[x \mapsto n_1] \vdash e_2 \Rightarrow n_2}{\sigma \vdash \text{val } x\text{=}e_1 \text{ in } e_2 \Rightarrow n_2} \quad \text{[Val]}$$

The remaining cases are n, $e_1 + e_2$, and $e_1 - e_2$. Rules for those cases are basically identical to the rules of AE. However, we need to additionally take environments into account.

Rule Num
n evaluates to n under σ.

Rule Add
If e_1 evaluates to n_1 under σ, and e_2 evaluates to n_2 under σ,
then $e_1 + e_2$ evaluates to $n_1 + n_2$ under σ.

Rule Sub
If e_1 evaluates to n_1 under σ, and e_2 evaluates to n_2 under σ,
then $e_1 - e_2$ evaluates to $n_1 - n_2$ under σ.

Integers, addition, and subtraction never update environments. An integer evaluates to itself. Addition and subtraction evaluates their subexpressions under the same environment.

We can express the rules in a natural language as the following inference rules:

$$\sigma \vdash n \Rightarrow n \quad [\textsc{Num}]$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad [\textsc{Add}]$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 - e_2 \Rightarrow n_1 - n_2} \quad [\textsc{Sub}]$$

The following proof tree proves that val x=1 in x + x evaluates to 2 under the empty environment. Note that $[x_1 \mapsto n_1, \ldots, x_m \mapsto n_m]$ denotes an environment whose domain includes from x_1 to x_m and each x_i is mapped to n_i.

$$\dfrac{\emptyset \vdash 1 \Rightarrow 1 \qquad \dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}([\mathsf{x} \mapsto 1])}{[\mathsf{x} \mapsto 1] \vdash \mathsf{x} \Rightarrow 1} \qquad \dfrac{\mathsf{x} \in \mathit{Domain}([\mathsf{x} \mapsto 1])}{[\mathsf{x} \mapsto 1] \vdash \mathsf{x} \Rightarrow 1}}{[\mathsf{x} \mapsto 1] \vdash \mathsf{x} + \mathsf{x} \Rightarrow 2}}{\emptyset \vdash \mathsf{val\ x{=}1\ in\ x + x} \Rightarrow 2}$$

7.4 Interpreter

The following Scala code implements the abstract syntax of VAE:

```
sealed trait Expr
case class Num(n: Int) extends Expr
case class Add(l: Expr, r: Expr) extends Expr
case class Sub(l: Expr, r: Expr) extends Expr
case class Val(x: String, i: Expr, b: Expr) extends Expr
case class Id(x: String) extends Expr
```

An identifier is an arbitrary string. `Val(`x`,` e_1`,` e_2`)` corresponds to val x=e_1 in e_2; `Id(`x`)` corresponds to x.

We use a map to represent an environment. The type of an environment is `Map[String, Int]`.

```
type Env = Map[String, Int]
```

We can add a pair of a key and a value to a map with the `+` operator. For example, where `m` is `Map(1 -> "one")`, `m + (2 -> "two")` is the same as `Map(1 -> "one", 2 -> "two")`.

```
def interp(e: Expr, env: Env): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env) + interp(r, env)
  case Sub(l, r) => interp(l, env) - interp(r, env)
  case Val(x, i, b) => interp(b, env + (x -> interp(i, env)))
  case Id(x) => env(x)
}
```

Since the structure of the code is almost identical to the semantics rules, there is nothing much to explain. In the `Id` case, when `x` is a key in `env`, the corresponding value becomes the result of `interp`. Otherwise, an exception is thrown, and the execution terminates without producing any results.

7.5 Exercises

1. Which of the following are examples of shadowing?
   a) val x=(val x=3 in $5 - x$) in $1 + x$
   b) val x=3 in val y=5 in $1 + x$
   c) val x=3 in val x=5 in $1 + x$
2. For each of the following expression:
   - val x=(val x=3 in $5 - x$) in $1 + x$
   - val x=3 in val y=5 in $1 + x$
   - val x=3 in val x=5 in $1 + x$

   a) Draw arrows from each bound occurrence to its binding occurrence.
   b) Draw dotted arrows from each shadowing variable to its shadowed variable.

First-Order Functions 8

A function is one of the most important concepts in programming languages. It is the key feature of functional languages, as the term functional implies. Even in imperative languages, functions are important and widely-used. This chapter focuses on first-order functions. *First-order functions* are functions that cannot take or return functions. They are much restrictive than first-class functions but still very useful.

Consider the following Scala program:

```
def twice(x: Int): Int = x + x

println(twice(3) + twice(5))
```

It defines a function, `twice`. The function takes one argument and returns twice of the argument. The program can call the function whenever we want. `twice(3)` passes `3` as an argument to `twice`. Its result is `6`, which is twice of `3`. Similarly, `twice(5)` results in `10`. Therefore, the program prints `16`.

This chapter defines F1VAE by adding first-order functions to VAE. Every function in F1VAE is top-level. It means that a function definition cannot be a part of an expression. We assume that a F1VAE program is a single expression that is evaluated under an environment and a list of function definitions. This design prevents us from exploring interesting topics like closures but enables us to focus on the semantics of function calls. The next chapter will introduce first-class functions and closures, which make functions more expressive.

8.1 Syntax

We can figure out the components of a function definition from the above example. If we ignore the type annotations, the definition consists of three parts: `twice`, `x`, and `x + x`. `twice` is the name of the function; `x` is the parameter of the function; `x + x` is the body of the function. Therefore, we can define the syntax of a function definition as follows:

$$d \ ::= \ \mathsf{def}\ x(x)\mathsf{=}e$$

The metavariable d ranges over function definitions. Let *FunDef* denote the set of every function definition. A function definition $\mathsf{def}\ x_1(x_2)\mathsf{=}e$ defines a function whose name is x_1, parameter is x_2, and body is e. Both x_1 and x_2 are binding occurrences. The scope of x_1 is the entire program; the scope of x_2 is e. In many real-world languages, a function has zero or more parameters. However, our syntax restricts a function to have one and only one parameter. We adopt this design to make the semantics

simple. Once you understand a function with a single parameter, you can easily extend the concept to a function with multiple parameters.

Using a function is to call the function. If we never call a function, the function is useless. We need to add a new kind of expression to the language: the function call expression. The following is the syntax of F1VAE: [1]

1: We omit the common part to VAE.

$$e ::= \cdots \mid x(e)$$

$x(e)$ is the function call expression. It calls a function named x. e determines the value of the argument of the call. Here, x is a bound occurrence. A function call always has only one argument since every function in F1VAE has only one parameter.

8.2 Semantics

Like that we have introduced environments to store the values of variables, we need a new semantic element that associates functions with their names. Let us call it a function environment, which is a finite partial function from identifiers to function definitions.

$$FEnv = Id \xrightarrow{\text{fin}} FunDef$$

$$\Lambda \in FEnv$$

The metavariable Λ ranges over function environments.

Evaluation of an expression requires not only an environment but also a function environment to handle function calls properly. Therefore, the semantics is a relation over Env, $FEnv$, E, and $\mathbb{Z}$.

$$\Rightarrow \subseteq Env \times FEnv \times E \times \mathbb{Z}$$

$(\sigma, \Lambda, e, n) \in \Rightarrow$ is true if and only if e evalautes to n under σ and Λ. We write $\sigma, \Lambda \vdash e \Rightarrow n$ instead of $(\sigma, \Lambda, e, n) \in \Rightarrow$.

The following rule describes the semantics of function calls:

Rule Call
If
- e evaluates to n' under σ and Λ,
- x is in the domain of Λ,
- $\Lambda(x)$ is def $x(x')$=e', and
- e' evaluates to n under $[x' \mapsto n']$ and Λ,

then
- $x(e)$ evaluates to n under σ and Λ.

To evaluate $x(e)$, we need to evalaute e first to decide the value of the argument. Then, we search for a function from the function environment with a given function name, x. x must be in the domain of the function environment. Otherwise, x is a free identifier, and a run-time error

happens. When x is in the domain, we can get the corresponding function definition. The function definition gives us the name of the parameter and the body. Since every function is top-level, the body of each function does not belong to the scope of any local variables. It can use no more than its own parameter. Thus, the body should be evaluated under $[x' \mapsto n']$, not $\sigma[x' \mapsto n']$. At the same time, since function calls do not affect function environments, the same function environment is used for the evaluation of the body. The result of the body is the result of the function call.

We can formulate the semantics as the following inference rule:

$$\frac{x \in \mathit{Domain}(\Lambda) \qquad \Lambda(x) = \mathsf{def}\ x(x')\text{=}e' \qquad \begin{array}{c}\sigma, \Lambda \vdash e \Rightarrow n' \\ [x' \mapsto n'], \Lambda \vdash e' \Rightarrow n\end{array}}{\sigma, \Lambda \vdash x(e) \Rightarrow n} \quad [\textsc{Call}]$$

The other rules should be revised to consider function environments. No expression modifies a function environment.

Rule Num
n evaluates to n under σ and Λ.

Rule Add
If e_1 evaluates to n_1 under σ and Λ, and e_2 evaluates to n_2 under σ and Λ, then $e_1 + e_2$ evaluates to $n_1 + n_2$ under σ and Λ.

Rule Sub
If e_1 evaluates to n_1 under σ and Λ, and e_2 evaluates to n_2 under σ and Λ, then $e_1 - e_2$ evaluates to $n_1 - n_2$ under σ and Λ.

Rule Val
If e_1 evaluates to n_1 under σ and Λ, and e_2 evaluates to n_2 under $\sigma[x \mapsto n_1]$ and Λ,
then $\mathsf{val}\ x\text{=}e_1\ \mathsf{in}\ e_2$ evaluates to n_2 under σ and Λ.

Rule Id
If x is in the domain of σ,
then x evaluates to $\sigma(x)$ under σ and Λ.

We can fix the inference rules in a similar way.

$$\sigma, \Lambda \vdash n \Rightarrow n \quad [\textsc{Num}]$$

$$\frac{\sigma, \Lambda \vdash e_1 \Rightarrow n_1 \qquad \sigma, \Lambda \vdash e_2 \Rightarrow n_2}{\sigma, \Lambda \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad [\textsc{Add}]$$

$$\frac{\sigma, \Lambda \vdash e_1 \Rightarrow n_1 \qquad \sigma, \Lambda \vdash e_2 \Rightarrow n_2}{\sigma, \Lambda \vdash e_1 - e_2 \Rightarrow n_1 - n_2} \quad [\textsc{Sub}]$$

$$\frac{\sigma, \Lambda \vdash e_1 \Rightarrow n_1 \qquad \sigma[x \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2}{\sigma, \Lambda \vdash \mathsf{val}\ x{=}e_1\ \mathsf{in}\ e_2 \Rightarrow n_2} \quad [\textsc{Val}]$$

$$\frac{x \in \mathit{Domain}(\sigma)}{\sigma, \Lambda \vdash x \Rightarrow \sigma(x)} \quad [\textsc{Id}]$$

8.3 Interpreter

The following Scala code expresses the abstract syntax of F1VAE: [2]

2: We omit the common part to VAE.

```
case class FunDef(f: String, x: String, b: Expr)

sealed trait Expr
...
case class Call(f: String, a: Expr) extends Expr
```

Just like environments, function environments can be expressed as maps. The type of a function environment is `Map[String, FunDef]` as it maps an identifier to a function definition.

```
type FEnv = Map[String, FunDef]
```

The following function evaluates a given expression under a given environment and a given function environment.

```
def interp(e: Expr, env: Env, fEnv: FEnv): Int = e match {
  case Num(n) => n
  case Add(l, r) =>
    interp(l, env, fEnv) + interp(r, env, fEnv)
  case Sub(l, r) =>
    interp(l, env, fEnv) - interp(r, env, fEnv)
  case Val(x, i, b) =>
    interp(b, env + (x -> interp(i, env, fEnv)), fEnv)
  case Id(x) => env(x)
  case Call(f, a) =>
    val FunDef(_, x, e) = fEnv(f)
    interp(e, Map(x -> interp(a, env, fEnv)), fEnv)
}
```

The implementation reflects the semantics exactly. You can easily check its correctness with the case-wise comparison.

8.4 Scope

The current semantics is called static scope. Static scope allows the scope of a binding occurrence to be determined statically, i.e. only by looking the code, without executing it. In other words, a function body can use only variables that have been defined already when the function is defined. For example, consider the following code:

```
def f(x)=x + y
```

Since every function is top-level, while variables are local in F1VAE, `y` does not belong to the scope of any binding occurrence of `y`. No variable can be defined before the function. Therefore, `y` is a free variable, and calling the function `f` must incur a run-time error. It is true under the current semantics. The current function call semantics evaluates the body of a function under the environment that has only the value of the parameter. The environments at function call-sites never affect the environment used for the evaluation of the body.

The opposite of static scope is dynamic scope, which makes every information in the environment at each call-site available to the function body. The behavior of a function depends on not only its argument but also its call-site. An identifier in the body of a function becomes associated with a different entity for each function call. It implies that the scope of a binding identifier cannot be determined statically; it is determined at run time, i.e. dynamically.

For example, the following expression evaluates to 3 when we assume the same definition of `f` as before:

```
(val y=1 in f(0)) + (val y=2 in f(0))
```

During the first function call, `y` in `f` is bound to the first `y` and denotes 1. However, during the second function call, it is bound to the second one and denotes 2. The scope of the first `y` includes not only `f(0)`, which is normal, but also the body of `f`. It is the same for the second `y`. As you can see, under dynamic scope, the scope of a binding identifier is not fixed; it becomes extended at run time due to function calls.

To adopt dynamic scope to F1VAE, we need to change the function call semantics as follows:

Rule Call-Dyn
If
- e evaluates to n' under σ and Λ,
- x is in the domain of Λ,
- $\Lambda(x)$ is def $x(x')$=e', and
- e' evaluates to n under $\sigma[x' \mapsto n']$ and Λ,

then
- $x(e)$ evaluates to n under σ and Λ.

It is equivalent to the following inference rule:

$$\frac{x \in \mathit{Domain}(\Lambda) \qquad \Lambda(x) = \mathsf{def}\ x(x')=e' \qquad \sigma, \Lambda \vdash e \Rightarrow n' \qquad \sigma[x' \mapsto n'], \Lambda \vdash e' \Rightarrow n}{\sigma, \Lambda \vdash x(e) \Rightarrow n} \text{ [Call-Dyn]}$$

The interpreter can be fixed like below.

```
case Call(f, a) =>
```

```
val (x, e) = fEnv(f)
interp(e, env + (x -> interp(a, env, fEnv)), fEnv)
```

Dynamic scope prevents programs from being modular. The environment at each call-site affects the behavior of a function. It hinders programmers from reasoning about the semantics of a function based on the definition. They need to additionally consider every possible call-site. It implies that different parts of a program unexpectedly interfere with each other. Therefore, dynamic scope makes programs error-prone. Because of the harmfulness of dynamic scope, most modern languages adopt static scope.

8.5 Exercises

1. With the following list of function definitions in F1VAE:
   ```
   def twice(x)=x + x
   def x(y)=y
   def f(x)=x + 1
   def g(g)=g
   ```
   Show the results of evaluating the following expressions under the empty environment. When it is an error, describe which error it is.
   a) `twice(twice)`
   b) `val x=5 in x(x)`
   c) `g(3)`
   d) `g(f)`
   e) `g(g)`

First-Class Functions 9

First-class functions are functions that can be used as values. They are much more expressive than first-order functions, which are the topic of the previous chapter. This chapter explains the semantics of first-class functions. We need to introduce the notion of a closure to make first-class functions work properly. We will see what closures are and why they are necessary.

This chapter defines FVAE by extending VAE with first-class functions. The only way to create a function in FVAE is to make an *anonymous function*, which is a function without a name. However, we can add named functions as syntactic sugar. In addition, we will see that even variable definitions can be considered as syntactic sugar.

9.1 Syntax

Consider an anonymous function in Scala:

```
(x: Int) => x + x
```

If we ignore its type annotation, it consists of two parts: `x` and `x + x`. `x` is the parameter of the function; `x + x` is the body of the function. This observation lets us know that an anonymous function consists of its name and parameter.

In F1VAE, the sytnax of a function call is $x(e)$. To call a function, the name of the function should be given. However, it is not true in languages with first-class functions. Let us see some function calls in Scala.

```
def twice(x: Int): Int = x + x
twice(1)
```

`twice(1)` is a function call, and it designates a function by its name.

```
def makeAdder(x: Int): Int => Int =
  (y: Int) => x + y
makeAdder(3)(5)
```

`makeAdder` is a function that returns a function. `makeAdder(3)` is a function call, and its result is a function. Therefore, we can call the resulting function again. `makeAdder(3)(5)` is an expression that calls `makeAdder(3)`. It designates a function by an expression, rather than just a name. We can conclude that the syntax of a function call in FVAE should be more general than F1VAE because of the presence of first-class functions. In FVAE, a function call consists of two expressions: one

determines the function to be called and the other determines the value of the argument.

We have used the term function call so far. In the context of functional programming, we use the term *function application* more frequently. When we see `f(1)`, we say "`f` is applied to `1`" instead of "`f` is called with the argument `1`." Applications sound more natural than calls especially when we are talking about first-class functions. For example, we usually say "`makeAdder(3)` is applied to `5`" rather than "`makeAdder(3)` is called with the argument `5`."

From the above observation on anonymous functions and function applications, we can define the syntax of FVAE. The following is the syntax of FVAE: [1]

1: We omit the common part to VAE.

$$e ::= \cdots \mid \lambda x.e \mid e\ e$$

- $\lambda x.e$
  It is called an anonymous function or a *lambda abstraction*. It denotes a function whose parameter is x and body is e. x is a binding occurrence, and its scope is e. A function has zero or more parameters in many real-world languages, but we restrict a function in FVAE to have one and only one parameter for simplicity as before.
- $e_1\ e_2$
  It is a function application, or just an application in short. e_1 denotes the function; e_2 denotes the argument.

9.2 Semantics

Integers are the only values in VAE. It is not true in FVAE. Since first-class functions are values, a value of FVAE is either an integer or a function. Thus, we define a new kind of semantic element, *value*. The metavariable v ranges over values. Also, let V be the set of every value.

$$v ::= n \mid \langle \lambda x.e, \sigma \rangle$$

A value is either an integer or a closure. A *closure*, which is a function as a value, has the form $\langle \lambda x.e, \sigma \rangle$. It is a pair of a lambda abstraction and an environment. A lambda abstraction in a closure may have free identifiers, but the environment of the closure can store the values denoted by the free identifiers.

To discuss the necessity of closures, consider the following expression:

$$(\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x} + \mathsf{y})\ 1\ 2$$

It is equivalent to `((x: Int) => (y: Int) => x + y)(1)(2)` in Scala. The function $\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x} + \mathsf{y}$ does not have any free identifiers. The scope of `x` is $\lambda \mathsf{y}.\mathsf{x} + \mathsf{y}$; the scope of `y` is `x + y`. Therefore, both `x` and `y` in `x + y` are bound occurrences. The whole expression must result in 3, which equals 1 + 2, without a run-time error. You can check it by running the equivalent Scala program.

If we consider a function value as just a lambda abstraction, not a closure, evaluation of the above expression becomes problematic. When the expression is evaluated, λx.λy.x + y is applied to 1 first. The result is a function value, which is a lambda abstraction: λy.x + y. Next, λy.x + y is applied to 2. The result of the application can be computed by evaluating x + y under the environment containing that y denotes 2. However, there is no way to find the value of x. x has become free identifier although it was not in the beginning.

We adopt the notion of a closure to resolve the problem. When a lambda expression evaluates to a function value, which is a closure, it captures the environment. Since λy.x + y is evaluated under the environment containing that x denotes 1, its result is $\langle \lambda \mathsf{y.x} + \mathsf{y}, [\mathsf{x} \mapsto 1] \rangle$. The captured environment of the closure records that x is not a free identifier and denotes 1. When the closure is applied to 2, its body x + y is evaluated under $[\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2]$, not $[\mathsf{y} \mapsto 2]$. The addition successfully results in 3.

In summary, we need closures to retain the static scope semantics. A first-class function can be passed as a value and thus applied to an argument at a different place from where it has been defined. However, the environments used for the evaluation of their bodies must be determined statically. In other words, the denotation of identifiers in the bodies of functions must be determined when the functions are defined, not used. Therefore, each closure captures the surrounding environment when it is created.

Now, let us define the semantics of FVAE. Most things are the same as the semantics of VAE, but we should be aware of that values now include not only integers but also closures.

An environment is a finite partial function from identifiers to values.

$$Env = Id \xrightarrow{\text{fin}} V$$

The semantics of FVAE is a ternay relation over Env, E, and V.

$$\Rightarrow \subseteq Env \times E \times V$$

$\sigma \vdash e \Rightarrow v$ is true if and only if e evaluates to v under σ.

A lambda abstraction creates a closure containing the current environment.

Rule Fun
$\lambda x.e$ evaluates to $\langle \lambda x.e, \sigma \rangle$ under σ.

$$\sigma \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \sigma \rangle \quad [\text{Fun}]$$

A function application evaluates its both subexpressions. Then, it evaluates the body of the closure under the environment obtained by adding the value of the argument to the environment of the closure.

Rule App

If
e_1 evaluates to $\langle \lambda x.e, \sigma' \rangle$ under σ,
e_2 evaluates to v' under σ, and
e evaluates to v under $\sigma'[x \mapsto v']$,
then
$e_1\ e_2$ evaluates to v under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle \qquad \sigma \vdash e_2 \Rightarrow v' \qquad \sigma'[x \mapsto v'] \vdash e \Rightarrow v}{\sigma \vdash e_1\ e_2 \Rightarrow v} \quad [\textsc{App}]$$

We can reuse Rule Num, Rule Add, Rule Sub, and Rule Id of VAE. However, it is important to note that FVAE has more cases that evaluation can fail than VAE. For example, consider Rule Add.

Rule Add
If e_1 evaluates to n_1 under σ, and e_2 evaluates to n_2 under σ,
then $e_1 + e_2$ evaluates to $n_1 + n_2$ under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad [\textsc{Add}]$$

The rule assumes the results of e_1 and e_2 to be integers. If the assumption is violated, a run-time error happens. For example, $(\lambda \mathsf{x}.\mathsf{x}) + 1$ incurs a run-time error becuase the left operand is a closure, not an integer.

We need to revise Rule Val of VAE a bit. Since every value is an integer in VAE, a variable of VAE can denote only an integer. In FVAE, a variable should be able to denote a general value, not only an integer.

Rule Val
If e_1 evaluates to v_1 under σ, and e_2 evaluates to v_2 under $\sigma[x \mapsto v_1]$,
then $\mathsf{val}\ x{=}e_1\ \mathsf{in}\ e_2$ evaluates to v_2 under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \mathsf{val}\ x{=}e_1\ \mathsf{in}\ e_2 \Rightarrow v_2} \quad [\textsc{Val}]$$

Now, a variable can denote a value, not only an integer.

The following proof trees prove that $(\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}+\mathsf{y})\ 1\ 2$ evaluates to 3 under the empty environment. The proof splits into three trees for readability. Suppose that $\sigma_1 = [\mathsf{x} \mapsto 1]$ and $\sigma_2 = [\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2]$.

$$\frac{\emptyset \vdash \lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}+\mathsf{y} \Rightarrow \langle \lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}+\mathsf{y}, \emptyset \rangle \qquad \emptyset \vdash 1 \Rightarrow 1 \qquad \sigma_1 \vdash \lambda \mathsf{y}.\mathsf{x}+\mathsf{y} \Rightarrow \langle \lambda \mathsf{y}.\mathsf{x}+\mathsf{y}, \sigma_1 \rangle}{\emptyset \vdash (\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}+\mathsf{y})\ 1 \Rightarrow \langle \lambda \mathsf{y}.\mathsf{x}+\mathsf{y}, \sigma_1 \rangle}$$

$$\frac{\dfrac{\mathsf{x} \in \mathit{Domain}(\sigma_2)}{\sigma_2 \vdash \mathsf{x} \Rightarrow 1} \qquad \dfrac{\mathsf{y} \in \mathit{Domain}(\sigma_2)}{\sigma_2 \vdash \mathsf{y} \Rightarrow 2}}{\sigma_2 \vdash \mathsf{x}+\mathsf{y} \Rightarrow 3}$$

$$\frac{\emptyset \vdash (\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}+\mathsf{y})\,1 \Rightarrow \langle \lambda \mathsf{y}.\mathsf{x}+\mathsf{y}, \sigma_1 \rangle \quad \emptyset \vdash 2 \Rightarrow 2 \quad \sigma_2 \vdash \mathsf{x}+\mathsf{y} \Rightarrow 3}{\emptyset \vdash (\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}+\mathsf{y})\,1\,2 \Rightarrow 3}$$

9.3 Interpreter

The following Scala code implements the syntax of FVAE: [2]

2: We omit the common part to VAE.

```
sealed trait Expr
...
case class Fun(x: String, b: Expr) extends Expr
case class App(f: Expr, a: Expr) extends Expr
```

`Fun(`x`, `e`)` represents $\lambda x.e$; `App(`e_1`, `e_2`)` represents $e_1\ e_2$.

A value of FVAE is either an integer or a closure. Thus, we represent a value as an ADT.

```
sealed trait Value
case class NumV(n: Int) extends Value
case class CloV(p: String, b: Expr, e: Env) extends Value
```

`NumV(`n`)` represents n; `CloV(`x`, `e`, `σ`)` represents $\langle \lambda x.e, \sigma \rangle$.

An environment is a finite partial function from identifiers to values. Therefore, the type of an environment is `Map[String, Value]`.

```
type Env = Map[String, Value]
```

The following function evaluates a given expression under a given environment:

```
def interp(e: Expr, env: Env): Value = e match {
  case Num(n) => NumV(n)
  case Add(l, r) =>
    val NumV(n) = interp(l, env)
    val NumV(m) = interp(r, env)
    NumV(n + m)
  case Sub(l, r) =>
    val NumV(n) = interp(l, env)
    val NumV(m) = interp(r, env)
    NumV(n - m)
  case Id(x) => env(x)
  case Fun(x, b) => CloV(x, b, env)
  case App(f, a) =>
    val CloV(x, b, fEnv) = interp(f, env)
    interp(b, fEnv + (x -> interp(a, env)))
}
```

In the `Num` case, the return value is `NumV(n)`, not `n`, since the function must return a value of the type `Value`.

In the `Add` and `Sub` cases, we cannot assume that the operands are integers any longer. We use pattern matching to discriminate integers from closures. If both operands are integers, addition or subtraction succeeds. Otherwise, at least one of them is a closure, and the interpreter crashes due to a pattern matching failure. Note that this code is equivalent to the following code:

```
case Add(l, r) =>
  interp(l, env) match {
    case NumV(n) => interp(r, env) match {
      case NumV(m) => NumV(n + m)
      case _ => error("not an integer")
    }
    case _ => error("not an integer")
  }
```

Similarly, in the `App` case, we use pattern matching to discriminate closures from integers. The first expression of `App` must yield a clsoure, not an integer, to make the execution succeed.

9.4 Syntactic Sugar

We can add named local functions to FVAE with the following change in the syntax:

$$e \ ::= \ \cdots \mid \textsf{def}\ x(x)\textsf{=}e\ \textsf{in}\ e$$

$\textsf{def}\ x_1(x_2)\textsf{=}e_1\ \textsf{in}\ e_2$ defines a function whose name is x_1, parameter is x_2, and body is e_1. The scope of x_1 is e_2, and thus the function does not allow recursion.

Instead of changing the semantics, FVAE can provide named local functions as syntactic sugar. Let s be a string transformed into $\textsf{def}\ x_1(x_2)\textsf{=}e_1\ \textsf{in}\ e_2$ by the parser of FVAE with named local functions embeded in the semantics. To treat named local functions as syntactic sugar, the parser should transform s into $\textsf{val}\ x_1\textsf{=}\lambda x_2.e_1\ \textsf{in}\ e_2$.

Variable definitions can be considered as syntactic sugar as well. Let s be a string transformed into $\textsf{val}\ x\textsf{=}e_1\ \textsf{in}\ e_2$. To make variable definitions syntactic sugar, the parser can transform s into $(\lambda x.e_2)\ e_1$. The evaluation of $(\lambda x.e_2)\ e_1$ evaluates e_1 first. Then, e_2 is evaluated under the environment that x denotes the result of e_1. This semantics is exactly the same as that of $\textsf{val}\ x\textsf{=}e_1\ \textsf{in}\ e_2$. Therefore, we can say that variable definitions are just syntactic sugar in FVAE.

Hereafter, we remove variable definitions from FVAE and call the language FAE. However, we may still use variable definitions in examples. It is completely fine because they are considered as syntactic sugar.

Furthermore, we can treat even integers, addition, and subtraction as syntactic sugar. The only things we need are variables, lambda abstractions, and function applications. We can write any programs with these three kinds of expressions. The *lambda calculus* is a language that provides only

the three features. This book does not discuss how integers, addition, and subtraction can be desugared into the lambda calculus.

9.5 Exercises

1. Consider the following expression:

   $$\texttt{val x=5 in val f=}\lambda\texttt{y.y + x in }(\lambda\texttt{g.f (g 1)})\ (\lambda\texttt{x.x})$$

   Describe a trace of the evalaution in terms of arguments to the `interp` function for every call. (There will be 16 calls.) The `interp` function takes two arguments—an expression and an environment—so show both for each call. For `Num`, `Id`, and `Fun` expressions, show their result values, which are immediate. You can use the following abbreviations and possibly more abbreviations:

   $$\begin{array}{lcl} E_0 & = & \text{the whole expression} \\ E_1 & = & \lambda\texttt{y.y + x} \\ E_2 & = & \lambda\texttt{g.f (g 1)} \\ E_3 & = & \lambda\texttt{x.x} \\ E_4 & = & \texttt{val f=}E_1\texttt{ in }E_2\ E_3 \end{array}$$

2. This exercise examines differences between semantics by changing scope. The following code is an implementation of an interpreter:

```
def interp(e: Expr, env: Env): Value = e match {
  case Num(n) => NumV(n)
  case Add(l, r) =>
    val NumV(n) = interp(l, env)
    val NumV(m) = interp(r, env)
    NumV(n + m)
  case Sub(l, r) =>
    val NumV(n) = interp(l, env)
    val NumV(m) = interp(r, env)
    NumV(n - m)
  case Id(x) => lookup(x, env)
  case Fun(x, b) => CloV(x, b, env)
  case App(f, a) =>
    val CloV(x, b, fEnv) = interp(f, env)
    interp(b, __________ + (x -> interp(a, env)))
}
```

   Describe the semantics of the `App` case in prose when we use each of the following for the blank above:

   - `env`
   - `Map()`
   - `fEnv`

3. This exercise extends FVAE to support multiple parameters. Consider the following language:

   $$\begin{array}{llcl} \text{Expression} & e & ::= & \cdots \mid \lambda x \cdots x.e \mid e(e, \cdots, e) \\ \text{Value} & v & ::= & \cdots \mid \langle \lambda x \cdots x.e, \sigma \rangle \end{array}$$

   The semantics of some constructs are as follows:

- Evaluating $\lambda x_1 \cdots x_n.e$ under σ yields a closure $\langle \lambda x_1 \cdots x_n.e, \sigma \rangle$.
- If
  - evaluating e_0 under σ yields a closure $\langle \lambda x_1 \cdots x_n.e, \sigma' \rangle$,
  - evaluating e_i under σ yields v_i for each $1 \leq i \leq n$, and
  - evaluating e under $\sigma'[x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$ yields v,

  then evaluating $e(e, \cdots, e)$ under σ yields v.

a) Write the operational semantics of the form $\boxed{\sigma \vdash e \Rightarrow v}$ for the expressions.
b) Write the evaluation derivation of the following expression:

$$\emptyset \vdash (\lambda \mathtt{f\ m.f(m)})(\lambda \mathtt{x.x}, 8) \Rightarrow$$

4. Rewrite the following FVAE expression to an FAE expression. You should not "evaluate" it. Consider the approach of Section 9.4.

$$\mathsf{val\ x}{=}\lambda \mathsf{y}.8 + \mathsf{y\ in}\ \lambda \mathsf{y.x}\ (10 - \mathsf{y})$$

5. This exercise modifies FVAE to check body expressions when evaluating function expressions. Consider we extend the value of FVAE to include a special value $\uparrow$ to represent an error during function body checking. Write the operational semantics of the form $\boxed{\sigma \vdash e \Rightarrow v}$ for a function expression $\lambda x.e$, when its semantics changes as follows:

   - If every free identifier of e is in the domain of σ or is x, then evaluation of $\lambda x.e$ under σ yields a closure containing the function expression and the environment.
   - Otherwise, evaluation of $\lambda x.e$ under σ yields $\uparrow$.

   You may use the semantic function fv, which takes an expression and returns the set of every free identifier in the expression. For example, $fv(\lambda \mathsf{x.y\ x}) = \{\mathsf{y}\}$.
6. This exercise extends FVAE with records. Consider the following language:

Field	f	$\in$	*Field*
Record	ρ	$\in$	$\mathit{Record} = \mathit{Field} \overset{\text{fin}}{\twoheadrightarrow} \mathit{Value}$
Expression	e	::=	$\cdots \mid \{f\ e, \ldots, f\ e\} \mid e.f \mid e; e$
Value	v	::=	$\cdots \mid \rho$

   The semantics of some constructs are as follows:

   - The evaluation of $\{f_1\ e_1, \cdots, f_k\ e_k\}$ under σ yields a finite map ρ, which maps $f_i \in \{f_1 \cdots, f_k\}$ to the value v_i which is evaluated from the expression e_i under σ.
   - The evaluation of $e.f$ under σ yields the value of the field f in the record ρ, where evaluation e under σ yields ρ.
   - If evaluation of e_1 yields some value under σ, and evaluation of e_2 yields v under σ, then evaluation of $e_1; e_2$ yields v under σ.

   Write the operational semantics of the form $\boxed{\sigma \vdash e \Rightarrow v}$
7. This exercise extends FVAE with pairs. Consider the following

language:

$$\begin{array}{llll} \text{Expression} & e & ::= & \cdots \mid (e, e) \mid e.1 \mid e.2 \\ \text{Value} & v & ::= & \cdots \mid \boxed{\text{(a)}} \end{array}$$

a) Write the syntax of a pair value in (a) and the operational semantics of the form $\sigma \vdash e \Rightarrow v$ for the expressions.
b) Write the evaluation derivation of the following expression:

$$\frac{}{\emptyset \vdash (8, (320, 42).1).2 \Rightarrow}$$

8. This exercise replaces the environment-based semantics of FVAE with substitution-based semantics. Consider the following implementation:

```
trait Expr
trait Value extends Expr
case class Num(n: Int) extends Expr with Value
case class Add(l: Expr, r: Expr) extends Expr
case class Sub(l: Expr, r: Expr) extends Expr
case class Val(x: String, e: Expr, b: Expr) extends Expr
case class Id(x: String) extends Expr
case class Fun(x: String, b: Expr) extends Expr with Value
case class App(f: Expr, a: Expr) extends Expr

def subst(e: Expr, x: String, v: Value): Expr = e match {
  case Num(n) => e
  case Add(l, r) =>
    Add(subst(l, x, v), subst(r, x, v))
  case Sub(l, r) =>
    Sub(subst(l, x, v), subst(r, x, v))
  case Val(y, i, b) =>
    val nb = if (y == x) b else subst(b, x, v)
    Val(y, subst(i, x, v), nb)
  case Id(name) =>
    if (name == x) v else e
  case Fun(y, b) =>
    Fun(y, if (y == x) b else subst(b, x, v))
  case App(f, a) =>
    App(subst(f, x, v), subst(a, x, v))
}

def interp(e: Expr): Value = e match {
  case Num(n) => Num(n)
  case Add(l, r) =>
    val Num(n) = interp(l)
    val Num(m) = interp(r)
    Num(n + m)
  case Sub(l, r) =>
    val Num(n) = interp(l)
    val Num(m) = interp(r)
    Num(n + m)
  case Val(x, i, b) =>
```

```
    interp(subst(b, x, interp(i)))
  case Id(x) => error("free identifier")
  case Fun(x, b) => Fun(x, b)
  case App(f, a) =>
    val Fun(x, b) = interp(f)
    interp(subst(b, x, interp(a)))
}
```

In this implementation, a value is either an integer or a lambda abstraction:

$$v ::= n \mid \lambda x.e$$

a) Write the operational semantics of the above implementation of the form $\boxed{e \Rightarrow v}$ where $e[x/v]$ denotes `subst(e, x, v)`.
b) Write the definition of the substitution $e[x/v]$ of the form $\boxed{e[x/v] = e}$:
c) Consider the following expression:

$$\text{val z=}\lambda\text{x.x} - \text{y in val y=10 in (z 32)}$$

i. What is the result of evaluating the expression under the empty environment in substitution-based FVAE?
ii. What is the result of evaluating the expression under the empty environment in environment-based FVAE?
iii. Why are the results different? [3]

3: We can make the semantics of substitution-based FVAE equivalent to environment-based FVAE by modifying `subst` function.

9. Consider the following language:

$$\begin{array}{lrcl} \text{Expression} & e & ::= & n \mid x \mid \lambda x \cdots x.e \mid e(e, \cdots, e) \mid \textsf{get}\ e \\ \text{Value} & v & ::= & n \mid \textsf{undefined} \mid \langle \lambda x \cdots x.e, \sigma \rangle \end{array}$$

The semantics of some constructs are as follows:

- The value of a function expression $\lambda x_1 \cdots x_n.e$ at an environment σ is a closure $\langle \lambda x_1 \cdots x_n.e, \sigma \rangle$.
- A function application $e_0(e_1, \cdots, e_n)$ is evaluated as follows:
  - Evaluate the subexpressions in order. The value of e_0 should be a closure $\langle \lambda x_1 \cdots x_m.e, \sigma \rangle$ that has m parameters.
  - Create an array α of size n and initialize the i-th value of the array with the value of e_{i+1} where $0 \leq i \leq n-1$.
  - Evaluate the closure body e under the environment σ extended as follows:
    - The value of the i-th parameter is the value of e_i where $1 \leq i \leq \min(m, n)$.
    - The value of the j-th parameter is the undefined value where $n < j \leq m$.

    and the array α.
- The value of get e is the n-th value of the array α where n is the value of e and the array indices start from 0.

For example, $(\lambda\text{xy.y})(4)$ evaluates to undefined, and $(\lambda\text{x.get } 0)(5)$ evaluate to 5.

a) Write the operational semantics of the form $\boxed{\sigma, \alpha \vdash e \Rightarrow v}$.

b) Write the evaluation derivation of the following expression:

$$\frac{}{\emptyset, \emptyset \vdash (\lambda \mathsf{x\ y.get}\ x)(2, 19, 141) \Rightarrow}$$

10. The following quote describes the JavaScript sequencing semantics:[4]

> The value of a *StatementList* is the value of the last value-producing item in the *StatementList*. For example, the following calls to the `eval` function all return the value 1:
>
> ```
> eval("1;;;;;")
> eval("1;()")
> eval("1;var a;")
> ```

Consider the following language:

$$\begin{array}{lrcl} \text{Expression} & e & ::= & () \mid x \mid \lambda x.e \mid e\ e \mid e; \cdots ; e \\ \text{Value} & v & ::= & () \mid \langle \lambda x.e, \sigma \rangle \end{array}$$

The value of the sequence expression $e_1; \cdots ; e_n$ is the value of the last expression whose value is not (). If the values of all the expressions $e_1, \cdots, e_n$ are (), the value of the sequence expression is (). Write the operational semantics of each expression of the form $\boxed{\sigma \vdash e \Rightarrow v}$.

11. Consider the following language:

$$\begin{array}{rcll} e & ::= & a & \text{atomic expression} \\ & \mid & e\ a & \text{function application} \\ & \mid & \mathsf{fn}\ m & \text{function expression} \\ a & ::= & n & \text{number} \\ & \mid & x & \text{identifier} \\ m & ::= & p \rightsquigarrow e & \text{pattern matching} \\ & \mid & p \rightsquigarrow e \mid m & \text{pattern matching sequence} \\ p & ::= & \_ & \text{wildcard pattern} \\ & \mid & n & \text{number pattern} \\ & \mid & x & \text{identifier pattern} \end{array}$$

where a value of the language v is either a number n or a closure $\langle m, \sigma \rangle$, a result of evaluation r is either a value v or a failure in pattern matching $\uparrow$, which is different from run-time errors, and an environment σ maps identifiers to their values.
The operational semantics rules for expressions and atomic expressions are as follows:
$\boxed{\sigma \vdash e \Rightarrow r}$

$$\frac{\sigma \vdash a \hookrightarrow v}{\sigma \vdash a \Rightarrow v} \qquad \frac{\sigma \vdash e \Rightarrow \uparrow}{\sigma \vdash e\ a \Rightarrow \uparrow} \qquad \sigma \vdash \mathsf{fn}\ m \Rightarrow \langle m, \sigma \rangle$$

$$\frac{\sigma \vdash e \Rightarrow \langle m, \sigma' \rangle \qquad \sigma \vdash a \Rightarrow v \qquad (\sigma', v) \vdash m \Rightarrow v'}{\sigma \vdash e\ a \Rightarrow v'}$$

$$\frac{\sigma \vdash e \Rightarrow \langle m, \sigma' \rangle \qquad \sigma \vdash a \Rightarrow v \qquad (\sigma', v) \vdash m \Rightarrow \uparrow}{\sigma \vdash e\ a \Rightarrow \uparrow}$$

4: https://tc39.es/ecma262/#sec-block-runtime-semantics-evaluation

$\boxed{\sigma \vdash a \hookrightarrow v}$

$$\sigma \vdash n \hookrightarrow n \qquad \frac{x \in \mathit{Domain}(\sigma)}{\sigma \vdash x \hookrightarrow \sigma(x)}$$

The semantics of pattern matching m and pattern p are as follows:

- Evaluation of $p \rightsquigarrow e$ under (σ, v) has two possibilities. First, when evaluation of p results in a new environment σ', the result of this pattern matching is the result of evaluation of e under $\sigma + \sigma'$, where $\sigma + \sigma'$ is a disjoint union of σ and σ'. Second, when evaluation of p produces $\uparrow$, the evaluation of this pattern matching produces $\uparrow$ as well.
- Evaluation of "$p \rightsquigarrow e \mid m$" under (σ, v) also has two possibilities. First, when evaluation of $p \rightsquigarrow e$ succeeds with a value v', the value of this pattern matching sequence is v'. Second, when evaluation of $p \rightsquigarrow e$ fails, the result of evaluation of this pattern matching sequence is the result of evaluation of m.
- Evaluation of the wildcard pattern $\_$ under (σ, v) produces the empty environment.
- Evaluation of the number pattern n under (σ, v) has two possibilities. If $v = n$, it produces the empty environment. Otherwise, it produces $\uparrow$.
- Evaluation of the identifier pattern x under (σ, v) produces a singleton environment $\{x \mapsto v\}$ if x is not in the domain of σ.

Write the operational semantics for m and p of the forms $\boxed{(\sigma, v) \vdash m \Rightarrow r}$ and $\boxed{(\sigma, v) \vdash p \Rightarrow \sigma/\uparrow}$, respectively, where $\boxed{(\sigma, v) \vdash p \Rightarrow \sigma/\uparrow}$ denotes $\boxed{(\sigma, v) \vdash p \Rightarrow \sigma}$ or $\boxed{(\sigma, v) \vdash p \Rightarrow \uparrow}$. Remember that the operational semantics do not specify run-time errors.

10 Recursion

Recursive functions are widely used in programming. We have discussed importance of recursion in Section 3.2. This chapter explains the semantics of recursive functions by defining RFAE, which extends FAE with recursive functions. In addition, we will see that recursive functions also are just syntactic sugar: we can express recursive functions with first-class functions.

Consider the following Scala program:

```
def sum(x: Int): Int =
  if (x == 0)
    0
  else
    x + sum(x - 1)

println(sum(10))
```

The function `sum` takes an integer `n` as an argument, and returns the sum of the integers between `0` to `n` (including `n`). [1] Therefore, the program prints `55`.

1: We ignore the case when the input is negative.

How can we implement an equivalent program in FAE? One naïve approach could be the following expression:[2]

$$\text{val sum}{=}\lambda\text{x.if0 x 0 (x + sum (x} - 1)) \text{ in sum } 10$$

2: FAE does not provide conditional expressions (if0), which is defined by RFAE, but we use it since we can easily add them to FAE.

However, it is wrong since the scope of `sum` includes `sum 10` but excludes λ`x.if0 x 0 (x + sum (x` $-$ `1))`. `sum` in the body of the function is a free identifier. Evaluation of the expression termiantes with a run-time error. It is nontrival to define recursive functions in FAE.

10.1 Syntax

We define RFAE by extending FAE with recursive functions. To demonstrate the usefulness of recursion with examples, we add conditional expressions as well.

The following is the abstract syntax of RFAE: [3]

$$e \ ::= \ \cdots \mid \text{if0 } e \ e \ e \mid \text{def } x(x){=}e \text{ in } e$$

3: We omit the common part to FAE.

- if0 e_1 e_2 e_3
  It is a conditional expression. e_1 is the condition; e_2 is the true branch; e_3 is the false branch. We consider the condition to be true when it denotes 0. All the other values, i.e. nonzero integers and closures, are considered as false.

- def $x_1(x_2)$=e_1 in e_2
  It defines a recursive function whose name is x_1, parameter is x_2, and body is e_1. Both x_1 and x_2 are binding occurrences. The scope of x_1 is e_1 and e_2; the scope of x_2 is e_1. If x_1 occurs in e_1, it is a bound occurrence, which implies that the function can be recursive.

In RFAE, we can implement a function computing the sum of consecutive integers like below.

```
def sum(x)=if0 x 0 (x + sum (x − 1)) in sum 10
```

10.2 Semantics

The semantics of conditional expressions is quite easy. The semantics consists of two rules: one for when the condition is true and the other for when the condition is false.

The following rule defines the semantics when the condition is true:

Rule If0-Z
If e_1 evaluates to 0 under σ and e_2 evaluates to v under σ, then if0 e_1 e_2 e_3 evaluates to v under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow 0 \qquad \sigma \vdash e_2 \Rightarrow v}{\sigma \vdash \text{if0 } e_1\ e_2\ e_3 \Rightarrow v} \quad [\text{If0-Z}]$$

When e_1 evaluates to 0, the condition is considered as true, and the true branch, e_2, is evaluated. The result of e_2 is the result of the whole expression.

The following rule defines the semantics when the condition is false:

Rule If0-Nz
If e_1 evaluates to v' under σ, where $v' \neq 0$, and e_3 evaluates to v under σ, then if0 e_1 e_2 e_3 evaluates to v under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow v' \qquad v' \neq 0 \qquad \sigma \vdash e_3 \Rightarrow v}{\sigma \vdash \text{if0 } e_1\ e_2\ e_3 \Rightarrow v} \quad [\text{If0-Nz}]$$

When e_1 evaluates to a value other than 0, the condition is considered as false, and the false branch, e_3, is evaluated. The result of e_3 is the result of the whole expression.

Now, let us discuss the semantics of recursive functions. Consider def $x_1(x_2)$=e_1 in e_2. e_2 is in the scope of x_1, and x_1 denotes a function. Therefore, we can start by defining $\sigma' = [x_1 \mapsto \langle \lambda x_2.e_1, \sigma \rangle]$ and evaluating e_2 under σ'. However, this apporach is incorrect. When the body of the closure, e_1, is evaluated, the environment is $\sigma[x_2 \mapsto v]$ for some v. The environment does not contain x_1, and therefore using x_1 in e_1 will incur a free identifier error, which is certainly wrong since x_1 is a recursive function. The environment of the closure must include x_1 and its value, which is the closure itself. From this observation, we obtain the

following rule:

Rule Rec
If e_2 evaluates to v under σ', where $\sigma' = \sigma[x_1 \mapsto \langle \lambda x_2.e_1, \sigma' \rangle]$,
then def $x_1(x_2)$=e_1 in e_2 evaluates to v under σ.

$$\frac{\sigma' = \sigma[x_1 \mapsto \langle \lambda x_2.e_1, \sigma' \rangle] \qquad \sigma' \vdash e_2 \Rightarrow v}{\sigma \vdash \mathsf{def}\ x_1(x_2)\text{=}e_1\ \mathsf{in}\ e_2 \Rightarrow v}\ [\textsc{Rec}]$$

The environment of the closure is σ', which has x_1 and the closure. σ' is recursively defined at the meta-level. It is not that surprising. We are defining a recursive function, so the defined function value itself should be recursive. When the body of the closure, e_1, is evaluated, the environment is $\sigma'[x_2 \mapsto v]$ for some v, which contains x_1. The function x_1 can be used in its body and thus recursive.

We can reuse the rules of FAE for the other expressions.

The following proof trees prove that def f(x)=if0 x 0 (x + f (x − 1)) in f 1 evaluates to 1 under the empty environment. The proof splits into three trees for readability. Suppose the following facts:

$$\begin{aligned} e_\mathsf{f} &= \mathsf{if0\ x\ 0\ (x + f\ (x - 1))} \\ v_\mathsf{f} &= \langle \lambda \mathsf{x}.e_\mathsf{f}, \sigma_1 \rangle \\ \sigma_1 &= [\mathsf{f} \mapsto v_\mathsf{f}] \\ \sigma_2 &= \sigma_1[\mathsf{x} \mapsto 1] \\ \sigma_3 &= \sigma_1[\mathsf{x} \mapsto 0] \end{aligned}$$

$$\dfrac{\dfrac{\mathsf{f} \in \mathit{Domain}(\sigma_2)}{\sigma_2 \vdash \mathsf{f} \Rightarrow v_\mathsf{f}} \quad \dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}(\sigma_2)}{\sigma_2 \vdash \mathsf{x} \Rightarrow 1} \quad \sigma_2 \vdash 1 \Rightarrow 1}{\sigma_2 \vdash \mathsf{x} - 1 \Rightarrow 0} \quad \dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}(\sigma_3)}{\sigma_3 \vdash \mathsf{x} \Rightarrow 0} \quad \sigma_3 \vdash 0 \Rightarrow 0}{\sigma_3 \vdash e_\mathsf{f} \Rightarrow 0}}{\sigma_2 \vdash \mathsf{f}\ (\mathsf{x} - 1) \Rightarrow 0}$$

$$\dfrac{\dfrac{\mathsf{f} \in \mathit{Domain}(\sigma_1)}{\sigma_1 \vdash \mathsf{f} \Rightarrow v_f} \quad \sigma_1 \vdash 1 \Rightarrow 1 \quad \dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}(\sigma_2)}{\sigma_2 \vdash \mathsf{x} \Rightarrow 1} \quad 1 \neq 0 \quad \dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}(\sigma_2)}{\sigma_2 \vdash \mathsf{x} \Rightarrow 1} \quad \sigma_2 \vdash \mathsf{f}\ (\mathsf{x} - 1) \Rightarrow 0}{\sigma_2 \vdash \mathsf{x} + \mathsf{f}\ (\mathsf{x} - 1) \Rightarrow 1}}{\sigma_2 \vdash e_\mathsf{f} \Rightarrow 1}}{\sigma_1 \vdash \mathsf{f}\ 1 \Rightarrow 1}$$

$$\frac{\sigma_1 = [\mathsf{f} \mapsto v_\mathsf{f}] \qquad \sigma_1 \vdash \mathsf{f}\ 1 \Rightarrow 1}{\emptyset \vdash \mathsf{def\ f(x)}\text{=}e_\mathsf{f}\ \mathsf{in\ f}\ 1 \Rightarrow 1}$$

10.3 Interpreter

The following Scala code implements the syntax of RFAE: [4]

4: We omit the common part to FAE.

```
sealed trait Expr
...
case class If0(c: Expr, t: Expr, f: Expr) extends Expr
case class Rec(f: String, x: String, b: Expr, e: Expr) extends Expr
```

`If0(`e_1`, `e_2`, `e_3`)` represents if0 e_1 e_2 e_3; `Rec(`x_1`, `x_2`, `e_1`, `e_2`)` represents def x_1(x_2)=e_1 in e_2.

```
sealed trait Value
case class NumV(n: Int) extends Value
case class CloV(p: String, b: Expr, var e: Env) extends Value
```

Values are defined in a similar way to FAE. The only difference is that the field `e`, which denotes the captured environment, of `CloV` is now mutable. Using mutation is the easiest way to make recursive values in Scala, though we can do it without mutation.

```
def interp(e: Expr, env: Env): Value = e match {
  ...
  case If0(c, t, f) =>
    interp(if (interp(c, env) == NumV(0)) t else f, env)
  case Rec(f, x, b, e) =>
    val cloV = CloV(x, b, env)
    val nenv = env + (f -> cloV)
    cloV.e = nenv
    interp(e, nenv)
}
```

In the `If0` case, the condition is evaluated first. According to the condition, one of the branches is evaluated.

In the `Rec` case, we construct a closure first. However, the closure is not complete at this point. We next create a new environment: the environment with the closure. The closure must capture the new environment. To achieve this, we change the environment of the closure to the new environment. Now, both closure and environment have recursive structures, and `e` can be evaluated under the environment.

10.4 Recursion as Syntactic Sugar

Even if a language does not support recursive functions, we can implement recursive functions with first-class functions, i.e. we can implement recursive functions in FAE. The key to desugar recursive functions into FAE is the following function:

$$Z = \lambda \mathtt{f}.(\lambda \mathtt{x}.\mathtt{f}\ (\lambda \mathtt{v}.\mathtt{x}\ \mathtt{x}\ \mathtt{v}))\ (\lambda \mathtt{x}.\mathtt{f}\ (\lambda \mathtt{v}.\mathtt{x}\ \mathtt{x}\ \mathtt{v}))$$

Z is called a *fixed point combinator*. In mathematics, a *fixed point* of a function f is a solution of the equation $f(x) = x$. A fixed point combinator is a function that computes a fixed point of a given function. A recursive function can be considered as a fixed point of a certain function. For example, consider the following function:

$$\lambda \mathtt{f}.\lambda \mathtt{v}.\mathtt{if0}\ \mathtt{v}\ 0\ (\mathtt{v} + \mathtt{f}\ (\mathtt{v} - 1))$$

Let us call the function f. Suppose that *sum* is given to the function as an

argument, where *sum* is a function that takes a natural number n as an argument and returns $\sum_{i=0}^{n} i$. Then, the return value of f is

$$\lambda \mathsf{v}.\mathsf{if0\ v\ 0\ (v} + \mathit{sum}\ (\mathsf{v} - 1))$$

It is a function again. When 0 is given to the function, the result is 0. When a positive integer n is given to the function, the result is $n + \mathit{sum}\ (n-1)$, which equals $\mathit{sum}\ n$. Therefore, $f\ \mathit{sum}$ equals *sum*, and we can say that *sum* is a fixed point of f.

The fixed point combinator Z takes a function as an argument and returns a fixed point of the function. Therefore, $Z\ f$ equals *sum*. Let us see the reason. We use e_f as an abbreviation of $\mathsf{if0\ v\ 0\ (v + f\ (v - 1))}$. Then, $Z\ f$ is the same as

$$Z\ (\lambda \mathsf{f}.\lambda \mathsf{v}.e_f)$$

It evaluates to $F\ F$ where F denotes

$$\lambda \mathsf{x}.(\lambda \mathsf{f}.\lambda \mathsf{v}.e_f)\ (\lambda \mathsf{v.x\ x\ v})$$

By expanding only the first F in $F\ F$, we get

$$(\lambda \mathsf{x}.(\lambda \mathsf{f}.\lambda \mathsf{v}.e_f)\ (\lambda \mathsf{v.x\ x\ v}))\ F$$

which evaluates to

$$(\lambda \mathsf{f}.\lambda \mathsf{v}.e_f)\ (\lambda \mathsf{v}.F\ F\ \mathsf{v})$$

By expanding e_f, we get

$$(\lambda \mathsf{f}.\lambda \mathsf{v.if0\ v\ 0\ (v + f\ (v - 1)))}\ (\lambda \mathsf{v}.F\ F\ \mathsf{v})$$

which evalutes to

$$\lambda \mathsf{v.if0\ v\ 0\ (v} + (\lambda \mathsf{v}.F\ F\ \mathsf{v})\ (\mathsf{v} - 1))$$

Let us call this function g. Note that we now know that $F\ F$ evaluates to g. If we apply g to 0, the result is 0. If we apply g to a positive integer n, the result is

$$n + (\lambda \mathsf{v}.F\ F\ \mathsf{v})\ (n-1)$$

which evalutes to

$$n + F\ F\ (n-1)$$

Since $F\ F$ evaluates to g, the above expression evaluates to

$$n + g\ (n-1)$$

Then, the semantics of g is

$$g\ n = \begin{cases} 0 & \text{if } n = 0 \\ n + g\ (n-1) & \text{otherwise} \end{cases}$$

It implies that g equals *sum*. Actually, this evaluation has started from $Z\ f$. Therefore, $Z\ f$ equals *sum*.

One may ask if we can use the following expression Z' instead of Z:

$$Z' = \lambda \mathsf{f}.(\lambda \mathsf{x.f\ (x\ x)})\ (\lambda \mathsf{x.f\ (x\ x)})$$

It is a natural question because x x does the same thing as λv.x x v when applied to an argument. However, the answer is no. Eta-expanding x x to λv.x x v has the effect of delaying computation on x x. While x x should be immediately evaluated, x x in λv.x x v can be evaluated only when the function is applied to an argument. This delaying effect is necessary in Z. Suppose that we use Z'. Then, Z' f evaluates to F' F' where F' denotes

$$\lambda \mathsf{x}.(\lambda \mathsf{f}.\lambda \mathsf{v}.e_f)\ (\mathsf{x}\ \mathsf{x})$$

By expanding only the first F', we get

$$(\lambda \mathsf{x}.(\lambda \mathsf{f}.\lambda \mathsf{v}.e_f)\ (\mathsf{x}\ \mathsf{x}))\ F'$$

which evaluates to

$$(\lambda \mathsf{f}.\lambda \mathsf{v}.e_f)\ (F'\ F')$$

To evaluate this expression, we need to evaluate the argument. However, the argument is F' F', which implies that we need the value of F' F' to compute the value of F' F'. Thus, at this point, the execution starts to evaluate F' F' forever and never terminates. For this reason, we should use Z, not Z', as a fixed point combinator.

Finally, we can define the syntactic transformation rule to desugar recursive functions: def $x_1(x_2)$=e_1 in e_2 is transformed into val x_1=Z $(\lambda x_1.\lambda x_2.e_1)$ in e_2. If x_1 in def $x_1(x_2)$=e_1 in e_2 denotes a function h, h is a fixed point of $\lambda x_1.\lambda x_2.e_1$. Therefore, Z $(\lambda x_1.\lambda x_2.e_1)$ is equal to h. Both def $x_1(x_2)$=e_1 in e_2 and val x_1=Z $(\lambda x_1.\lambda x_2.e_1)$ in e_2 evaluate e_2 under the environment that x_1 denotes h. Therefore, they have the same semantics, and the desugaring is correct.

10.5 Exercises

1. Explain why the following expression does not terminate and describe how to fix it.

```
val z=(λf.
  val x=(λy.
    val g=y y in
    f g
  ) in
  x x
) in
val f=z (λf.λv.if0 v 0 (v + f (v − 1))) in
f 10
```

2. Consider the following definition of z and its use to define the recursive function f.

```
val z=(λf.
  val x=(λy.
    val g=λa.y y a in
    f g
  ) in
  x x
) in
val f=z (λf.λv.if0 v 0 (v + f (v − 1))) in
f 10
```

   Describe conditions that an argument given to z must satisfy so

that z can make its recursive version.

3. Consider the following expression:

```
val f=(
  val x=λy.(
    val f=λv.y y v in
    λv.if0 v 0 (v + f (v − 1))
  ) in
  x x
) in
f 10
```

   a) Draw arrows on the above expression from each bound variable to its binding occurrence.
   b) Draw dotted arrows on the above expression from each shadowing variable to its shadowed variable.
   c) Write the value of `f` at the last line by using the following Scala types:

```
trait Value
case class NumV(n: Int) extends Value
case class CloV(p: String, b: Expr, e: Env) extends Value
type Env = Map[String, Value]
```

4. Consider the following language:

$$
\begin{array}{lrcl}
\text{Expression} & e & ::= & n \mid e - e \mid b \mid e \wedge e \mid \neg e \mid \textsf{if } e\ e\ e \mid x \\
& & \mid & \lambda x \cdots x.e \mid e(e, \cdots, e) \mid \textsf{def } x(x, \cdots, x)\text{=}e \textsf{ in } e \\
\text{Value} & v & ::= & n \mid b \mid \langle \lambda x \cdots x.e, \sigma \rangle
\end{array}
$$

   Note that b ranges over boolean values. The language does not support the short-circuiting semantics, i.e. e_2 must be evaluated in expression $e_1 \wedge e_2$ whenever e_1 evaluates to true or not. Write the operational semantics of the form $\boxed{\sigma \vdash e \Rightarrow v}$ for the expressions.

5. What are the results of the following expression:

```
def f(x)=if0 x x (1 + (f (x − 1))) in
val f=λx.42 + (f x) in
f 7
```

   in different scoping semantics when we evaluate it under the following environment?

```
Map("f" -> CloV("x", Add(Num(13), Id("x")), Map()))
```

   a) Dynamic scope
   b) Static scope

11 Boxes

Mutation is a widely-used feature. It is an important concept in imperative languages. Even functional languages support mutation. Few languages are purely functional, i.e. do not allow any mutation: e.g. Haskell and Coq. Muation is important since many programs can be implemented concisely and efficiently with mutation. At the same time, mutation often makes programs difficult to be reasoned about and error-prone. While binding of identifiers works modularly and allows local reasoning, mutation has a global effect on execution and enables uncontrolled interference between distinct parts of a program. Mutation should be used with extreme care of programmers.

This chapter introduces mutation by defining BFAE, which extends FAE with boxes. A *box* is a cell in memory that contains a single value. The value contained in a box can be modified anytime after the creation of the box. Boxes in BFAE are higher-order. Each box can contain any value, which can be a box or a closure, rather than only an integer. A box itself is rarely found in real-world languages: it is almost the same as a reference in OCaml (`ref`). However, it is a good abstraction of more general mutation mechanisms including mutable objects and data structures. We can find such concepts in most languages, and boxes are useful to understand those concepts.

We can consider mutable objects in Scala as generalization of boxes. By going the opposite direction, we can say that boxes can be represented as objects. Consider the following class definition in Scala:

```
case class Box(var value: Any)
```

The class `Box` has one field: `value`. Like any other classes, we can construct instances of `Box` and read the fields of the instances.

```
val b = Box(3)
println(b.value)
```

It prints `3`. In addition, since the field `value` is declared as mutable, we can mutate its value.

```
b.value = 10
println(b.value)
```

It changes the value of the field to `10` and prints `10`.

11.1 Syntax

The above Scala example shows three kinds of expressions regarding boxes: creating a new box, reading the value of a box, and changing the

value of a box. To create a box, we need an expression that determines the initial value of the box. To read the value of a box, we need an expression that determines the box. To change the value of a box, we need an expression that determines the box and an expression that determines the new value of the box.

In addition, there is another kind of expression that has been implicitly used: the sequencing expression. Usually, an expression mutating a box is useless per se. There should be other expressions that observe the change and do other interesting things based on the change. To do so, we need to combine multiple expressions to form a single expression. Such an expression is the sequencing expression.

We can define the syntax of BFAE based on the observations. The following is the syntax of BFAE: [1]

1: We omit the common part to FAE.

$$e ::= \cdots \mid \mathsf{box}\ e \mid !e \mid e{:=}e \mid e; e$$

- box e
  It creates a new box, cf. `Box(3)` in the example. e determines the initial value of the box.
- $!e$
  It reads the value of a box, cf. `b.value` in the example. e determines the box to be read.
- $e_1{:=}e_2$
  It changes the value of a box, cf. `b.value = 10` in the example. e_1 determines the box to be updated; e_2 determines the new value.
- $e_1; e_2$
  It is a sequencing expression. e_1 is the first expression to be evaluated; e_2 is the second. Many real-world languages allow sequencing of an arbitrary number of expressions. For brevity, BFAE allows only sequencing of two expressions. Sequencing of multiple expressions can be easily expresssed by nested sequencing. For example, $e_1; e_2; e_3$ can be expresssed as $(e_1; e_2); e_3$.

11.2 Semantics

Defining mutable memory is crucial to define the semantics of BFAE. We call the memory of a program a *store*. A store records the values of boxes. Each box is distinguished from another box by its address, i.e. every box has its own address, which differs from the addresses of any other boxes. The metavarible a ranges over addresses. Let *Addr* be the set of every possible address. We do not care about what *Addr* really is.

$$a \in Addr$$

A store is a finite partial function from addresses to values. If the store maps an address a to a value v, the value of the box whose address is a is v. The metavarible M ranges over stores. Let *Sto* be the set of every store.

$$Sto = Addr \xrightarrow{\text{fin}} V$$

$$M \in Sto$$

The semantics does not require a concrete notion of a box. Since every box is uniquely identified by an address, the semantics can consider each address as a box. Thus, we treat an address as a value of BFAE, instead of introducing a new semantic element denoting boxes. For example, an expression creating a box evaluates to an address. We need to revise the definition of a value to include addresses. [2]

2: We omit the common part to FAE.

$$v ::= \cdots \mid a$$

Note that we keep using the concept of a box for explanation. Even though the semantics abstracts boxes with addresses, boxes do exist from the programmers' perspective. The term box and the term address will be interchangeably used.

How are stores used in the semantics? First, consider an expression reading a box. Evaluating !e needs not only an environment but also a store. If e denotes a box, the store has the value of the box. The value becomes the result of !e. Without a store, there is no way to find the value of a box and yield a result. It implies that evaluation requires a store to be given.

Now, let us consider the other kinds of expressions related to boxes. box e creates a new box; e_1:=e_2 changes the content of a box. Both modify stores. Modifying a store differs from extending an environment with a new identifier.

A change in an environment is propagated to the subexpressions of an expression that has caused the change. Consider val x=e_1 in e_2. It extends the environment with x, but only e_2 uses the extended environment because the scope of x is e_2 but nowhere else. A variable definition can affect only its subexpressions. For instance, in (val x=e_1 in e_2) + e_3, e_3 does not belong to the scope of x. The extended environment must be used for only e_2, but not e_3. Therefore, we say that binding and environments are local and modular.

On the other hand, the modified store is unnecessary for the subexpressions of an expression that modifies the store, while other parts of the program need the modified one. Consider (x:=2); !x as an example. Assume that x denotes a box. !x must be aware of that x:=2 has changed the value of the box to 2. Otherwise, !x will get the previous value of the box and produce a wrong result. Note that !x is not a subexpression of x:=2. However, in x:=2, the evaluation of 2, which is a subexpression of x:=2, must not be affected by the change in the value of the box since the change happens after the evaluation of 2. Therefore, how stores change due to expressions is important. If an expression contains two subexpressions, the store obtained by evaluating the first subexpression has to be passed to the evaluation of the second subexpression. Stores are completely different from environments. Any change in a store affects the entire remaining computation. Stores are global and not modular.

From the observation, we can conclude that evaluation of an expression needs to take a store in addition to an environment as input and output a new store along with a result value. We can define the semantics as a

relation over *Env*, *Sto*, *E*, *V*, and *Sto*. The former store is input, and the latter store is output.

$$\Rightarrow \subseteq Env \times Sto \times E \times V \times Sto$$

$\sigma, M_1 \vdash e \Rightarrow v, M_2$ is true if and only if e evaluates to v and changes the store from M_1 to M_2 under σ. We call this way of defining semantics a *store-passing style*. The style allows defining BFAE, featuring mutability, without any mutable concepts at the meta-level.

Now, let us define the semantics of each expression. We can easily reuse Rule Num, Rule Id, and Rule Fun of BFAE by adding stores. They maintain the contents of given stores.

Rule Num
n evaluates to n and changes the store from M to M under σ.

$$\sigma, M \vdash n \Rightarrow n, M \quad \text{[Num]}$$

Rule Id
If x is in the domain of σ,
then x evaluates to $\sigma(x)$ and changes the store from M to M under σ.

$$\frac{x \in Domain(\sigma)}{\sigma, M \vdash x \Rightarrow \sigma(x), M} \quad \text{[Id]}$$

Rule Fun
$\lambda x.e$ evaluates to $\langle \lambda x.e, \sigma \rangle$ and changes the store from M to M under σ.

$$\sigma, M \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \sigma \rangle, M \quad \text{[Fun]}$$

During the evaluation of a certain expression, the order of the evaluation among the subexpressions matters as they can modify the store. Suppose that x denotes a box, and the box contains 1. If the left operand of addition is evaluated before the right operand, in (x:=2)+!x, !x evaluates to 2 since it is affected by the previous change. On the other hand, if the right operand is evaluated first, !x evaluates to 1 because its evaluation precedes the modification and can observe only the original value.

The order among the premises in a semantics rule does not specify the order of evaluation. So far, we have not specified the order of evaluation in the semantics as the order does not matter if there are no side effects.[3] However, BFAE supports mutation, and we should specify the order in the semantics. This goal can be naturally achieved by passing stores. If we define the semantic to use the store that comes out from the evaluation of the left operand as input of the evaluation of the right operand, the order is determined to evaluate the left first.

A sequencing expression per se cannot modify a given store, but its subexpressions can.

3: Side effects are any observable behaviors of expressions except the results. Mutation and exceptions are side effects.

Rule Seq
If
e_1 evaluates to v_1 and changes the store from M to M_1 under σ, and
e_2 evaluates to v_2 and changes the store from M_1 to M_2 under σ,
then
$e_1; e_2$ evaluates to v_2 and changes the store from M to M_2 under σ.

$$\frac{\sigma, M \vdash e_1 \Rightarrow v_1, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash e_1; e_2 \Rightarrow v_2, M_2} \text{ [Seq]}$$

e_1 is evaluated before e_2. The evaluation of e_2 must be aware of any modifications of the store made by e_1. For this purpose, the rule passes M_1, obtained by evaluating e_1, to the evaluation of e_2. The result of e_1 is just discarded. The final result is the same as the result of e_2.

Rule Add, Rule Sub, and Rule App are similar to Rule Seq. They cannot modify stores, but their subexpressions can. The evaluation order is the same as the sequencing expression. BFAE chooses the left-to-right order for every expression, but other languages may use a different order.

Rule Add
If
e_1 evaluates to n_1 and changes the store from M to M_1 under σ, and
e_2 evaluates to n_2 and changes the store from M_1 to M_2 under σ,
then
$e_1 + e_2$ evaluates to $n_1 + n_2$ and changes the store from M to M_2 under σ.

$$\frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 + e_2 \Rightarrow n_1 + n_2, M_2} \text{ [Add]}$$

Rule Sub
If
e_1 evaluates to n_1 and changes the store from M to M_1 under σ, and
e_2 evaluates to n_2 and changes the store from M_1 to M_2 under σ,
then
$e_1 - e_2$ evaluates to $n_1 - n_2$ and changes the store from M to M_2 under σ.

$$\frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 - e_2 \Rightarrow n_1 - n_2, M_2} \text{ [Sub]}$$

Rule App
If
e_1 evaluates to $\langle \lambda x.e, \sigma' \rangle$ and changes the store from M to M_1 under σ,
e_2 evaluates to v' and changes the store from M_1 to M_2 under σ, and
e evaluates to v and changes the store from M_2 to M_3 under $\sigma'[x \mapsto v']$,
then
$e_1\ e_2$ evaluates to v and changes the store from M to M_3 under σ.

$$\frac{\begin{array}{c}\sigma, M \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle, M_1 \\ \sigma, M_1 \vdash e_2 \Rightarrow v', M_2 \qquad \sigma'[x \mapsto v'], M_2 \vdash e \Rightarrow v, M_3\end{array}}{\sigma, M \vdash e_1\ e_2 \Rightarrow v, M_3} \text{ [App]}$$

Note that the evaluation of the body of a closure can modify the store as well.

Now, let us define the semantics of expressions treating boxes. box e is an expression creating a new box. The result of e becomes the initial value of the box. The result of box e is the new box.

Rule NewBox
If
e evaluates to v and changes the store from M to M_1 under σ, and
a is not in the domain of M_1,
then
box e evaluates to a and changes the store from M to $M_1[a \mapsto v]$ under σ.

$$\frac{\sigma, M \vdash e \Rightarrow v, M_1 \qquad a \notin Domain(M_1)}{\sigma, M \vdash \mathsf{box}\ e \Rightarrow a, M_1[a \mapsto v]} \quad [\textsc{NewBox}]$$

To get the initial value, e is evaluated first. The address of the new box must not belong to M_1, the store attained by evaluating e. There is no additional condition the address must satisfy, so we can freely choose any address that is not in M_1. Note that if we check the domain of M, not M_1, we result in multiple boxes sharing the same address, which is certainly wrong, when e also creates boxes. The result is the address of the box. Also, we add a mapping from the address of the box to the value of the box to the final store.

!e is an expression reading the value of a box. e determines the box to be read. If e does not evaluate to a box, a run-time error occurs. Otherwise, e is some box, and the final result is the value of the box.

Rule OpenBox
If
e evaluates to a and changes the store from M to M_1 under σ, and
a is in the domain of M_1,
then
!e evaluates to $M_1(a)$ and changes the store from M to M_1 under σ.

$$\frac{\sigma, M \vdash e \Rightarrow a, M_1 \qquad a \in Domain(M_1)}{\sigma, M \vdash !e \Rightarrow M_1(a), M_1} \quad [\textsc{OpenBox}]$$

To get a box, e is evaluated. The result of e must be an address that belongs to M_1. The rule uses M_1 instead of M to find the value of the box. The reason is that e can create a new box and give the box as a result. Consider !(box 1). If the semantics is correct, this expression must evaluate to 1. The initial store is empty, but evaluating box 1 makes the store contain the address of the box. It means that the address can be obtained only by looking into M_1, not M. Therefore, the correct semantics uses M_1 to find the value of the box.

e_1:=e_2 is an expression changing the value of a box. e_1 determines the box to be updated, and e_2 determines the new value of the box. Just like when we open a box, e_1 must evaluate to a box. Otherwise, a run-time error will happen.

Rule SetBox
If
e_1 evaluates to a and changes the store from M to M_1 under σ, and
e_2 evaluates to v and changes the store from M_1 to M_2 under σ,
then
e_1:=e_2 evaluates to v and changes the store from M to $M_2[a \mapsto v]$ under σ.

$$\frac{\sigma, M \vdash e_1 \Rightarrow a, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow v, M_2}{\sigma, M \vdash e_1{:=}e_2 \Rightarrow v, M_2[a \mapsto v]} \text{ [SetBox]}$$

Like all the other expressions, an expression modifying a box uses the left-to-right order. If e_1 evaluates to an address a, the value associated with a in the store changes into the value denoted by e_2. Also, the value is the result of the whole expression. This semantics follows the semantics of many real-world imperative languages. For example, `x = 1` in C changes the value of `x` to `1` and results in `1`. On the other hand, functional languages usually use unit as the results of expressions for mutation. We can easily adopt the semantics in BFAE by adding unit to the language.

11.3 Interpreter

The following Scala code implements the syntax of BFAE: [4]

4: We omit the common part to FAE.

```
sealed trait Expr
...
case class NewBox(e: Expr) extends Expr
case class OpenBox(b: Expr) extends Expr
case class SetBox(b: Expr, e: Expr) extends Expr
case class Seqn(l: Expr, r: Expr) extends Expr
```

`NewBox(`e`)` represents box e; `OpenBox(`e`)` represents !e; `SetBox(`e_1`, `e_2`)` represents e_1:=e_2; `Seqn(`e_1`, `e_2`)` represents e_1; e_2.

Addresses should be defined. We treat addresses as integers in Scala.

```
type Addr = Int
```

In addition, we add a new variant of `Value` to represent boxes. [5]

5: We omit the common part to FAE.

```
sealed trait Value
...
case class BoxV(a: Addr) extends Value
```

`Box(`a`)` represents a.

We use a map to represent a store. The type of a store is `Map[Addr, Value]`.

```
type Sto = Map[Addr, Value]
```

`interp` takes an expression, an environment, and a store as arguments and returns a pair of a value and a store.

```
def interp(e: Expr, env: Env, sto: Sto): (Value, Sto) =
  e match {
    ...
  }
```

Let us see each case of the pattern matching.

```
case Num(n) => (NumV(n), sto)
case Id(x) => (env(x), sto)
case Fun(x, b) => (CloV(x, b, env), sto)
```

The `Num`, `Id`, and `Fun` cases use given stores as the results.

```
case Seqn(l, r) =>
  val (_, ls) = interp(l, env, sto)
  interp(r, env, ls)
case Add(l, r) =>
  val (NumV(n), ls) = interp(l, env, sto)
  val (NumV(m), rs) = interp(r, env, ls)
  (NumV(n + m), rs)
case Sub(l, r) =>
  val (NumV(n), ls) = interp(l, env, sto)
  val (NumV(m), rs) = interp(r, env, ls)
  (NumV(n - m), rs)
case App(f, a) =>
  val (CloV(x, b, fEnv), ls) = interp(f, env, sto)
  val (v, rs) = interp(a, env, ls)
  interp(b, fEnv + (x -> v), rs)
```

The `Seqn`, `Add`, `Sub`, and `App` cases do not directly modify or read stores, but pass the stores returned from the recursive calls to the next recursive calls or use them as results.

```
case NewBox(e) =>
  val (v, s) = interp(e, env, sto)
  val a = s.keys.maxOption.getOrElse(0) + 1
  (BoxV(a), s + (a -> v))
```

The `NewBox` case computes the initial value of the box first. Then, it computes an address not used in the store. We use the method `maxOption`. If a collection is empty, the method returns `None`. Otherwise, the result is `Some(n)`, where `n` is the greatest value in the collection. By `.getOrElse(0)`, we can get `n` from `Some(n)` and `0` from `None`. Consequently, `sto.keys.maxOption.getOrElse(0)` results in the maximum key in the store when the store is nonempty and `0` otherwise. `a` is one greater than that value and thus does not belong to the store. Therefore, we can use `a` as the address of the box. The result of the function consists of the address and the extended store.

```
case OpenBox(e) =>
  val (BoxV(a), s) = interp(e, env, sto)
  (s(a), s)
```

The `OpenBox` case evaluates the subexpression to get an address. If the result is not a box, an exception is thrown due to a pattern matching failure. The address is used to find the value of the box from the store. The result of the function consists of the value of the box and the store from the evaluation of the subexpression.

```
case SetBox(b, e) =>
  val (BoxV(a), bs) = interp(b, env, sto)
  val (v, es) = interp(e, env, bs)
  (v, es + (a -> v))
```

The `SetBox` case evaluates both subexpressions and modifies the store.

11.4 Exercises

1. Consider the following expression:

   $$(\lambda \mathsf{x}.(\lambda \mathsf{y}.\mathsf{x}{:=}8; !\mathsf{y})\ \mathsf{x})\ \mathsf{box}\ 7$$

   Write out the arguments to and results of `interp` each time it is called during the evaluation of the expression. Write them out in the order in which the calls to `interp` occur during evaluation. Write down the environments and stores using the `{}` notation. Also, use the arrow notation for both stores and environments. For example, the solution to the following `interp` call is like below.

   ```
   interp(Add(Id("x"), Id("y")),
          Map("x" -> NumV(3), "y" -> NumV(4)),
          Map.empty)

   exp: x + y
   env: {x -> NumV(3), y -> NumV(4)}
   sto: {}
   res: NumV(7) {}

   exp: x
   env: {x -> NumV(3), y -> NumV(4)}
   sto: {}
   res: NumV(3) {}

   exp: y
   env: {x -> NumV(3), y -> NumV(4)}
   sto: {}
   res: NumV(4) {}
   ```

12 Mutable Variables

BFAE of the previous chatper provides boxes. Boxes are good abstraction of mutable objects and data structures but do not explain mutable variables well. Boxes, mutable objects, mutable data structures are values, while mutable variables are names. Mutable variables allow the values associated with names to change. We can find the notion of a mutable variable in many real-world languages except a few functional languages including OCaml and Haskell.

The semantics of mutable variables seem trivial. We can change the values of mutable variables. However, if we use mutable variables with closures, we can do many interesting things. Consider the following Scala program:

```
def makeCounter(): () => Int = {
  var x = 0
  def counter(): Int = {
    x += 1
    x
  }
  counter
}

val counter1 = makeCounter()
val counter2 = makeCounter()

println(counter1())
println(counter2())
println(counter1())
println(counter2())
```

The program defines the function `makeCounter`. The function has a mutable variable `x` whose initial value is `0`. Also, it defines and returns the function `counter`. `counter` increases the value of `x` by one every time it is called. We make two counters by calling `makeCounter` twice. Then, we call each counter in turn and print the return value. What does the program print? The first value will be `1` since `counter1` will increase `x` by one from zero and return `x`. However, predicting the other ones is difficult. We need the exact semantics of mutable variables to answer the question.

This chapter defines MFAE by extending FAE with mutable variables. We will see the semantics of mutable variables. Addition of mutable variables gives us a chance to explore a different design of the function application semantics. We will see what is the call-by-reference semantics and how it differs from the call-by-value semantics.

12.1 Syntax

As variables are mutable in MFAE, we need to add expressions that change the values of variables. The following is the abstract syntax of MFAE: [1]

1: We omit the common part to FAE.

$$e ::= \cdots \mid x{:=}e$$

$x{:=}e$ is an expression changing the value of a variable. x is the variable to be updated; e determines the new value of the variable. Unlike $e_1{:=}e_2$ in BFAE, the left-hand-side of an assignment is restricted to a variable. The reason is that variables are not values. We cannot get a variable by evaluating an expression. The only way to designate a variable is to write the name of the variable, and the syntax reflects this point.

Note that MFAE lacks sequencing expressions, which exist in BFAE. Actually, it is not problematic at all. We can desugar sequencing expressions into lambda abstractions and function applications: transform $e_1; e_2$ into $(\lambda x.e_2)\ e_1$, where x is not free in e_2. The semantics of $e_1; e_2$ is that evaluating e_1 first and then e_2. The evaluation of $(\lambda x.e_2)\ e_1$ is the same. First, $\lambda x.e_2$ evaluates to a closure, which means that e_2 is not evaluated. Then, the argument, e_1, is evaluated. Finally, the body of the closure, e_2, is evaluated. Therefore, e_1 is evaluated before e_2. Also, since x is not a free identifier in e_2, the result of e_1 is never used even though it is passed to the function as an argument. Thus, we can conclude that the desugaring is correct. We may use sequencing expressions in examples as they can be easily desugared.

12.2 Semantics

Since MFAE provides mutation, its semantics uses store-passing just like BFAE. Therefore, a store is a finite partial function from addresses to values.

$$Sto = Addr \xrightarrow{\text{fin}} V$$

$$M \in Sto$$

The semantics is a relation over Env, Sto, E, V, and Sto.

$$\Rightarrow \subseteq Env \times Sto \times E \times V \times Sto$$

Like FAE, a value of MFAE is either an integer or a closure. It is different from BFAE, which allows addresses to be values. BFAE treats addresses as values becuase expressions creating boxes evaluate to addresses. However, there are mutable variables instead of boxes in MFAE. MFAE has addresses to support mutation, but they are used only for tracking the value of each variable. Addresses are not exposed to programmers as values.

An environment of MFAE is a finite partial function from identifiers to addresses, but not values.

$$Env = Id \xrightarrow{\text{fin}} Addr$$

$$\sigma \in Env$$

The semantics needs environments to find the value denoted by a variable. FAE, whose variables are immutable, is satisfied with environments that take identifiers as input and return values. However, variables of MFAE are mutable. Evaluation outputs a value and a store. Environments are not the output of evaluation. Therefore, we cannot use environments to record changes in the values of variables. On the other hand, we can use stores to record the changes as stores are output of evaluation. It implies that stores must contain the values of variables to make variables mutable. Since the value of a certain variable is stored at a particular address of a store, an environment must know the address of each variable.

One may ask if we can remove environments from the semantics and consider a store as a partial function from identifiers to values. However, in fact, removing environments from the semantics prevents use of static scope. Assume that the semantics lacks environments, and a store is a partial function from an identifier to a value. Consider $((\lambda \mathsf{x}.\mathsf{x}:=1)\ 0); \mathsf{x}$. To evaluate the function application, the value of the argument should be recorded in the store. After evaluating the function body, the store will be passed to the evaluation of x. Then, x evaluates to 1 without a run-time error since x is in the store. On the contrary, under static scope, the scope of x includes only $x := 1$. The expression should result in a run-time error because x outside the function is a free identifier. Environments are essential for resolving this problem. Environments enable static scope, and stores make variables mutable. The semantics must have both.

Because of the change in the definition of an environment, the semantics of identifiers need to be revised. An environment has the address of a given identifier, and a store has the value at a given address. Therefore, we need two steps to find the value of a variable: find the address of a variable from the environment and find the value at the address from the store.

Rule Id
If
x is in the domain of σ, and
$\sigma(x)$ is in the domain of M,
then
x evaluates to $M(\sigma(x))$ and changes the store from M to M under σ.

$$\frac{x \in Domain(\sigma) \qquad \sigma(x) \in Domain(M)}{\sigma, M \vdash x \Rightarrow M(\sigma(x)), M} \quad [\text{Id}]$$

Like boxes in BFAE, each variable of MFAE has its own address. New variables can be defined only by function applications. Hence, function applications are the only expressions that create new addresses. Let us see the semantics of function applications.

Rule App

If
e_1 evaluates to $\langle \lambda x.e, \sigma' \rangle$ and changes the store from M to M_1 under σ,
e_2 evaluates to v' and changes the store from M_1 to M_2 under σ,
a is not in the domain of M_2, and
e evaluates to v and changes the store from $M_2[a \mapsto v']$ to M_3 under $\sigma'[x \mapsto a]$,
then
$e_1\ e_2$ evaluates to v and changes the store from M to M_3 under σ.

$$\frac{\sigma, M \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow v', M_2 \qquad a \notin \mathit{Domain}(M_2) \qquad \sigma'[x \mapsto a], M_2[a \mapsto v'] \vdash e \Rightarrow v, M_3}{\sigma, M \vdash e_1\ e_2 \Rightarrow v, M_3} \text{ [App]}$$

The evalution of the subexpressions is the same as BFAE. However, the remaining procedure is different. We cannot store the value of the argument in the environment. It should go into the store. To put the value into the store, we need a fresh address. The name of the parameter becomes associated with the address in the environment; the address becomes associated with the value of the argument in the store. Finally, the function body is evaluated.

Changing the value of a variable is similar to changing the value of a box of BFAE. However, we have to evaluate an expression to find the address of the box to be updated in BFAE. On the other hand, we can find the address of the variable to be updated from the environment by using its name in MFAE.

Rule Set
If
x is in the domain of σ, and
e evaluates to v and changes the store from M to M_1 under σ,
then
x:=e evaluates to v and changes the store from M to $M_1[\sigma(x) \mapsto v]$ under σ.

$$\frac{x \in \mathit{Domain}(\sigma) \qquad \sigma, M \vdash e \Rightarrow v, M_1}{\sigma, M \vdash x{:=}e \Rightarrow v, M_1[\sigma(x) \mapsto v]} \text{ [Set]}$$

We can reuse the rules of BFAE for the other expressions.

Now, we can answer the question in the beginning of the chapter. At each call to `makeCounter`, a new address is allocated to store the value of `x`. Therefore, `x` of `counter1` uses a different address from `x` of `counter2`. Both of the first two lines of `println` print `1`. Also, each address is permanent throughout the execution. When a call to `counter1` updates the value of `x`, the change remains until the next call to `counter1`. Thus, both of the last two lines of `println` print `2`.

12.3 Interpreter

The following Scala code implements the syntax of MFAE: [2]

2: We omit the common part to FAE.

```
sealed trait Expr
...
case class Set(x: String, e: Expr) extends Expr
```

`Set(`x`, `e`)` represents x:=e.

The types of an address and a store can be defined as in BFAE.

```
type Addr = Int
type Sto = Map[Addr, Value]
```

We need to change the type of an environment.

```
type Env = Map[String, Addr]
```

As in BFAE, `interp` takes an expression, an environment, and a store as arguments and returns a pair of a value and a store. [3]

3: We omit the common part to BFAE.

```
def interp(e: Expr, env: Env, sto: Sto): (Value, Sto) =
  e match {
    ...
    case Id(x) => (sto(env(x)), sto)
    case App(f, a) =>
      val (CloV(x, b, fEnv), ls) = interp(f, env, sto)
      val (v, rs) = interp(a, env, ls)
      val addr = rs.keys.maxOption.getOrElse(0) + 1
      interp(b, fEnv + (x -> addr), rs + (addr -> v))
    case Set(x, e) =>
      val (v, s) = interp(e, env, sto)
      (v, s + (env(x) -> v))
  }
```

In the `Id` case, the function finds the address of the variable first and then the value at the address.

In the `App` case, we use the same strategy to the interpreter of BFAE to compute a new address. The body of the function is evaluated under the extended environment and the extended store.

The `Set` case uses the environment to find the address of the variable. Then, it updates the store to change the value of the variable.

12.4 Call-by-Reference

Novices in programming often implement a swap function incorrectly. For example, consider the following C++ program:

```
void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

```
int a = 1, b = 2;
swap(a, b);
std::cout << a << " " << b << std::endl;
```

They expect the program to print `2 1` as `swap` has been called. On the contrary, their expectation is wrong. The result is `1 2`. We can explain the reason based on the content of this chapter. When `swap` is called, two new fresh addresses are allocated for `x` and `y`. The values of `a` and `b` are copied and stored in the addresses, respectively. The function affects only the values in the addresses of `x` and `y`. It never touches the addresses of `a` and `b`. As a consequence, while the values of `x` and `y` are swapped, the values of `a` and `b` are not.

This is the usual semantics of function applications. The values of arguments are copied and saved at fresh addresses. This semantics is called *call-by-value* (CBV) as function calls pass the values of arguments.

People have explored another semantics for function applications to implement functions like `swap` easily. The semantics is called *call-by-reference* (CBR). In this semantics, function calls pass the references, i.e. addresses, when variables are used as arguments.

The following rule defines the semantics of a function application using CBR when its argument is a variable:

Rule App-Cbr
If
e_1 evaluates to $\langle \lambda x'.e', \sigma' \rangle$ and changes the store from M to M_1 under σ,
x is in the domain of σ, and
e' evaluates to v and changes the store from M_1 to M_2 under $\sigma'[x' \mapsto \sigma(x)]$,
then
$e\ x$ evaluates to v and changes the store from M to M_2 under σ.

$$\frac{\sigma, M \vdash e_1 \Rightarrow \langle \lambda x'.e', \sigma' \rangle, M_1 \qquad x \in \mathit{Domain}(\sigma) \qquad \sigma'[x' \mapsto \sigma(x)], M_1 \vdash e' \Rightarrow v, M_2}{\sigma, M \vdash e\ x \Rightarrow v, M_2} \quad [\text{App-Cbr}]$$

The rule does not evaluate the argument to get a value. It simply uses the address of the variable. Then, the parameter of the function has the exactly same address to the argument. Any change in the parameter that happens in the function body affects the variable outside the function. We say that the parameter is an *alias* of the argument as they share the same address.

Even if we want to adopt the CBR semantics in MFAE, we cannot use it when the argument is not a variable. We cannot get an address from an expression that is not a variable. In such cases, we fall back to the CBV semantics. The following rule specifies such cases:

Rule App-Cbv

If
  e_1 evaluates to $\langle \lambda x.e, \sigma' \rangle$ and changes the store from M to M_1 under σ,
  e_2 is not an identifier,
  e_2 evaluates to v' and changes the store from M_1 to M_2 under σ,
  a is not in the domain of M_2, and
  e evaluates to v and changes the store from $M_2[a \mapsto v']$ to M_3 under $\sigma'[x \mapsto a]$,
then
  $e_1\ e_2$ evaluates to v and changes the store from M to M_3 under σ.

$$\frac{\sigma, M \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle, M_1 \qquad e_2 \notin Id \qquad \sigma, M_1 \vdash e_2 \Rightarrow v', M_2 \qquad a \notin Domain(M_2) \qquad \sigma'[x \mapsto a], M_2[a \mapsto v'] \vdash e \Rightarrow v, M_3}{\sigma, M \vdash e_1\ e_2 \Rightarrow v, M_3} \quad [\textsc{App-Cbv}]$$

It is the same as Rule App except that it has one more premise to ensure that the argument is not a variable.

The interpreter needs the following change:

```
case App(f, a) =>
  val (CloV(x, b, fEnv), ls) = interp(f, env, sto)
  a match {
    case Id(y) =>
      interp(b, fEnv + (x -> env(y)), ls)
    case _ =>
      val (v, rs) = interp(a, env, ls)
      val addr = rs.keys.maxOption.getOrElse(0) + 1
      interp(b, fEnv + (x -> addr), rs + (addr -> v))
  }
```

It uses pattern matching on the argument expression of a function application. When it is an identifier, the CBR semantics can be used. Otherwise, it falls back to the CBV semantics.

We can find a few languages that support CBR in real-world. One example is C++. In C++, if there is an ampersand in front of the name of a parameter, the parameter uses the CBR semantics. We can fix the function `swap` with this feature.

```
void swap(int &x, int &y) {
    int tmp = x;
    x = y;
    y = tmp;
}

int a = 1, b = 2;
swap(a, b);
std::cout << a << " " << b << std::endl;
```

It is enough to fix only the first line to make the parameters use CBR. When `swap` is applied to `a` and `b`, the addresses of `a` and `b` are passed. The address of `x` is the same as that of `a`, and the address of `y` is the same as that of `b`. Therefore, the function swaps not only the values of `x` and `y` but also the values of `a` and `b`. The program prints `2 1` as intended.

12.5 Exercises

1. This exercise extends MFAE with pointers. Consider the following language:

$$e ::= \cdots \mid *e \mid \&x \mid *e{:=}e$$
$$v ::= \cdots \mid a$$

The semantics of some constructs are as follows:

- The value of $*e$ is the value in the store at the address denoted by the expression.
- The value of $\&x$ is the address denoted by the identifier in the environment.
- The evaluation of $*e_1{:=}e_2$ evaluates e_2 first, which is the value of the whole expression. Then, it evaluates e_1, and it maps the address denoted e_1 to the value of e_2.

Write the operational semantics of the form $\boxed{\sigma, M \vdash e \Rightarrow v, M}$ for the expressions.

2. The following code is an excerpt from the implementation of the interpreter for MFAE:

```
def interp(e:Expr, env:Env, sto:Sto): (Value, Sto) =
  e match {
    ...
    case App(f, a) =>
      val (CloV(x, b, fEnv), ls) = interp(f, env, sto)
      a match {
        case Id(y) =>
          interp(b, fEnv + (x -> env(y)), ls)
        case _ =>
          val (v, rs) = interp(a, env, ls)
          val addr = rs.keys.maxOption.getOrElse(0) + 1
          interp(b, fEnv + (x -> addr), rs + (addr -> v))
      }
  }
```

a) What is this semantics? CBV or CBR?

Consider the following expression in MFAE:

$$\texttt{val n=42 in val f=}\lambda\texttt{g.(g n) in (f (}\lambda\texttt{x.x + 8))}$$

b) Show the environment and store just before evaluating addition in the CBV semantics.

c) Show the environment and store just before evaluating addition in the CBR semantics.

13 Lazy Evaluation

This chapter is about lazy evaluation. *Lazy evaluation* means delaying the evaluation of an expression until the result is required. The opposite of lazy evaluation is *eager evaluation*, which evaluates an expression even if in the case that it is unknown whether the result of the expression is necessary for future computation. There are multiple features that lazy evaluation can be applied to in programming languages. For example, arguments for function applications can be evaluated lazily; each argument of a function application is evaluated when the argument is used in the function body, not before the evaluation of the function body starts. Another example is a variable definition. The initialization of variable happens when the variable is used for the first time, not when it is defined. Actually, as you have seen already, local variables can be consdiered as syntactic sugar of function parameters, it is enough to focus on laziness in function applications. Therefore, this book considers lazy evaluation only on arguments of function applications.

All the previously defined languages in the book are eager languages. They use the CBV semantics for function applications. CBV can be consdiered equivalent to eager evaluation. CBV means that every argument is passed as a value. The value of an argument can be acquired only by evaluating the argument expression. It implies that every argument is evaluated before the evaluation of the function body regardless of whether the argument is used during the body evaluation. Thus, the CBV semantics is equal to the eager evaluation semantics.[1]

This chapter mainly discusses *call-by-name* (CBN) semantics, which is one form of lazy evaluation. In the CBN semantics, each argument is passed as its name, i.e. the expression itself, rather than the value denoted by the expression. Since it is passed as an expression, there is no need to evaluate the expression to obatin its value. The expression will be evalauted when the value is required by the function body. As you can see, CBN delays the computation of arguments by passing them as expressions and can be considered as lazy evaluation. We will define LFAE, which is a lazy version of FAE, in this chapter. LFAE adopts the CBN semantics. In addition, we will introduce call-by-need,[2] which is another form of lazy evaluation, as an optimized version of CBN.

Before we discuss the CBN semantics and LFAE, let us see why lazy evaluation is valuable in practice. We can find lazy evaluation in a few real-world languages. Haskell is well-known as treating every computation lazily by default.[3] On the other hand, Scala is an eager language but allows programmers to selectively apply lazy evaluation to their programs. We will see code examples in Scala as it is the language used in this book, though Haskell is the most famous lazy language. Consider the following Scala code:

```
def square(x: Int): Int = {
  Thread.sleep(5000)  // models some expensive computation
```

1: One may wonder whether CBR is eager or not. The best answer is that CBR is irrelevant to distinction between eagerness and laziness. As shown in the previous chapter, CBR is possible only when an argument is a variable, whose address is known. In that case, there is nothing to evaluate. We have to make a choice between passing the address (CBR) and passing the value at the address (CBV). On the other hand, choosing one of eagerness and laziness is about a choice between evaluating the argument and not evaluating the argument. CBR can be used in both eager and lazy languages.

2: Some people use the term lazy evaluation to denote call-by-need only. In that sense, CBN is not considered as lazy evaluation. However, this book views lazy evaluation as a term that can be used broadly to mean any form of delayed computation. In this sense, both CBN and call-by-need belong to lazy evaluation.

3: Programmers can force evaluation if they really want.

```
  x * x
}

def foo(x: Int, b: Boolean): Int =
  if (b) x else 0

val file = new java.io.File("a.txt")
foo(square(10), file.exists)
```

The function `square` takes an integer as an argument and returns its square. It always takes five seconds to return due to `Thread.sleep(5000)`, which makes the thread sleep for five seconds. Of course, no one will write such code in practice, but it is an analogy of highly expensive computation that takes a long time.

The function `foo` takes one integer and one boolean. It returns the integer when the boolean is true and zero otherwise. Therefore, the integer value is required only when the boolean is true.

The last line of the program applies `foo` to `square(10)` and `file.exists`. The second argument is true if and only if there exists a file named `a.txt`. If the file does not exist, `foo` returns zero, and thus the value of `square(10)` is unnecessary. However, as Scala uses eager evaluation by default, `square(10)` is evaluated and spends five seconds regardless of the existence of the file. If we modify the program not to evaluate `square(10)` when the file is absent, we can save time in many cases without changing the behavior of the program.

Lazy evaluation gives us an easy solution to this issue. If the first argument for `foo` is evaluated lazily, the program will evaluate `square(10)` only when the file exists. In Scala, we can make a certain parameter use the CBN semantics by adding `=>` in front of the type of the parameter. Thus, the following fix completely resolves the problem:

```
def foo(x: => Int, b: Boolean): Int =
  if (b) x else 0
```

We call `x` a by-name parameter in Scala. Since `x` is a by-name parameter, the first argument for `foo` is evaluated only when the value of `x` is needed during the evaluation of the body.

13.1 Semantics

We do not explain the syntax of LFAE as it is the same as FAE. We can move on to the semantics immediately. The definition of an environment is the same as FAE. Also, as in FAE, $\sigma \vdash e \Rightarrow v$ is true if and only if e evaluates to v under σ. We can use Rule Num of LFAE since evaluation of integers are not affected by lazy evaluation. Similarly, Rule Fun can be reused as well. Rule Id also remains the same. The value of an identifier can be found in an environment.

One that certainly requires a change is the semantics of function applications. It is the most distinctive feature of lazy languages compared to eager languages. In the CBV semantics, we store the values of arguments

in environments and use the environments to evaluate function bodies. We still need to store arguments in environments in the CBN semantics. However, arguments are passed as expressions, and expressions are not values. We need a way to put expressions in environments. The simplest solution is to define a new kind of values as follows: [4]

4: We omit the common part to FAE.

$$v ::= \cdots \mid (e, \sigma)$$

(e, σ) is an expression as a value; we call it an expression-value. It denotes that the computation of e under σ has been delayed. We must keep the environment together with the expression since the expression can have free identifiers whose values are available in the environment. The reason is the same as why we need the notion of a closure, which is a function with an environment. The structure of (e, σ) is quite similar to the structure of a closure, $\langle \lambda x.e, \sigma \rangle$, except that an expression-value lacks the name of a parameter. The similarity is not just a coincidence. Both kinds of values denote delay of computation. The evaluation of e in (e, σ) is postponed until the value of the argument becomes required, and the evaluation of e in $\langle \lambda x.e, \sigma \rangle$ is postponed until the closure is applied to a value.

Because of the addition of expression-values, we need to define another form of evaluation. We call it strict evaluation. The purpose of strict evaluation is to force an expression-value to be a "normal" value, which is an integer or a closure. Strict evaluation is required because the ability of an expression-value is limited. It can be passed as an argument or stored in an environment like normal values but cannot be used as an operand of an arithmetic expression or applied to a value as a function. There must be a way to convert an expression-value to a normal value, and strict evaluation takes this role.

Strict evaluation is defined as a binary relation over V and V. We use $\Downarrow$ to denote strict evaluation.

$$\Downarrow \subseteq V \times V$$

$v_1 \Downarrow v_2$ is true if and only if v_1 strictly evaluates to v_2. Here, v_1 can be any value. However, v_2 cannot be an expression-value; it must be a normal value. The reason obviously comes from the purpose of strict evaluation: converting an expression-value to a normal value.

The following rules define strict evaluation of normal values:

Rule Strict-Num
n strictly evaluates to n

$$n \Downarrow n \quad \text{[Strict-Num]}$$

Rule Strict-Clo
$\langle \lambda x.e, \sigma \rangle$ strictly evaluates to $\langle \lambda x.e, \sigma \rangle$

$$\langle \lambda x.e, \sigma \rangle \Downarrow \langle \lambda x.e, \sigma \rangle \quad \text{[Strict-Clo]}$$

A normal value strictly evaluates to itself since it is already a normal value.

The following rule defines strict evaluation of expression-values:

Rule Strict-Expr
If e evaluates to v_1 under σ, and v_1 strictly evaluates to v_2, then (e, σ) strictly evaluates to v_2.

$$\frac{\sigma \vdash e \Rightarrow v_1 \qquad v_1 \Downarrow v_2}{(e, \sigma) \Downarrow v_2} \quad \text{[Strict-Expr]}$$

An expression-value (e, σ) is strictly evaluated by evaluating e under σ. The result of e can be an expression-value again, and thus we need repeated strict evaluation until reaching a normal value.

Now, we can define the semantics of function applications by using the notions of expression-values and strict evaluation.

Rule App
If
e_1 evaluates to v_1 under σ,
v_1 strictly evaluates to $\langle \lambda x.e, \sigma' \rangle$, and
e evaluates to v under $\sigma'[x \mapsto (e_2, \sigma)]$,
then
$e_1\ e_2$ evaluates to v under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow \langle \lambda x.e, \sigma' \rangle \qquad \sigma'[x \mapsto (e_2, \sigma)] \vdash e \Rightarrow v}{\sigma \vdash e_1\ e_2 \Rightarrow v} \quad \text{[App]}$$

e_1 evaluates to v_1 first. v_1 may be an expression-value, while we need a closure. Therefore, we strictly evaluate v_1 to get a closure. On the other hand, in the CBN semantics, the argument must not be evaluated before the evaluation of the function body. Instead of evaluating e_2, we make an expression-value with e_2 and σ and then put the value into the environment.

Let us see the semantics of addition and subtraction.

Rule Add
If
e_1 evaluates to v_1 under σ,
v_1 strictly evaluates to n_1,
e_2 evaluates to v_2 under σ, and
v_2 strictly evaluates to n_2,
then
$e_1 + e_2$ evaluates to $n_1 + n_2$ under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad \text{[Add]}$$

Rule Sub

If
  e_1 evaluates to v_1 under σ,
  v_1 strictly evaluates to n_1,
  e_2 evaluates to v_2 under σ, and
  v_2 strictly evaluates to n_2,
then
  $e_1 - e_2$ evaluates to $n_1 - n_2$ under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 - e_2 \Rightarrow n_1 - n_2} \text{ [Sub]}$$

There is nothing difficult. They are similar to the rules of FAE but additionally require strict evaluation since addition and subtraction are possible only by using integers, not expression-values.

The semantics is a correct instance of CBN but has a flaw from a practical perspective. Consider $(\lambda x.x)\ (1 + 1)$. It results in $(1 + 1, \emptyset)$, not 2. Most programmers are likely to perfer 2 as a result. We need to apply one last strict evaluation at the end of the evaluation to resolve the problem. It is to say that "the result of a program e is v when $\emptyset \vdash e \Rightarrow v'$ and $v' \Downarrow v$." Note that it is different from applying strict evaluation to the evaluation of every expression in the program. Strict evaluation is applied to only the result of the whole expression, which is the program. In this way, we can make the result of the above expression 2 and eliminate the flaw.[5]

5: It is not a flaw in real-world programming languages like Haskell. A program shows its result by output operations (e.g. to files) rather than the value of a single expression. Each output operation applies strict evaluation to its argument (like Rule Add, Rule Sub, and Rule App in LFAE), and the value of each expression does not need to be a normal value.

If evaluating an expression in the CBV semantics results in a value, then the CBN semantics yields the same value. It is known as a corollary of the standardization theorem of lambda calculus [Rey09]. Note that it is true only in languages without side effects. The result of an expression with side effects varies in the order of the evaluation. For example, if an argument is an expression changing the value of a box, and the body of the function reads the value of the box without using the argument, the program can behave differently in CBV and CBN. In CBV, the read value will be the value after the update. On the other hand, in CBN, the update never happens, and the read value will be the original value of the box.

While the CBN semantics preserves the results of the CBV semantics, the converse is false even without mutation, i.e. there are expressions that yield results only in CBN. For instance, consider a function application whose argument is a nonterminating expression. If the function returns zero without using the argument, evaluation with CBN results in zero, while evaluation with CBV does not terminate.

13.2 Interpreter

We need to add a new variant to `Value` to represent expression-values. [6]

6: We omit the common part to FAE.

```
sealed trait Value
...
case class ExprV(e: Expr, env: Env) extends Value
```

`ExprV(`e`, `σ`)` represents (e, σ).

The following function implements strict evlauation:

```
def strict(v: Value): Value = v match {
  case ExprV(e, env) => strict(interp(e, env))
  case _ => v
}
```

We can implement `interp` as follows: [7]

7: We omit the common part to FAE.

```
def interp(e: Expr, env: Env): Value = e match {
  ...
  case Add(l, r) =>
    val NumV(n) = strict(interp(l, env))
    val NumV(m) = strict(interp(r, env))
    NumV(n + m)
  case Sub(l, r) =>
    val NumV(n) = strict(interp(l, env))
    val NumV(m) = strict(interp(r, env))
    NumV(n - m)
  case App(f, a) =>
    val CloV(x, b, fEnv) = strict(interp(f, env))
    interp(b, fEnv + (x -> ExprV(a, env)))
}
```

Each case matches the corresponding rule, so there is nothing difficult.

13.3 Call-by-Need

The current implementation is efficient when a parameter appears once or less in the function body. However, using a parameter twice or more leads to redundant calculation. Consider the following Scala program:

```
def square(x: Int): Int = {
  Thread.sleep(5000)  // models some expensive computation
  x * x
}

def bar(x: => Int, b: Boolean): Int =
  if (b) x + x else 0

val file = new java.io.File("a.txt")
bar(square(10), file.exists)
```

`x` appears twice in the body of `bar`. If `a.txt` exists, `square(10)` is evaluated twice. Actually, we do not need to evaluate `square(10)` twice since its result is always the same. Since `square` is an expensive function, it is desirable to reduce the number of function calls as much as possible. If we use CBV instead of CBN, it is possible to evaluate `square(10)` only once when the file exists. However, going back to CBV is not a good choice. It will make the program evaluate `square(10)` even when the file does not exist.

The way to solve this problem is to store the value of an argument and use the value again. This strategy is as optimal as CBV when a parameter appears multiple times; it is as optimal as CBN when a parameter is not used at all. For programmers, it is tedious to implement such logic in their programs by themselves. Instead, programming languages can provide the optimization. This optimization is called *call-by-need* as each argument is evaluated based on need for its value. It is evaluated once if needed and is not otherwise.

Call-by-need is not different semantics from CBN in purely functional languages. The behaviors of a program in call-by-need and CBN are completely equal. Call-by-need is just an optimization strategy of interpreters and compilers. On the other hand, call-by-need is different semantics from CBN in languages with side effects. In such languages, the number of computation of a certain expression can affect the result. For example, consider an argument that is an expression that increases the value of the box by one. Suppose that its value is used twice in the function body. Then, the value of the box increases by two in CBN, while it increases by one in call-by-need.

Since LFAE lacks side effects, we can adopt call-by-need to the language as optimization of the interpreter. There is no need to newly define the call-by-need version of the semantics.

To store the strict value of an expression-value, we add a new field to the class `ExprV`.

```
case class ExprV(
  e: Expr, env: Env, var v: Option[Value]
) extends Value
```

The field is declared as mutable. Initially, the value of the expression is unknown, and `v` equals `None`. When the value is calculated for the first time, the value is stored in `v`. The fact that `v` equals `Some(a)` for some `a` implies that the value of the expression is `a`. In the next time we need the value again, `a` can be used without any redundant computation.

The function `strict` requires the following change:

```
def strict(v: Value): Value = v match {
  case ExprV(_, _, Some(cache)) => cache
  case ev @ ExprV(e, env, None) =>
    val cache = strict(interp(e, env))
    ev.v = Some(cache)
    cache
  case _ => v
}
```

It checks whether there exists a cached value. If it is the case, the function simply returns the cached value. Otherwise, `e` is evaluated under `env` like before. In addition, the function stores the value in `v`.

The function `interp` needs only one fix. When a new `ExprV` instance is created in the `App` case, one additional argument is required to initialize the field `v`.

```
case App(f, a) =>
  val CloV(x, b, fEnv) = strict(interp(f, env))
  interp(b, fEnv + (x -> ExprV(a, env, None)))
```

Since we do not know the value of `a`, the initial value of `v` is `None`.

Purely functional languages with lazy evaluation usually adopt call-by-need because it is just optimization but not a change in their semantics. On the other hand, impure languages cannot consider call-by-need as optimization and often allow programmers to choose one of them at each place. For example, Scala uses CBN for by-name parameters and call-by-need for lazy variables. We can define lazy variables with the `lazy` modifier.

```
lazy val x = {
  println(1)
  1
}
val y = x + x
```

The program prints `1` only once. By using both by-name parameter and lazy variable, we can simulate the call-by-need semantics in Scala.

```
def bar(_x: => Int, b: Boolean): Int = {
  lazy val x = _x
  if (b) x + x else 0
}
```

If `b` is true, the first argument is evaluated only once. Otherwise, it is not evaluated at all.

13.4 Exercises

1. Which of the following produce different results in a CBV language and a CBN language? Both produce the same result if they both produce the same number or they both produce closures (even if they do not behave exactly the same when applied).

   a) $(\lambda y.y\ 3)\ (\lambda x.1\ 2)$
   b) $(\lambda y.y\ \lambda x.10)\ (\lambda x.x\ (1\ 2))$
   c) $(\lambda y.y\ \lambda x.10)\ (\lambda x.1\ 2)$
   d) $(\lambda y.y)\ (1 + \lambda x.x)$
   e) $(\lambda y.1 + 2)\ (1 + \lambda x.x)$

2. Show the results of each expression in a CBV language and a CBN language.

   a) $(\lambda x.8) + 10$
   b) $(\lambda x.8)\ (1\ 2)$
   c) $\lambda x.((\lambda y.42)\ (9\ 2))$
   d) $1 + ((\lambda x.x + 13)\ (1 + \lambda y.7))$
   e) $1 + ((\lambda x.1 + 13)\ (1 + \lambda y.7))$

3. Note that there is a recursive call in the following function:

```
def strict(v: Value): Value = v match {
  case ev @ ExprV(e, env, None) =>
    val cache = strict(interp(e, env))
    ev.v = Some(cache)
    cache
  case ExprV(_, _, Some(cache)) => cache
  case _ => v
}
```

Write an example LFAE expression showing the need for the recursive `strict` call.

4. Consider the following expression:

$$\begin{array}{l}\text{val f=}\lambda\text{x.y} + 7 \text{ in}\\ \text{val y=5 in}\\ \text{f } (42 + \lambda\text{y.3})\end{array}$$

Explain the result of evaluating it under the following semantics:

a) Lazy evaluation with static scope
b) Lazy evaluation with dynamic scope
c) Eager evaluation with static scope
d) Eager evaluation with dynamic scope

5. For each of the following LFAE expressions, show how it is evaluated and its result.

a) $(\lambda\text{x.x } 8)\ (42\ + \lambda\text{y.y} - \text{x})$
b) $(\lambda\text{y.}(\lambda\text{x.3} + \text{x})\ \text{y})\ 5$

Continuations 14

Many real-world languages support *control diverters*, which alter *control flows* of programs. For example, programmers can use `return` in Scala code. Consider the following Scala program:

```
def foo(x: Int): Int = {
  val y = return x
  x + x
}

foo(3)
```

Its result is `3`, not `6`. When `return x` is evaluated, the value denoted by `x` is immediately returned by the function. `x + x` is never evaluated. It shows how `return` is different from many other expressions, including addition and function application. Most expressions produce values as results. However, `return` changes the control flow by making the function immediately return. We can find various control diverters other than `return` in real-world languages: `break`, `continue`, `goto`, `throw` (or `raise`), and so on.

Control diverters are useful for writing programs with complex control flows. For instance, consider a function `numOfWordsInFile` that takes the name of a file and a string as arguments and returns how many times the string occurs in the file.

```
def numOfWordsInFile(name: String, word: String): Int = ...
```

If such a file does not exist, the function returns `-1`. When the file is read for the first time, its content is cached, so the function must check the cache first to reuse the cached result if available.

Assume that we have the following helper functions:

```
// checks whether a cached result exists
def cached(name: String): Boolean

// gets the cached result
def getCache(name: String): String

// checks whether a given file exists
def exists(name: String): Boolean

// reads the content of the file and caches it
def read(name: String): String

// counts the number of occurrences of ‘word‘ in ‘content‘
def numOfWords(content: String, word: String): Int
```

Then, we can implement `numOfWordsInFile` with the helper functions and `return`.

```
def numOfWordsInFile(name: String, word: String): Int = {
  val content =
    if (cached(name))
      getCache(name)
    else if (exists(name))
      read(name)
    else
      return -1
  numOfWords(content, word)
}
```

First, the function checks whether there is a cached result by calling `cached`. If so, the cached result is acquired with `getCache` and stored in the variable `content`. Otherwise, the function checks whether the file exists with `exists`. The file is read with `read` if the file exists, and its content is stored in `content`. When the file is missing, the function immediately returns `-1` with `return -1`. Finally, the function counts the number of `word` in `content` with `numOfWords` to compute the result. Note that when both cached result and file are absent, `numOfWords` is never called because `return -1` terminates the function beforehand.

Without `return`, we need to call `numOfWords` in multiple places.

```
def numOfWordsInFile(name: String, word: String): Int = {
  if (cached(name))
    numOfWords(getCache(name), word)
  else if (exists(name))
    numOfWords(read(name), word)
  else
    -1
}
```

The code looks fine, but we cannot directly express the idea that the function needs to call `numOfWords` except one errorneous case, where both cache and file are not found. It is not a big flaw in the current implementation of `numOfWordsInFile`. However, if we write a function with a large number of conditions, we would prefer `return` (Figure 14.1) to call `numOfWords` multiple times (Figure 14.2).

Like mutation, control diverters make languages impure. In pure languages, the order of evaluation does not matter. Each expression only produces a result; there is no other *side effect*. On the other hand, in impure languages, the order of evaluation matters. Expressions can perform side effects, including mutation and control flow changes. Evaluting a certain expression can change the result of other expressions or make other expressions not evaluated. Therefore, programs written in impure languages require global reasoning, while programs wirtten in pure languages require local, modular reasoning. Mutation and control diverters make the reasoning of programs difficult despite their usefulness. Control diverters must be used with extra care of programmers.

```
def numOfWordsInFile(name: String, word: String): Int = {
  val content =
    if (A)
      a
    else if (B)
      b

    ...

    else if (F)
      f
    else
      return -1
  numOfWords(content, word)
}
```

Figure 14.1: `numOfWordsInFile` with `return`

```
def numOfWordsInFile(name: String, word: String): Int = {
  if (A)
    numOfWords(a, word)
  else if (B)
    numOfWords(b, word)

  ...

  else if (F)
    numOfWords(f, word)
  else
    -1
}
```

Figure 14.2: `numOfWordsInFile` without `return`

This chapter and the following two chapters introduce continuations, which are the most general explanation of control flow. This chapter focuses on what continuations are and how we can write programs and define semantics while exposing continuations explicitly. Then, the next chapter explains how we can add control diverters to languages based on the notion of a continuation.

14.1 Redexes and Continuations

Evaluating an expression requires one or more steps of computation. Consider (1 + 2) + (3 + 4). Evaluation of this expression consists of the following seven steps of computation:[1]

1. Evaluate the expression 1 to get the integer value 1.
2. Evaluate the expression 2 to get the integer value 2.
3. Add the integer value 1 to the integer value 2 to get the integer value 3.
4. Evaluate the expression 3 to get the integer value 3.
5. Evaluate the expression 4 to get the integer value 4.
6. Add the integer value 3 to the integer value 4 to get the integer value 7.
7. Add the integer value 3 to the integer value 7 to get the integer

1: Assume that we use the left-to-right order for subexpression evaluation in this chapter.

value 10.

We can split N steps of computation into two parts: the former n steps and the remaining $N - n$ steps. We call the expression evaluated by the former n steps a *redex*[2] and the remaining computation described by the latter $N - n$ steps the *continuation* of the redex.

2: The term redex stands for a **red**ucible **ex**pression. However, we introduce the notion of a redex without explaining what "reduction" or "reducible" is. The notion can be understood without knowing what reduction is, so we do not care about the origin of the term.

For example, if we split the above steps into step 1 and steps 2-7, then the redex is the expression 1, and the continuation consists of the remaining steps. The important point is that the continuation requires the result of the redex to complete the evaluation. Without 1, the result of step 1, the continuation cannot proceed beyond step 3. Step 3 can be accomplished only when the result of the redex is provided. Therefore, we can consider the continuation as an expression with a hole that must be filled with the result of a redex. Intuitively, the continuation of 1 can be written as $(\square + 2) + (3 + 4)$, where $\square$ denotes the place in which the result of the redex is used. Since the continuation takes the result of a redex as input and completes the remaining computation, the continuation can be interpreted as a function. Following this interpretation, we can express the continuation of 1 as $\lambda x.(x + 2) + (3 + 4)$.

There are multiple ways to split the steps. The following table shows three different ways of splitting the evaluation of $(1 + 2) + (3 + 4)$ to find a redex and the continuation.

Redex		Continuation		
Steps	Expression	Steps	Hole	Function
1	1	2-7	$(\square + 2) + (3 + 4)$	$\lambda x.(x + 2) + (3 + 4)$
1-3	$1 + 2$	4-7	$\square + (3 + 4)$	$\lambda x.x + (3 + 4)$
1-7	$(1 + 2) + (3 + 4)$	·	$\square$	$\lambda x.x$

Note that 2, 3, 4, and $3+4$ are not redexes, while 1, $1+2$, and $(1+2)+(3+4)$ are redexes. A redex is an expression that can be evaluated first. Since 2 cannot be evaluated until 1 is evaluated, 2 is not a redex. Similarly, 3, 4, and $3 + 4$ cannot be evaluated until $1 + 2$ is evaluated, so they are not redexes. On the other hand, $1 + 2$ is a redex because there is nothing need to be done before the evaluation of $1 + 2$.

Since a continuation also consists of multiple steps of computation, it can split again into a redex and the continuation of the redex. For example, consider the continuation of 1, which consists of steps 2-7. If we split it into step 2 and steps 3-7, the redex is the expression 2, and the continuation is $(\underline{1} + \square) + (3 + 4)$. Here, the line below 1 expresses that 1 is an integer value, not an expression.

Therefore, evaluation of an expression repeats evaluation of a redex and application of a continuation. A given expression splits into a redex and a continuation. The redex evaluates to a value, and the continuation is applied to the value. Then, the continuation splits again into a redex and a continuation, and the redex is evaluated. This process repeats until there is no more remaining computation, i.e. the continuation becomes the identity function.

14.2 Continuation-Passing Style

Continuation-passing style (CPS) is a style of programming that passes remaining computations to function calls. In CPS, programs never use return values; they pass continuations as arguments instead. Therefore, by writing programs in CPS, continuations become explicit in the source code of the programs.

Before moving on to the detail of CPS, recalling store-passing style, used in Chapter 11 and Chapter 12, would help you understand CPS. Those chapters choose to use store passing in order to show how we can implement mutation without mutation. The following code is part of the implementation of a BFAE interpreter in Chapter 11.

```
def interp(e: Expr, env: Env, sto: Sto): (Value, Sto) =
  e match {
    ...
    case Add(l, r) =>
      val (NumV(n), ls) = interp(l, env, sto)
      val (NumV(m), rs) = interp(r, env, ls)
      (NumV(n + m), rs)
    case NewBox(e) =>
      val (v, s) = interp(e, env, sto)
      val a = sto.keys.maxOption.getOrElse(0) + 1
      (BoxV(a), s + (a -> v))
  }
```

A store-passing interpreter passes the current store to each function call, and each function call returns its resulting store.

If we use mutable maps, we can implement interpreters of BFAE and MFAE without store-passing style. The following code is part of the implementation of a BFAE interpreter using a mutable map.

```
type Sto = scala.collection.mutable.Map[Addr, Value]
val sto: Sto = scala.collection.mutable.Map()

def interp(e: Expr, env: Env): Value =
  e match {
    ...
    case Add(l, r) =>
      val NumV(n) = interp(l, env)
      val NumV(m) = interp(r, env)
      NumV(n + m)
    case NewBox(e) =>
      val v = interp(e, env)
      val a = sto.keys.maxOption.getOrElse(0) + 1
      sto += (a -> v)
      BoxV(a)
  }
```

The variable `sto` denotes a mutable map. The function `interp` depends on `sto`, instead of passing stores as an argument and a return value. The `Add` case does not pass the resulting store from the evaluation of `l` to the

evaluation of `r`. The `NewBox` case simply mutates `sto` to create a new box. Note that `sto += (a -> v)` mutates the map by adding a mapping from `a` to `v`. Store passing is unnecessary since there is a global, mutable map, which records every update.

Two code snippets clearly compare interpreters with and without store passing. In the former, with store passing, the current store at each point of execution is explicit. When we see the `Add` case, it is clear that `sto` is used for the evaluation of `l`, which may change the store, and the resulting store of `l` is used for the evaluation of `r`. However, in the latter, without store passing, the current store at each point is implicit. The code does not reveal the fact that `interp(l, env)` can change the store and, therefore, affect the result of `interp(r, env)`. Implementation with store-passing style explicitly shows the use and flow of stores by passing stores from functions to functions, while implementation without store-passing style hides the use and flow of stores and makes the code shorter.

CPS is similar to store-passing style. The difference is that CPS passes continuations, while store-passing style passes stores. Like that store-passing style exposes the store used by each function application, CPS exposes the continuation of each function application.

This section illustrates how we can write programs in CPS by giving factorial as an example. Consider a function calculating the factorial of a given integer. The following function does not use CPS:

```
def factorial(n: Int): Int =
  if (n <= 1)
    1
  else
    n * factorial(n - 1)
```

Since `factorial` does not use CPS, the continuation is implicit. For example, in `factorial(5) + 1`, the continuation of `factorial(5)` is to add `1` to the result, i.e. `x => x + 1`. Although the continuation of `factorial(5)` does exist and is executed during the evaluation of `factorial(5) + 1`, we cannot find `x => x + 1` in the code per se. The reason is that `factorial` does not use CPS.

Let us transform this function to use CPS. Since each function in CPS takes a continuation as an argument, the first thing to do is to add a parameter to a function. The continuation of a function application uses the return value of certain computation. Therefore, a continuation can be interpreted as a function that takes the return value as input. In the case of `factorial`, the continuation takes an integer as input. On the other hand, there is no restriction on what the continuation computes; it can do whatever it wants. In `factorial(5) + 1`, the continuation of `factorial(5)` results in an integer. At the same time, `factorial(5) + 1` results in an integer, too. In `120 == factorial(5)`, the continuation of `factorial(5)`, which is `x => 120 == x`, results in a boolean. The whole expression `120 == factorial(5)` also results in a boolean. Therefore, the output of a continuation can have any type, but the type must be the same as the type of the whole expression.

Based on these observations, we can define the type of the continuation of `factorial`. It is a function type whose parameter type is `Int`. The

return type can be any type, but for brevity, we fix the return type to `Int`.

```
type Cont = Int => Int
```

Now, we can add a parameter that denotes the continuation of a function call to `factorial`. We call this new function `factorialCps`.

```
def factorialCps(n: Int, k: Cont): Int = ...
```

`k` is the continuation of the function. Thus, `factorialCps(n, k)` means evaluating `factorial(n)` where its continuation is `k`. According to the definition of a continuation, `k(factorial(n))` must equal `factorialCps(n, k)`. The return type of `factorialCps` must be the same as the return type of `k`. Since the return type of `k` is fixed to `Int`, the return type of `factorialCps` also is `Int`.

The most naïve implementation of `factorialCps` is as follows:

```
def factorialCps(n: Int, k: Cont): Int =
  k(factorial(n))
```

Obviously, it is not the correct implementation of `factorialCps` because it still depends on `factorial`; we want `factorialCps` to be independent of `factorial`. The current version is just a specification, not an implementation, of `factorialCps`. However, it is a good starting point. We can replace `factorial(n)` with the body of `factorial` to obtain the following code:

```
def factorialCps(n: Int, k: Cont): Int =
  k(
    if (n <= 1)
      1
    else
      n * factorial(n - 1)
  )
```

Note that `f(if (e1) e2 else e3)` is the same as `if (e1) f(e2) else f(e3)`. Therefore, the above code is equivalent to the following code:

```
def factorialCps(n: Int, k: Cont): Int =
  if (n <= 1)
    k(1)
  else
    k(n * factorial(n - 1))
```

It looks better than before but still depends on `factorial`, which is not a function written in CPS. The use of `factorial` must disappear. Now, the goal is eliminating `factorial` in `k(n * factorial(n - 1))`. It is possible by using `factorialCps` instead of `factorial`. The key intuition to achieve the goal is to recall that `k(factorial(n))` equals

`factorialCps(n, k)`. Based on the equality, we can conclude that the following equation is true:

```
    k(n * factorial(n - 1))
 =  (x => k(n * x))(factorial(n - 1))
 =  factorialCps(n - 1, x => k(n * x))
```

`(x => k(n * x))(factorial(n - 1))` applies `x => k(n * x)` to the result of `factorial(n - 1)` and, thus, equals `k(n * factorial(n - 1))`. Since `k(factorial(n))` equals `factorialCps(n, k)`, `(x => k(n * x))(factorial(n - 1))` equals `factorialCps(n - 1, x => k(n * x))`. By utilizing the equality, we attain the following code:

```
def factorialCps(n: Int, k: Cont): Int =
  if (n <= 1)
    k(1)
  else
    factorialCps(n - 1, x => k(n * x))
```

The function uses CPS because its recursive call explicitly passes the continuation as an argument. When `n` is greater than one, `factorialCps(n, k)` computes $(n-1)!$, multiplies the result by `n`, and applies `k` to the result of the multiplication. The first step, computing $(n-1)!$, is done by calling `factorialCps` itself. The subsequent two steps are the continuation of the recursive call. In the implementation, the continuation is `x => k(n * x)`. It exactly coincides with the aforementioned steps: multiplying the result by `n` and applying `k`.

Now, we can compute 5! with `factorialCps` by writing `factorialCps(5, x => x)`. The continuation is `x => x` because there is nothing more to do with 5!, which is the desired result. In `factorial(5)`, the continuation is implicit since `x => x` is not written in the code. On the other hand, `x => x` is explicitly written in `factorialCps(5, x => x)`, which clearly illustrates the main characteristic of CPS. Similarly, to compute $5! + 1$, we can write `factorialCps(5, x => x + 1)` instead of `factorial(5) + 1`. To obtain $5! + 1$ from 5!, the only thing to do is adding 1. Therefore, the continuation is `x => x + 1`. Just like before, the code with `factorialCps` directly shows the continuation, while the code with `factorial` does not.

Since the output type of a continuation is `T`, any code using `factorial` can be rewritten with `factorialCps`. For example, `factorial(5) % 2 == 0` checks whether 5! is an even integer. It is equivalent to `factorialCps(5, x => x % 2 == 0)`, which explicitly shows the continuation. Similarly, `println(factorial(5))` prints `120`, which is 5!. It is the same as `factorialCps(5, println)`, which also reveals the continuation.

The code written in CPS has the following characteristics:

- Each function takes a continuation as an argument, and each function application passes a continuation as an argument.
- A continuation is used—called or passed to another function—once and at most once in a function body.
- The return value of every function application is never used.
- Every function call is a tail call.
- Every function ends with a function call.

These are not individual ones; they are connected and express the same idea. Since a continuation is given as an argument, the only way to finish a computation is calling the continuation. Therefore, a continuation is used once and at most once in a function body. Also, there is no need to do additional computation with the return value of a function application. The continuation does every additional computation with the return value, so return values are not used at all. Since we do not use return values, every call is a tail call. Once a function calls another function, the result of the callee is the result of caller. Moreover, there is no way of returning from a function without calling any function. If the function is the last step of a computation, it must call its continuation. Otherwise, it needs to call another function to proceed the computation. Therefore, every function ends with a function call.

While CPS may seem to be needlessly complex, it is useful in various cases. If we compare `factorial` and `factorialCps`, the former looks more concise. It is difficult to implement programs correctly in CPS. One benefit of CPS is that it makes every function call be a tail call. If implementation languages support tail-call optimization, CPS can be used to avoid stack overflow. However, this book uses Scala, which optimizes only tail-recursive calls. Scala programs written in CPS can suffer from stack overflow despite the use of CPS. Then, why does this section introduce CPS? The first reason is to help readers understand the notion of a continuation. The other reason is that the characteristic of CPS, passing a continuation explicitly as a value, is sometimes useful. The next chapter shows such an example: an interpreter of a language with first-class continuations. We will see how CPS can contribute to the implementation of an interpreter in the next chapter.

14.3 Interpreter in CPS

Let us implement an interpreter of FAE in CPS. As explained already, there is no reason to implement an interperter of FAE in CPS. However, CPS is an appropriate implementation strategy for the interpreter of the next chapter, and this section is preparation for the next chapter.

First, recall the previous implementation:

```
def interp(e: Expr, env: Env): Value = e match {
  case Num(n) => NumV(n)
  case Add(l, r) =>
    val v1 = interp(l, env)
    val v2 = interp(r, env)
    val NumV(n) = v1
    val NumV(m) = v2
    NumV(n + m)
  case Sub(l, r) =>
    val v1 = interp(l, env)
    val v2 = interp(r, env)
    val NumV(n) = v1
    val NumV(m) = v2
    NumV(n - m)
  case Id(x) => env(x)
```

```
  case Fun(x, b) => CloV(x, b, env)
  case App(f, a) =>
    val fv = interp(f, env)
    val av = interp(a, env)
    val CloV(x, b, fEnv) = fv
    interp(b, fEnv + (x -> av))
}
```

To re-implement the interpreter in CPS, we should add the type of a continuation.

```
type Cont = Value => Value
```

A continuation takes a value of `Value` as input because `interp` returns a value of `Value`. The return type of a continuation can be any type, but like before, we choose `Value` just for brevity.

The following function, `interpCps`, is the CPS version of `interp`:

```
def interpCps(e: Expr, env: Env, k: Cont): Value = e match {
  ...
}
```

For any `e`, `env`, and `k`, `interpCps(e, env, k)` must equal `k(interp(e, env))`.

Now, we need to implement each case of the pattern matching. First, consider the `Num` case. `interp(Num(n), env)` equals `NumV(n)`. Hence, `k(interp(Num(n), env))` equals `k(NumV(n))`. Since `interpCps(Num(n), env, k)` must also equal `k(NumV(n))`, the `Num` case can be implemented as follows:

```
case Num(n) => k(NumV(n))
```

It is similar to `k(1)` of `factorialCps` when `n` is less than or equal to one. Since there is no need of a recursive call, the function simply passes `NumV(n)` to the continuation.

The `Id` and `Fun` cases are similar to the `Num` case.

```
case Id(x) => k(env(x))
case Fun(x, b) => k(CloV(x, b, env))
```

The remaining cases are `Add`, `Sub`, and `App`. They are similar in the sense that each sort of expression consists of two subexpressions, so if you understand one of them, the others are straightforward. Let us consider the `Add` case first. The previous implementation is as follows:

```
val v1 = interp(l, env)
val v2 = interp(r, env)
add(v1, v2)
```

where `add(v1, v2)` denotes `val NumV(n) = v1; val NumV(m) = v2; NumV(n + m)`. Since `interpCps(e, env, k)` equals `k(interp(e, env))`, we can start from the following code:

```
val v1 = interp(l, env)
val v2 = interp(r, env)
k(add(v1, v2))
```

By desugaring variable definitions into an anonymous function and a function application, we can find the continuation of `interp(l, env)`. Recall that `val x = e1; e2` is equivalent to `(x => e2)(e1)` as shown in Chapter 9. Desugaring yields the following code:

```
(v1 => {
  val v2 = interp(r, env)
  k(add(v1, v2))
})(interp(l, env))
```

The function applied to `interp(l, env)` is the continuation of `interp(l, env)`. Then, `interp` can be replaced with `interpCps` because `k(interp(e, env))` is the same as `interpCps(e, env, k)`.

```
interpCps(l, env, v1 => {
  val v2 = interp(r, env)
  k(add(v1, v2))
})
```

Now, let us focus on the body of the continuation.

```
val v2 = interp(r, env)
k(add(v1, v2))
```

In a similary way, desugaring reveals the continuation of `interp(r, env)`.

```
(v2 =>
  k(add(v1, v2))
)(interp(r, env))
```

Just like before, `interp` can be replaced with `interpCps`.

```
interpCps(r, env, v2 =>
  k(add(v1, v2))
)
```

Then, we can use this new expression as the body of the continuation.

```
interpCps(l, env, v1 =>
  interpCps(r, env, v2 =>
    k(add(v1, v2))
  )
)
```

Finally, we obtain the complete implementation of the `Add` case by replacing `add` with its definition.

```
case Add(l, r) =>
  interpCps(l, env, v1 =>
    interpCps(r, env, v2 => {
      val NumV(n) = v1
      val NumV(m) = v2
      k(NumV(n + m))
    })
  )
```

The code explicitly shows the continuation of each function application. The first function application evaluates `l` under `env`. Its continuation is `v1 => interpCps(r, env, v2 => k(add(v1, v2)))`. Therefore, we can say that `r` is evaluated after the evaluation of `l`. In `k(add(v1, v2))`, `v1` denotes the result of `l`. The second function application evaluates `r`. Its continuation is `v2 => k(add(v1, v2))`. In `k(add(v1, v2))`, `v2` denotes the result of `r`. `k(add(v1, v2))` makes `NumV(n + m)` from `v1` and `v2` and passes `NumV(n + m)` to `k`, the continuation of evaluating `Add(l, r)`.

The `Sub` case is similar to the `Add` case.

```
case Sub(l, r) =>
  interpCps(l, env, v1 =>
    interpCps(r, env, v2 => {
      val NumV(n) = v1
      val NumV(m) = v2
      k(NumV(n - m))
    })
  )
```

The `App` case is also similar but needs extra care. The previous implementation is as follows:

```
val fv = interp(f, env)
val av = interp(a, env)
val CloV(x, b, fEnv) = fv
k(interp(b, fEnv + (x -> av)))
```

By applying the same strategy, we attain the following code:

```
interpCps(f, env, fv =>
  interpCps(a, env, av => {
    val CloV(x, b, fEnv) = fv
    k(interp(b, fEnv + (x -> av)))
  })
)
```

Unlike `Add` and `Sub`, an `interp` function call still exists. It is not CPS because the result of the function call is used by being passed to `k`. Replacing `interp` with `interpCps` resolves the problem.

```
case App(f, a) =>
  interpCps(f, env, fv =>
```

```
    interpCps(a, env, av => {
      val CloV(x, b, fEnv) = fv
      interpCps(b, fEnv + (x -> av), k)
    })
  )
```

The code does not directly call `k`. Instead of calling `k`, it passes `k` as an argument. Look at `interpCps(b, fEnv + (x -> av), k)`. `k` is passed to `interpCps` and, therefore, eventually called.

14.4 Small-Step Operational Semantics

This section defines semantics that explicitly shows continuations for FAE. Previous chapters define semantics in a big-step style. Big-step semantics is intuitive, and its inference rules give nice clues to interpreter implementers. A single rule usually corresponds to a single case of pattern matching in an interpreter, so it is straightforward to implement an interpreter based on the big-step semantics and write semantics rules based on the implementation of an interpreter.

However, it is difficult to formalize continuations in big-step semantics. The problem is that an inference rule of big-step semantics describes the result of an expression instead of one step of computation. For instance, consider the following rule:

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

The rule implies that the value of $e_1 + e_2$ is $n_1 + n_2$. It does not explain how e_1 evaluates to n_1 and e_2 evaluates to n_2. Therefore, the rule does not describe steps that evaluate e_1 to n_1 and e_2 to n_2. The only step the rule describes is the last step: adding n_1 and n_2.

Another problem is that rules of big-step semantics do not specify the order of computation. Consider the above rule again. The rule does not decide the order between e_1 and e_2. The only two things the rule requires are that e_1 evaluates to n_1 and that e_2 evaluates to n_2. The evaluation of e_1 may precede that of e_2, and vice versa.

These two characteristics of big-step semantics hamper us from formalizing continuations. Continuations highly rely on precise steps of computation whose order is fixed. Big-step semantics neither shows all the steps nor specifies the order. Therefore, we do not use big-step semantics in this section.

Small-step operational semantics is another way of defining the semantics of a language. While big-step semantics defines a relation over expressions and values, small-step semantics defines a relation between states and states. If one step of computation transfers a program from state A to state B, A and B are related by the relation. Such one step of computation is called *reduction*. If reduction from A to B is possible, we say that A is reducible and reduced to B. On the other hand, if a state cannot be reduced to any state, the state is irreducible. In small-step semantics, the definition of a state varies. One possible definition of a state is an

expression. For example, 1 + 2 and 3 are states, and 1 + 2 is reduced to 3 by one step of computation. However, this section's small-step semantics does not use an expression as a state. We will introduce the definition of a state soon. The execution of a program is defined as repeated reduction. The execution starts from an initial state, and the state becomes reduced if possible. When no further reduction is possible, the execution stops, and the final state is the result.

Small-step semantics fits formalizing continuations. Since execution is a sequence of multiple reduction steps, every step of computation can be identified. The order between the steps naturally arises. By splitting the steps into two parts, we can formally describe a redex and a continuation.

Now, let us define states of FAE. A state of FAE is a pair of a computation stack and a value stack. The following defines computation and value stacks:

$$\begin{array}{lll} k & ::= & \square \mid \sigma \vdash e :: k \mid (+) :: k \mid (-) :: k \mid (@) :: k \\ s & ::= & \blacksquare \mid v :: s \end{array}$$

where the metavariable k ranges over computation stacks and the metavariable s ranges over value stacks. Let S_{Comp} be the set of every computation stack and S_{Val} be the set of every value stack. We write $k \mid\mid s$ to denote a state that consists of a computation stack k and a value stack s. k includes remaining steps of computation, and s includes values used by those steps.

A computation stack is a stack containing remaining steps of computation. The white square, $\square$, denotes the empty computation stack, which implies that there is nothing more to do.[3] If the computation stack of a state is $\square$, no further reduction is possible, and the evaluation ends. There are four kinds of computation: $\sigma \vdash e$, $(+)$, $(-)$, and $(@)$. Their detailed descriptions and formal definitions will be provided soon. The computation at the top of the stack is the first step of computation. After finishing the step, the corresponding computation in the stack is popped. For example, $\emptyset \vdash 1 :: \square$ has one step of compuation: $\emptyset \vdash 1$. After finishing the step, the stack becomes $\square$, and the evaluation finishes.

3: Note that $\square$ has no relation to $\square$ used to represent continuations intuitively. In the continuation $\square + 1$, $\square$ means a hole, which should be filled by the result of a redex. On the other hand, in the small-step semantics, $\square$ is the empty computation stack. The overlapped use of $\square$ is just coincidence.

A value stack is a stack containing values. The black square, $\blacksquare$, denotes the empty value stack. Therefore, $1 :: 2 :: \blacksquare$ is a stack that contains 1 and 2. 1 is at the top of the stack. Since a value stack is a stack, the only possible operations are push and pop. If we push 0 onto $1 :: 2 :: \blacksquare$, we obtain $0 :: 1 :: 2 :: \blacksquare$. If we pop the top element from $1 :: 2 :: \blacksquare$, we obtain $2 :: \blacksquare$.

Before moving on to the formal defintion of reduction, let us see the high-level idea of reduction first. Each reduction step pops the element at the top of the computation stack and manipluates the computation and value stacks depending on the popped computation. $\sigma \vdash e$ is the only kind that pushes a value onto the value stack. It evaluates e under σ and pushes the result of e onto the value stack. Therefore, if the current state is $\sigma \vdash e :: k \mid\mid s$, then the redex is e, and the continuation is $k \mid\mid s$. Applying the continuation to a value is done by pushing the value onto the value stack. Therefore, if e evaluates to v, the continuation is applied to v, so the state becomes $k \mid\mid v :: s$. For example, $\emptyset \vdash 1 :: k \mid\mid s$ is reduced

to $k \mid\mid 1 :: s$. On the other hand, the other kinds of computation consume values in the value stack. (+) pops two values from the value stack and then pushes the sum of the values onto the value stack. For instance, $(+) :: k \mid\mid 2 :: 1 :: s$ is reduced to $k \mid\mid 3 :: s$. (−) and (@) are similar to (+). (−) performs subtraction and (@) performs function application.

It becomes clear that each kind of a computation step coincides with our intuitive notion of a computation step when we see an example. Consider the evaluation of 1 + 2 under the empty environment. The evaluation consists of three steps:

1. Evaluate the expression 1 to get the integer value 1.
2. Evaluate the expression 2 to get the integer value 2.
3. Add the integer value 1 to the integer value 2 to get the integer value 3.

Steps 1 and 2 can be represented by $\emptyset \vdash 1$ and $\emptyset \vdash 2$, repectively. $\emptyset \vdash 1$ evalutes 1 to get 1 and pushes 1 onto the value stack for the continuation. Similarly, $\emptyset \vdash 2$ evaluates 2 to get 2 and pushes 2 onto the value stack for the continuation. Finally, step 3, which can be represented by (+), pops both values and computes the sum. The sum, 3, is pushed onto the value stack and becomes the result of the execution. Therefore, the evalution of 1 + 2 can be decomposed into three steps in the computation stack: $\emptyset \vdash 1 :: \emptyset \vdash 2 :: (+) :: \square$.

Now, we formally define reduction. Reduction is a relation over S_{Comp}, S_{Val}, S_{Comp}, and S_{Val}.

$$\rightarrow \subseteq S_{\text{Comp}} \times S_{\text{Val}} \times S_{\text{Comp}} \times S_{\text{Val}}$$

We write $k_1 \mid\mid s_1 \rightarrow k_2 \mid\mid s_2$ if $k_1 \mid\mid s_1$ is reduced to $k_2 \mid\mid s_2$. For example, we write $\emptyset \vdash 1 :: \square \mid\mid \blacksquare \rightarrow \square \mid\mid 1 :: \blacksquare$

As mentioned already, execution of a program is to repeat reduction until no more reduction is possible. When the state cannot be reduced any longer, the execution terminates, and the state represents the result. To formalize the notion of execution, we define the repeated reduction as the reflexive, transitive closure of the reduction relation.

Reflexive relations

Let A be a set and R be a binary relation over A and A.

If $(a, a) \in R$ for every $a \in A$, R is reflexive.

Transitive relations

Let A be a set and R be a binary relation over A and A.

If $(a, b), (b, c) \in R$ implies $(a, c) \in R$ for every $a, b, c \in A$, R is transitive.

Reflexive, transitive closures

Let A be a set and R be a binary relation over A and A.

The reflexive, transitive closure of R is the smallest set S such that $R \subseteq S \subseteq A \times A$ and S is reflexive and transitive.

$\rightarrow^*$ denotes repeated reduction. $k_1 \mid\mid s_1 \rightarrow^* k_n \mid\mid s_n$ implies $k_1 \mid\mid s_1 \rightarrow k_2 \mid\mid s_2$, $k_2 \mid\mid s_2 \rightarrow k_3 \mid\mid s_3$, ..., and $k_{n-1} \mid\mid s_{n-1} \rightarrow k_n \mid\mid s_n$. The following rules formalize the relation:

$$k \mid\mid s \rightarrow^* k \mid\mid s$$

$$\frac{k_1 \mid\mid s_1 \rightarrow^* k_2 \mid\mid s_2 \qquad k_2 \mid\mid s_2 \rightarrow k_3 \mid\mid s_3}{k_1 \mid\mid s_1 \rightarrow^* k_3 \mid\mid s_3}$$

By definition, $\rightarrow^*$ is indeed the reflexive, transitive closure of $\rightarrow$. Intuitively, $k_1 \mid\mid s_1 \rightarrow^* k_2 \mid\mid s_2$ implies that $k_2 \mid\mid s_2$ can be reached from $k_1 \mid\mid s_1$ by zero or more steps of reduction.

Note that $\rightarrow^*$ does not require the resulting state to be irreducible. It just denotes zero or more steps of reduction. When $k_1 \mid\mid s_1 \rightarrow k_2 \mid\mid s_2$ and $k_2 \mid\mid s_2 \rightarrow k_3 \mid\mid s_3$, all of $k_1 \mid\mid s_1 \rightarrow^* k_1 \mid\mid s_1$, $k_1 \mid\mid s_1 \rightarrow^* k_2 \mid\mid s_2$, and $k_1 \mid\mid s_1 \rightarrow^* k_3 \mid\mid s_3$ are true. Therefore, we cannot say a program that starts with $k_1 \mid\mid s_1$ terminates with $k_2 \mid\mid s_2$ even if we know $k_1 \mid\mid s_1 \rightarrow^* k_2 \mid\mid s_2$. $k_2 \mid\mid s_2$ might not be the final state. To say that the program terminates with $k_2 \mid\mid s_2$, we need an additional condition: $k_2 \mid\mid s_2$ is irreducible.

In small-step semantics, there are two kinds of termination. One kind is normal termination, which produces a value as a result. The execution terminates normally when the computation stack is empty. Since each reduction step pops a computation step, reduction is impossible when the stack is empty. The empty stack implies that there is no remaining computation, so this kind of termination is intended. The other kind is abnormal termination, i.e. termination due to a run-time error. It happens when the value stack contains values that cannot be used by the current computation step. For example, if a popped value is not an integer, both addition or subtraction are impossible, so reduction cannot happen when the popped computation is $(+)$ or $(-)$. It prevents further reduction even when there is remaining computation. Therefore, such termination is considered erroneous and harmful.

When evaluation according to the small-step semantics succesfully produces a value, we can reach the same conclusion using the big-step semantics. Recall that $\sigma \vdash e \Rightarrow v$ implies that e evaluates to v under σ. In small-step semantics, evaluation of e under σ starts with $\sigma \vdash e :: \square \mid\mid \blacksquare$. The redex and the continuation of the state are e and the identity function, repectively, so the state does evaluate e. If the evaluation results in v, the final state is $\square \mid\mid v :: \blacksquare$. The computation stack is empty, and the value stack contains only v, which is the result. Thus, the following proposition is true.

$$\forall\sigma.\forall e.\forall v.(\sigma \vdash e \Rightarrow v) \leftrightarrow (\sigma \vdash e :: \square \mid\mid \blacksquare \rightarrow^* \square \mid\mid v :: \blacksquare)$$

More generally, the following statement is true:

$$\forall\sigma.\forall e.\forall v.\forall k.\forall s.(\sigma \vdash e \Rightarrow v) \leftrightarrow (\sigma \vdash e :: k \mid\mid s \rightarrow^* k \mid\mid v :: s)$$

Now, we define the rules for reduction based on the interpreter imple-

mentation of the previous section. Consider the `Num` case first.

```
case Num(n) => k(NumV(n))
```

When n is the redex and `k` is the continuation, the evaluation proceeds by applying `k` to n. This state is represented by $\sigma \vdash n :: k \mid\mid s$, where the continuation `k` is represented by $k \mid\mid s$. Applying $k \mid\mid s$ to n is to evaluate $k \mid\mid n :: s$. Therefore, we define the following reduction rule:

$$\sigma \vdash n :: k \mid\mid s \longrightarrow k \mid\mid n :: s \quad \text{[Red-Num]}$$

The rule also matches our high-level intuition on reduction. $\sigma \vdash n$ evaluates n and gets n. After the reduction, the computation step is removed from the computation stack, and the result is pushed onto the value stack.

In a similar manner, we can define the rules for the `Id` and `Fun` cases.

```
case Id(x) => k(env(x))
case Fun(x, b) => k(CloV(x, b, env))
```

$$\sigma \vdash x :: k \mid\mid s \longrightarrow k \mid\mid \sigma(x) :: s \quad \text{[Red-Id]}$$

$$\sigma \vdash \lambda x.e :: k \mid\mid s \longrightarrow k \mid\mid \langle \lambda x.e, \sigma \rangle :: s \quad \text{[Red-Fun]}$$

Now, let us consider the `Add` case.

```
case Add(l, r) =>
  interpCps(l, env, v1 =>
    interpCps(r, env, v2 =>
      k(add(v1, v2))
    )
  )
```

When $e_1 + e_2$ is the redex, the evaluation splits into three parts:

1. Evaluate e_1 to get v_1. (`interpCps(l, env, v1 =>`)
2. Evaluate e_2 to get v_2. (`interpCps(r, env, v2 =>`)
3. If both v_1 and v_2 are integers, add v_1 and v_2 to get $v_1 + v_2$, and apply the continuation to $v_1 + v_2$. (`k(add(v1, v2))`)

Steps 1 and 2 can be represented by $\sigma \vdash e_1$ and $\sigma \vdash e_2$, respectively. First, $\sigma \vdash e_1$ pushes the result of e_1 onto the value stack, and then $\sigma \vdash e_2$ pushes the result of e_2 onto the value stack. Step 3 can be represented by $(+)$ since it pops two values from the value stack and adds them. From this observation, we define the following rules:

$$\sigma \vdash e_1 + e_2 :: k \mid\mid s \longrightarrow \sigma \vdash e_1 :: \sigma \vdash e_2 :: (+) :: k \mid\mid s \quad \text{[Red-Add1]}$$

$$(+) :: k \mid\mid n_2 :: n_1 :: s \longrightarrow k \mid\mid n_1 + n_2 :: s \quad \text{[Red-Add2]}$$

Rule Red-Add1 splits the evaluation of $e_1 + e_2$ into aforementioned three parts. Then, the subsequent reduction steps evaluate e_1 and e_2 and push their results onto the value stack. After the evaluation of e_2, $(+)$ is at the top of the computation stack. One more reduction step, which is defined by Rule Red-Add2, pops the values from the value stack and pushes their sum to the value stack. The following figure summarizes this process:

$$\begin{array}{lrcl}
 & \sigma \vdash e_1 + e_2 :: k & || & s \\
\rightarrow & \sigma \vdash e_1 :: \sigma \vdash e_2 :: (+) :: k & || & s \\
\rightarrow^* & \sigma \vdash e_2 :: (+) :: k & || & n_1 :: s \\
\rightarrow^* & (+) :: k & || & n_2 :: n_1 :: s \\
\rightarrow & k & || & n_1 + n_2 :: s
\end{array}$$

We can define the rules for the `Sub` case in a similar way:

$$\sigma \vdash e_1 - e_2 :: k \;||\; s \rightarrow \sigma \vdash e_1 :: \sigma \vdash e_2 :: (-) :: k \;||\; s \quad \text{[Red-Sub1]}$$

$$(+) :: k \;||\; n_2 :: n_1 :: s \rightarrow k \;||\; n_1 - n_2 :: s \quad \text{[Red-Sub2]}$$

The only remaining case is `App`.

```
case App(f, a) =>
  interpCps(f, env, fv =>
    interpCps(a, env, av => {
      val CloV(x, b, fEnv) = fv
      interpCps(b, fEnv + (x -> av), k)
    })
  )
```

When $e_1\ e_2$ is the redex, the evaluation splits into three parts:

1. Evaluate e_1 to get v_1. (`interpCps(f, env, fv =>`)
2. Evaluate e_2 to get v_2. (`interpCps(a, env, av =>`)
3. If v_1 is $\langle \lambda x.e, \sigma' \rangle$, evaluate e under $\sigma'[x \mapsto v_2]$ with the given continuation. (`interpCps(b, fEnv + (x -> av), k)`)

The first two steps are the same as those of `Add` and `Sub`. Therefore, we define the following rule:

$$\sigma \vdash e_1\ e_2 :: k \;||\; s \rightarrow \sigma \vdash e_1 :: \sigma \vdash e_2 :: (@) :: k \;||\; s \quad \text{[Red-App1]}$$

However, the last step is a bit different. In `Add` and `Sub`, the last step applies the continuation to a certain value, which is obtained by addition or subtraction. In `App`, the body of the function must be evaluated. Thus, we define the rule to evaluate the body with the same continuation instead of directly applying the continuation to a particular value.

$$(@) :: k \;||\; v :: \langle \lambda x.e, \sigma \rangle :: s \rightarrow \sigma[x \mapsto v] \vdash e :: k \;||\; s \quad \text{[Red-App2]}$$

The following figure summarizes the evaluation of `App`:

$$
\begin{array}{rrcr}
 & \sigma \vdash e_1\ e_2 :: k & \| & s \\
\rightarrow & \sigma \vdash e_1 :: \sigma \vdash e_2 :: (@) :: k & \| & s \\
\rightarrow^* & \sigma \vdash e_2 :: (@) :: k & \| & \langle \lambda x.e, \sigma' \rangle :: s \\
\rightarrow^* & (@) :: k & \| & v_2 :: \langle \lambda x.e, \sigma' \rangle :: s \\
\rightarrow & \sigma'[x \mapsto v_2] \vdash e :: k & \| & s \\
\rightarrow^* & k & \| & v :: s
\end{array}
$$

The following shows the reduction steps of (1+2)−(3+4) as an example:

$$
\begin{array}{rrcr}
 & \emptyset \vdash (1+2) - (3+4) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 1+2 :: \emptyset \vdash 3+4 :: (-) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 1 :: \emptyset \vdash 2 :: (+) :: \emptyset \vdash 3+4 :: (-) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 2 :: (+) :: \emptyset \vdash 3+4 :: (-) :: \square & \| & 1 :: \blacksquare \\
\rightarrow & (+) :: \emptyset \vdash 3+4 :: (-) :: \square & \| & 2 :: 1 :: \blacksquare \\
\rightarrow & \emptyset \vdash 3+4 :: (-) :: \square & \| & 3 :: \blacksquare \\
\rightarrow & \emptyset \vdash 3 :: \emptyset \vdash 4 :: (+) :: (-) :: \square & \| & 3 :: \blacksquare \\
\rightarrow & \emptyset \vdash 4 :: (+) :: (-) :: \square & \| & 3 :: 3 :: \blacksquare \\
\rightarrow & (+) :: (-) :: \square & \| & 4 :: 3 :: 3 :: \blacksquare \\
\rightarrow & (-) :: \square & \| & 7 :: 3 :: \blacksquare \\
\rightarrow & \square & \| & -4 :: \blacksquare
\end{array}
$$

The following shows the reduction steps of $(\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x} + \mathsf{y})\ 1\ 2$ as an example:

$$
\begin{array}{rrcr}
 & \emptyset \vdash e\ 1\ 2 :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash e\ 1 :: \emptyset \vdash 2 :: (@) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash e :: \emptyset \vdash 1 :: (@) :: \emptyset \vdash 2 :: (@) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 1 :: (@) :: \emptyset \vdash 2 :: (@) :: \square & \| & \langle e, \emptyset \rangle :: \blacksquare \\
\rightarrow & (@) :: \emptyset \vdash 2 :: (@) :: \square & \| & 1 :: \langle e, \emptyset \rangle :: \blacksquare \\
\rightarrow & \sigma_1 \vdash \lambda \mathsf{y}.\mathsf{x} + \mathsf{y} :: \emptyset \vdash 2 :: (@) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 2 :: (@) :: \square & \| & \langle \lambda \mathsf{y}.\mathsf{x} + \mathsf{y}, \sigma_1 \rangle :: \blacksquare \\
\rightarrow & (@) :: \square & \| & 2 :: \langle \lambda \mathsf{y}.\mathsf{x} + \mathsf{y}, \sigma_1 \rangle :: \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{x} + \mathsf{y} :: \square & \| & \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{x} :: \sigma_2 \vdash \mathsf{y} :: (+) :: \square & \| & \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{y} :: (+) :: \square & \| & 1 :: \blacksquare \\
\rightarrow & (+) :: \square & \| & 2 :: 1 :: \blacksquare \\
\rightarrow & \square & \| & 3 :: \blacksquare
\end{array}
$$

where

$$
\begin{aligned}
e &= \lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x} + \mathsf{y} \\
\sigma_1 &= [\mathsf{x} \mapsto 1] \\
\sigma_2 &= [\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2]
\end{aligned}
$$

First-Class Continuations 15

The previous chapter defines the small-step semantics of FAE and implements the interpreter of FAE in CPS. Conceptually, continuations exist during the evaluation of FAE programs. However, they are not exposed to programmers. Programmers cannot utilize continuations directly while writing programs in FAE.

A first-class entity of a language is an entity treated as a value. Since it is a value, it can be the value denoted by a variable, an argument for a function call, and the return value of a function. For example, first-class functions are functions used as values.

First-class continuations are continuations used as values. If a language supports first-class continuations, continuations can be the value of a variable, an argument for a function call, and the return value of a function. A continuation can be considered as a function since it takes a value and performs computation. Programmers can call continuations like calling functions. However, continuations are different from functions. A function call returns a value, and the execution continues with the return value. On the other hand, a continuation call does not come back to its call site. The continuation at some point of execution is the remaining computation. Once a continuation is called and evaluated, the execution finishes. Calling a continuation changes the current continuation to the called one. It changes the control flow of the execution. First-class continuations allow programmers to express computations with complex control flows concisely.

This chapter defines KFAE by extending FAE with first-class continuations. It defines the small-step semantics of KFAE and implements an interpreter of KFAE in CPS. While implementing the interpreter, you will see why CPS is required. In addition, this chapter shows utilization of first-class continuations in programming.

15.1 Syntax

The syntax of KFAE is as follows: [1]

$$e ::= \cdots \mid \mathsf{vcc}\ x\ \mathsf{in}\ e$$

An expression vcc x in e evaluates e while x denotes the continuation of vcc x in e. The term "cc" of vcc stands for the **c**urrent **c**ontinuation. The scope of x equals e. When a continuation is called, the continuation replaces the continuation of that point.

1: We omit the common part to FAE.

15.2 Semantics

Before going deep into the semantics of first-class continuations, it would be better to understand the high-level idea with some examples. Consider 1 + (vcc x in (x 2) + 3). The continuation of (vcc x in (x 2) + 3) is to add the result to 1. Intuitively, we can use $\underline{1}$ + □ to represent the continuation.[2] The continuation is the value of x. After binding x, x 2 is evaluated. The continuation of x 2 is $\underline{1}$ + (□ + 3). Therefore, if x is a normal function, the result of function application fills the hole, and the evaluation continues. However, x is a continuation, not a function. The evaluation of x 2 completely ignores the original continuation $\underline{1}$ + (□ + 3). It replaces the continuation with the continuation denoted by x and fills the hole with the argument, 2. Thus, x 2 results in evaluating 1 + 2. Since the original continuation is ignored, there is nothing more do to after the evaluation of 1 + 2. The result of the whole expression is 3.

2: Like before, the line below 1 denotes that 1 is an integer value, not an expression.

To compare first-class continuations and functions, consider the following expression:

1 + (val x=λy.1 + y in (x 2) + 3)

In the previous expression, x denotes a continuation, but in this expression, x denotes a function. The continuation and the function are almost the same. Both take an argument and add 1 to the argument. However, continuations change the control flow, while functions do not. Therefore, in this case, x 2 preserves its continuation, $\underline{1}$ + (□ + 3). The return value of the function application is 3, and it fills the hole in the original continuation. After the function returns, 1 + (3 + 3) is evaluated, and the whole expression results in 7.

Let us consider another example:

vcc x in (vcc y in x (1 + (vcc z in y z))) 3

What is the result of this expression? The first thing happens during the evaluation is binding of x. x denotes the continuation of the whole expression, which is the identity function, i.e. □. Then, (vcc y in x (1 + (vcc z in y z))) 3 is evaluated. Any function application evaluates the expression at the function position first. Thus, the redex is vcc y in x (1 + (vcc z in y z)), and the continuation is □ 3. The redex defines y, which denotes the continuation, □ 3. Under the environment containing x and y, x (1 + (vcc z in y z)) is evaluated. x directly evaluates to a continuation, and the argument expression becomes the redex. At this point, the continuation is (x □) 3. The argument expression is 1 + (vcc z in y z), and 1 evaluates to 1. Then, vcc z in y z becomes the redex, and the continuation is (x (1 + □)) 3. Therefore, z denotes (x (1 + □)) 3. When y is applied to z, the original continuation is ignored, and z fills the hole in the continuation denoted by y. Now, the remainig computation is z 3, which is obtained by filling the hole of □ 3 with z. Applying z to 3 ignores the continuation again, and (x (1 + 3)) 3 is obtained by filling the hole of (x (1 + □)) 3 with 3. Since 1 + 3 evaluates to 4, x is applied to 4. Then, 4 fills the hole of □ and becomes the final result.

Now, we define the semantics of KFAE. First, since continuations are values, values must be extended.

$$v ::= \cdots \mid \langle k, s \rangle$$

A continuation as a value is a pair of a computation stack and a value stack. $\langle k, s \rangle$ denotes a continuation whose computation stack is k and value stack is s. It corresponds to the state $k \mid\mid s$. The previous chapter shows that applying a continuation $k \mid\mid s$ to a value v changes the state to $k \mid\mid v :: s$. Therefore, applying $\langle k, s \rangle$ to v reduces the state to $k \mid\mid v :: s$.

Since KFAE has a new sort of an expression, vcc x in e, we need to define the reduction rule for vcc x in e.

$$\sigma \vdash \text{vcc } x \text{ in } e :: k \mid\mid s \longrightarrow \sigma[x \mapsto \langle k, s \rangle] \vdash e :: k \mid\mid s \quad \text{[Red-Vcc]}$$

Expression vcc x in e evaluates e where x denotes the continuation of vcc x in e. If the current state is $\sigma \vdash$ vcc x in $e :: k \mid\mid s$ and the redex is vcc x in e, the continuation is $k \mid\mid s$. This continuation can be represented by a value $\langle k, s \rangle$. Therefore, reduction changes the top of the computation stack to $\sigma[x \mapsto \langle k, s \rangle] \vdash e$.

Adding Rule Red-Vcc is not enough to define the semantics of KFAE. Due to the existence of first-class continuations, function application expressions must be able to handle not only closures but also first-class continuations. To define the reduction rule, recall the reduction rule that handles function application.

$$(@) :: k \mid\mid v :: \langle \lambda x.e, \sigma \rangle :: s \longrightarrow \sigma[x \mapsto v] \vdash e :: k \mid\mid s \quad \text{[Red-App2]}$$

If a continuation is applied instead of a function, the state should be $(@) :: k \mid\mid v :: \langle k', s' \rangle :: s$. The current continuation is $k \mid\mid s$. Since a continuation is applied to a value, the original continuation is ignored. The new continuation that replaces the original one is $k' \mid\mid s'$, which comes from $\langle k', s' \rangle$ in the value stack. The argument is v, which is at the top of the value stack. Thus, reduction results in the state $k' \mid\mid v :: s'$.

$$(@) :: k \mid\mid v :: \langle k', s' \rangle :: s \longrightarrow k' \mid\mid v :: s' \quad \text{[Red-App2-Cont]}$$

The following shows that $1 + (\text{vcc x in } ((\text{x } 2) + 3))$ evaluates to 3 by applying reduction according to the semantics:

$$\begin{array}{rrcr}
 & \emptyset \vdash 1 + (\text{vcc x in } ((\text{x } 2) + 3)) :: \square & \mid\mid & \blacksquare \\
\longrightarrow & \emptyset \vdash 1 :: \emptyset \vdash \text{vcc x in } ((\text{x } 2) + 3) :: (+) :: \square & \mid\mid & \blacksquare \\
\longrightarrow & \emptyset \vdash \text{vcc x in } ((\text{x } 2) + 3) :: (+) :: \square & \mid\mid & 1 :: \blacksquare \\
\longrightarrow & \sigma \vdash (\text{x } 2) + 3 :: (+) :: \square & \mid\mid & 1 :: \blacksquare \\
\longrightarrow & \sigma \vdash \text{x } 2 :: \sigma \vdash 3 :: (+) :: (+) :: \square & \mid\mid & 1 :: \blacksquare \\
\longrightarrow & \sigma \vdash \text{x} :: \sigma \vdash 2 :: (@) :: \sigma \vdash 3 :: (+) :: (+) :: \square & \mid\mid & 1 :: \blacksquare \\
\longrightarrow & \sigma \vdash 2 :: (@) :: \sigma \vdash 3 :: (+) :: (+) :: \square & \mid\mid & v :: 1 :: \blacksquare \\
\longrightarrow & (@) :: \sigma \vdash 3 :: (+) :: (+) :: \square & \mid\mid & 2 :: v :: 1 :: \blacksquare \\
\longrightarrow & (+) :: \square & \mid\mid & 2 :: 1 :: \blacksquare \\
\longrightarrow & \square & \mid\mid & 3 :: \blacksquare
\end{array}$$

where σ is $[\mathsf{x} \mapsto \langle (+) :: \square \mid\mid 1 :: \blacksquare \rangle]$ and v is $\langle (+) :: \square \mid\mid 1 :: \blacksquare \rangle$.

The following shows that vcc x in (vcc y in x (1 + (vcc z in y z))) 3 evaluates to 4:

$$
\begin{array}{rrcr}
 & \emptyset \vdash \mathsf{vcc\ x\ in\ (vcc\ y\ in\ x\ (1 + (vcc\ z\ in\ y\ z)))}\ 3 :: \square & \mid\mid & \blacksquare \\
\rightarrow & \sigma_1 \vdash \mathsf{(vcc\ y\ in\ x\ (1 + (vcc\ z\ in\ y\ z)))}\ 3 :: \square & \mid\mid & \blacksquare \\
\rightarrow & \sigma_1 \vdash \mathsf{vcc\ y\ in\ x\ (1 + (vcc\ z\ in\ y\ z))} :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{x\ (1 + (vcc\ z\ in\ y\ z))} :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{x} :: \sigma_2 \vdash \mathsf{1 + (vcc\ z\ in\ y\ z)} :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{1 + (vcc\ z\ in\ y\ z)} :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & v_1 :: \blacksquare \\
\rightarrow & \sigma_2 \vdash 1 :: \sigma_2 \vdash \mathsf{vcc\ z\ in\ y\ z} :: (+) :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & v_1 :: \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{vcc\ z\ in\ y\ z} :: (+) :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & 1 :: v_1 :: \blacksquare \\
\rightarrow & \sigma_3 \vdash \mathsf{y\ z} :: (+) :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & 1 :: v_1 :: \blacksquare \\
\rightarrow & \sigma_3 \vdash \mathsf{y} :: \sigma_3 \vdash \mathsf{z} :: (@) :: (+) :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & 1 :: v_1 :: \blacksquare \\
\rightarrow & \sigma_3 \vdash \mathsf{z} :: (@) :: (+) :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & v_2 :: 1 :: v_1 :: \blacksquare \\
\rightarrow & (@) :: (+) :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & v_3 :: v_2 :: 1 :: v_1 :: \blacksquare \\
\rightarrow & \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & v_3 :: \blacksquare \\
\rightarrow & (@) :: \square & \mid\mid & 3 :: v_3 :: \blacksquare \\
\rightarrow & (+) :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & 3 :: 1 :: v_1 :: \blacksquare \\
\rightarrow & (@) :: \sigma_1 \vdash 3 :: (@) :: \square & \mid\mid & 4 :: v_1 :: \blacksquare \\
\rightarrow & \square & \mid\mid & 4 :: \blacksquare
\end{array}
$$

where

$v_1 = \langle \square, \blacksquare \rangle$
$v_2 = \langle \sigma_1 \vdash 3 :: (@) :: \square, \blacksquare \rangle$
$v_3 = \langle (+) :: (@) :: \sigma_1 \vdash 3 :: (@) :: \square, 1 :: v_1 :: \blacksquare \rangle$
$\sigma_1 = [\mathsf{x} \mapsto v_1]$
$\sigma_2 = \sigma_1[\mathsf{y} \mapsto v_2]$
$\sigma_3 = \sigma_2[\mathsf{z} \mapsto v_3]$

15.3 Interpreter

The following Scala code implements the syntax of KFAE: [3]

3: We omit the common part to FAE.

```
sealed trait Expr
...
case class Vcc(x: String, b: Expr) extends Expr
```

`Vcc(`x`, `e`)` represents vcc x in e.

In addition, we add a new variant of `Value` to represent first-class continuations.[4]

4: We omit the common part to FAE.

```
sealed trait Value
...
case class ContV(k: Cont) extends Value
```

Since the interpreter treats a continuation as a Scala function, a `ContV` instance has a single field whose type is `Cont`. `ContV(k)` represents a continuation `k` as a value.

We need to add the `Vcc` case to `interp` and revise the `App` case to handle continuation application properly. Consider the `Vcc` case first.

```
case Vcc(x, b) =>
  interp(b, env + (x -> ContV(k)), k)
```

The environment is extended with the continuation as a value, `ContV(k)`. Then, the body is evaluated under the extended environment.

The above code clearly shows why the interpreter needs to use CPS. Without CPS, the continuation cannot be accessed in the source code of the interpreter. There is no way to construct `ContV(k)`. By using CPS, the `interp` function always receives the continuation as an argument and becomes able to use the continuation to construct `ContV(k)`. CPS takes the key role in implementing an interpreter of a language that provides first-class continuations.

Now, let us fix the `App` case. The `interp` function must handle continuation applications in the `App` case.

```
case App(f, a) =>
  interp(f, env, fv =>
    interp(a, env, av => fv match {
      case CloV(x, e, fenv) =>
        interp(e, fenv + (x -> av), k)
      case ContV(nk) => nk(av)
    })
  )
```

When `fv` is a `CloV` instance, the interpreter behaves just like before. If `fv` is a `ContV` instance, the expression applies the continuation to the argument. The applied continuation is the field of `fv`, and the argument is `av`. Since the interpreter expresses a continuation with a Scala function, applying the Scala function, the field of `fv`, to `av` is enough. There is no `interp` call. It coincides with that Rule Red-App2-Cont never adds $\sigma \vdash e$ to the computation stack.

15.4 Use of First-Class Continuations

Imperative languages provide control diverters, such as `return`, `break`, and `continue`, to allow programmers to change control flows. However, KFAE supports only first-class continuations.

In fact, first-class continuations are the most general form of control flow manipulation. Continuations represent control flows of programs because the continuation at some point of execution is the remaining computation. Changing the current continuation is equivalent to changing the control flow by making the program compute different things. In KFAE, programmers can change the current continuation freely by calling continuations. Therefore, first-class continuations allow arbitrary changes of control flows. Expressions like `return` change control flows in a fixed way according to their semantics. On the other hand, programmers using KFAE can make first-class continuations with `vcc` and call them at any points. The expressivity of first-class continuations surpasses that of other control diverters.

Although first-class continuations are much more powerful than other control diverters, it is difficult to utilize first-class continuations to correctly implement desired control flow changes. To resolve the difficulty, language designers can provide other control diverters as syntacic sugar.

By doing so, the designers can make their language convenient for programmers while preventing the language from being complicated.

Return

A `return` expression makes a function immediately return. Instead of adding `return` to KFAE, we can desugar `return` to vcc and a continuation application.

If a programmer writes `return` e in the body of a function, `return` e can be desugared to `r` e, where `r` is just an identifier. [5] Then, `r` e is just an application expression.

5: Assume that the rest of the expression does not use `r` at all.

Since `r` is a free identifier yet, it must be bound somewhere. The correct way to bind `r` is to use vcc because applying `r` changes the control flow, which is possible only by a first-class continuation. Then, where should we put vcc? When there is no `return`, the only way to return from a function is to finish the evaluation of the function body. After the body is evaluated, its result is used for the remaining compuation, which is the continuation of the function body. Thus, applying the continuation of the function body to the return value is the same as returning from the function. After adding `return`, applying `r` to a value makes the function immediately return with the given value. It implies that `r` has to denote the continuation of the function body. Therefore, every function that contains `return` in the body needs to be desugared. An expression $\lambda x.e$ is desugared to $\lambda x.$vcc `r` in e.

The following example uses `return`:

$((\lambda$`x.(return 1) + x) 2) + 3`

By desugaring, the above expression becomes the following expression:

$((\lambda$`x.vcc r in (r 1) + x) 2) + 3`

While evaluating $(\lambda$`x.vcc r in (r 1) + x) 2`, `r` denotes $\square + 3$. When `r` is applied to 1, the original continuation disappears, and the only remainig computation is $1 + 3$. Therefore, the final result is 4. The result matches the expected semantics of `return`.

Break and Continue

Many imperative languages provide `break` and `continue`. Programmers use them inside loops to change control flows. A `break` expression immediately terminates the loop, and a `continue` expression makes the loop skip the current iteration and directly move on to the beginning of the next iteration.

Since KFAE lacks loops, we need to add loops first. Suppose that while0 e_1 e_2 evaluates e_2 repeatedly while e_1 evaluates to `0`. When the evaluation terminates, we define the result to be 0, which is just an arbitrarily chosen value.

Now, we can add `break` and `continue` to KFAE as syntactic sugar. They can occur only inside loops.

When a loop terminates, the continuation of the loop is applied to 0,

which is the result of the loop. Since `break` terminates the loop, it applies the continuation of the loop to 0. Thus, when `b` denotes the continuation of the loop, `break` can be desugared to `b` 0. [6] To make `b` the continuation of a loop, `vcc` that binds `b` should enclose the loop. Therefore, every loop containing `break` must be desugared. An expression `while0` e_1 e_2 is desugared to `vcc b in while0` e_1 e_2.

6: Assume that the rest of the expression does not use `b` at all.

For example, consider the following expression:

`while0` 0 `break`

This expression is desugared to the following expression:

`vcc b in while0` 0 (`b` 0)

Then, `b` is the continuation finishing the evaluation. Inside the loop, `b` is applied to 0, so the program terminates and produces 0 as a result. It coincides with the expected behavior of `break`. Even though the condition of the loop never changes from 0, the loop terminates due to the use of `break`.

While `break` terminates the loop, `continue` just skips the current iteration. It makes the program jump to the condition expression of the loop. Evaluating the condition expression is the continuation of the body of the loop because the condition is evaluated after the evaluation of the body. Therefore, an expression `while0` e_1 e_2 is desugared to `while0` e_1 (`vcc c in` e_2) when e_2 contains `continue`, and `continue` is desugared to `c` 0. [7]

7: Assume that the rest of the expression does not use `c` at all.

For example, consider the following expression:

`while0` 0 (`continue`; $(1 + \lambda x.x)$)

This expression is desugared to the following expression:

`while0` 0 (`vcc c in` ((`c` 0); $(1 + \lambda x.x)$))

At each iteration, when `c` 0 is evaluated, the result of whole `vcc c in` ((`c` 0); $(1 + \lambda x.x)$) becomes 0 without evaluating $1 + \lambda x.x$. Then, the loop proceeds to the next iteration without incurring a run-time error. Thus, the expression never terminates. It is what we expect from `continue`. Without `continue`, the expression causes a run-time error because it is impossible to add a number to a function. However, `continue` prevents the addition from being evaluated, so the expression never terminates.

Note that the selection of 0 in `c` 0 is completely arbitrary since the result of the loop body is never used. We may desugar `continue` to `c` 42 instead. It is different from the case of `break`, which must apply `b` to 0 to make the result of the loop 0.

15.5 Exercises

1. What is the result of the following expression?
   `vcc x in (vcc y in x (2 + (vcc z in y z)))` 8
2. Write the reduction steps of the following expression:
   `(vcc x in 42 + (x 2)) + 8`

First-Order Representation of Continuations

16

The previous chapter implements an interpreter of KFAE with first-class functions in Scala. The interpreter treats continuations as Scala functions. Since the interpreter uses CPS, continuations are passed from functions to functions. In addition, continuations sometimes need to be stored in `ContV` because KFAE supports first-class continuations. `ContV` instances are stored inside environments or returned from `interp`. Therefore, the interpreter relies on the fact that Scala provides first-class functions. Since functions are values in Scala, the interpreter can represent continuations with functions and uses them as values.

The use of first-class functions is problematic for some cases. First, low-level languages, such as C, lack first-class functions.[1] There must be another way to implement an interpreter of KFAE for those who use low-level languages. Second, functions do not give useful information. The only ability of functions is being applied to arguments. However, in particular programs like debuggers, it is necessary to figure out what a given first-class continuation does. The current implementation disallows such analysis on continuations. On the other hand, a `CloV` instance, which represents a closure, can give the exact information about the parameter, body, and environment of the closure. Alas, `ContV` instances do not have such capabilities.

1: C provides function pointers but not closures. Closures are necessary to represent continuations.

This chapter hows how we can represent continuations without first-class functions. By avoiding first-class functions, an interpreter of a language with first-class continuations can be written in low-level languages. In addition, if continuations are not functions and have specific structures instead, debuggers can analyze what a given continuation denotes.

16.1 First-Order Representation of Continuations

The following code shows the KFAE interpreter implemented in the previous chapter:[2]

2: Since the `Sub` case is similar to the `Add` case, this chapter omits the `Sub` case.

```
def interp(e: Expr, env: Env, k: Cont): Value = e match {
  case Num(n) => k(NumV(n))
  case Id(x) => k(lookup(x, env))
  case Fun(x, b) => k(CloV(x, b, env))
  case Add(e1, e2) =>
    interp(e1, env, v1 =>
      interp(e2, env, v2 =>
        k(add(v1, v2))
      )
    )
  case App(e1, e2) =>
    interp(e1, env, v1 =>
```

```
      interp(e2, env, v2 => v1 match {
        case CloV(x, e, fenv) =>
          interp(e, fenv + (x -> v2), k)
        case ContV(nk) => nk(v2)
      })
    )
  case Vcc(x, b) =>
    interp(b, env + (x -> ContV(k)), k)
}
```

We can change the implementation a bit by replacing `k(v)` with `continue(k, v)`.

```
def continue(k: Cont, v: Value): Value = k(v)

def interp(e: Expr, env: Env, k: Cont): Value = e match {
  case Num(n) => continue(k, NumV(n))
  case Id(x) => continue(k, lookup(x, env))
  case Fun(x, b) => continue(k, CloV(x, b, env))
  case Add(e1, e2) =>
    interp(e1, env, v1 =>
      interp(e2, env, v2 =>
        continue(k, add(v1, v2))
      )
    )
  case App(e1, e2) =>
    interp(e1, env, v1 =>
      interp(e2, env, v2 => v1 match {
        case CloV(x, e, fenv) =>
          interp(e, fenv + (x -> v2), k)
        case ContV(nk) => continue(nk, v2)
      })
    )
  case Vcc(x, b) =>
    interp(b, env + (x -> ContV(k)), k)
}
```

Since evaluating `k(v)` is everything `continue(k, v)` does, the new implementation behaves the same as the previous implementation. For now, the change seems needless, but it will become useful soon.

We need to know which continuations are used in the original interpreter to define values representing continuations. There are four sorts of continuations in the interpreter:

- `v1 => interp(e2, env, v2 => continue(k, add(v1, v2)))`
- `v2 => continue(k, add(v1, v2))`
- `v1 => interp(e2, env, v2 => v1 match ...)`
- `v2 => v1 match ...`

The first continuation, `v1 => interp(e2, env, v2 => continue(k, add(v1, v2)))`, is used after the evaluation of the left operand of addition. It evaluates the right operand, calculates the sum, and passes the sum to the continuation of the entire addition. The parameter `v1` denotes the

value of the left operand. The function body contains three free variables: `e2`, `env`, and `k`. `e2` is the right operand; `env` is the current environment; `k` is the continuation of the addition. If the values of the free variables are determined, the behavior of the continuation is also determined. Therefore, `(e2, env, k)`, which is a triple of an expression, an environment, and a continuation, can represent the continuation.

Currently, `continue` continues the evaluation with a given function, which represents the continuation. Since the continuation is a Scala function, it can be directly applied to a given value. However, if we use a triple to represent the continuation instead of a function, it cannot be applied to a value. We need a new way to continue evaluation when a continuation and a value are given. The clue already exists—look at the body of the function representing a continuation. When the function is applied to `v1`, the result is `interp(e2, env, v2 => continue(k, add(v1, v2)))`. Now, `(e2, env, k)` and `v1` are provided instead of the function and `v1`. It is enough to evaluate `interp(e2, env, v2 => continue(k, add(v1, v2)))` with `v1`, `e2`, `env`, and `k`. It evaluates the same thing as the original function application.

Below compare the previous and current strategies:

- Previous: `v1 => interp(e2, env, v2 => continue(k ,add(v1, v2)))` and `v1` are given. Then, evaluate `interp(e2, env, v2 => continue(k, add(v1, v2)))` by applying `v1 => interp(e2, env, v2 => continue(k ,add(v1, v2)))` to `v1`.
- Current: `(e2, env, k)` and `v1` are given. Then, evaluate `interp(e2, env, v2 => continue(k, add(v1, v2)))` with `e2`, `env`, `k`, and `v1`.

Both strategies evaluate `interp(e2, env, v2 => continue(k, add(v1, v2)))` in the end. While the previous strategy represents continuations with functions, the current strategy represents continuations with triples.

Once you understand the first sort of continuations, the remaining ones are straightforward. The second continuation, `v2 => continue(k, add(v1, v2))`, is used after the evaluation of the right operand of addition. It calculates the sum of the operands and passes the sum to the continuation of the addition. The parameter `v2` denotes the value of the right operand. The function body contains two free variables: `k` and `v1`. `k` is the continuation of the addition; `v1` is the value of the left operand. In a similar fashion, `v1` and `k` are enough to determine what the continuation does. Therefore, `(v1, k)`, a pair of a value and a continuation, can represent the continuation. To continue the evaluation when `(v1, k)` and `v2` are given, `continue(k, add(v1, v2))` needs to be evaluated.

- Previous: `v2 => continue(k, add(v1, v2))` and `v2` are given. Then, evaluate `continue(k, add(v1, v2))` by applying `v2 => continue(k, add(v1, v2))` to `v2`.
- Current: `(v1, k)` and `v2` are given. Then, evaluate `continue(k, add(v1, v2))` with `v1`, `k`, and `v2`.

The third continuation, `v1 => interp(e2, env, v2 => v1 match ...)`, is used after the evaluation of the expression at the function position of a function application. It evaluates an argument and applies a function (or a continuation) to the argument. `v1` denotes the value of the expres-

sion at the function position. The body of the function representing the continuation contains three free variables: `e2`, `env`, and `k`.[3] `e2` is the expression at the argument position; `env` is the current environment; `k` is the continuation of the application. Therefore, `e2`, `env`, and `k` determine what the continuation does, and `(e2, env, k)`, a triple of an expression, an environment, and a continuation, can represent the continuation. Continuing the evaluation is evaluating `interp(e2, env, v2 => v1 match ...)`, which can be done with `(e2, env, k)` and `v1`.

3: `k` is in ....

- Previous: `v1 => interp(e2, env, v2 => v1 match ...)` and `v1` are given. Then, evaluate `v1 => interp(e2, env, v2 => v1 match ...)` by applying `v1 => interp(e2, env, v2 => v1 match ...)` to `v1`.
- Current: `(e2, env, k)` and `v1` are given. Then, evaluate `interp(e2, env, v2 => v1 match ...)` with `e2`, `env`, `k`, and `v1`.

The fourth continuation, `v2 => v1 match ...`, is used after the evaluation of the argument of a function application. It applies a function (or a continuation) to the argument. `v2` denotes the value of the argument. The body of the function representing the continuation contains two free variables: `v1` and `k`.[4] `v1` is the value of the expression at the function position; `k` is the continuation of the application. Therefore, `(v1, k)`, a pair of a value and a continuation, can represent the continuation. Continuing the evaluation is evaluating `v1 match ...` with `(v1, k)` and `v2`.

4: `k` is in ....

- Previous: `v2 => v1 match ...` and `v2` are given. Then, evaluate `v1 match ...` by applying `v2 => v1 match ...` to `v2`.
- Current: `(v1, k)` and `v2` are given. Then, evaluate `v1 match ...` with `v1`, `k`, and `v2`.

In fact, there is one more continuation, which does not appear in the implementation of `interp`. It is the one that is represented as the identity function and is passed to `interp` in the beginning. The identity function returns a given argument without any changes. No additional information is necessary to determine the behavior of the continuation. Therefore, `()`, the zero-length tuple (the `Unit` value in Scala) can represent the continuation. To continue the evaluation with the continuation and a value `v`, it is enough to give `v` as the result.

- Previous: An identity function and `v` are given. Then, evaluate `v` by applying the identity function to `v`.
- Current: `()` and `v` are given. Then, evaluate `v` with `v`.

In summary, the KFAE interpreter uses continuations of the following five sorts:

- `(e2: Expr, env: Env, k: Cont)`
- `(v1: Value, k: Cont)`
- `(e2: Expr, env: Env, k: Cont)`
- `(v1: Value, k: Cont)`
- `()`

Note that the first and the third are different even though they look the same. The first continuation computes `interp(e2, env, v2 => continue(k, add(v1, v2)))` with its data, while the third continuation computes `interp(e2, env, v2 => v1 match ...)` with its data.

Similarly, the second and the fourth are diffent as well. The second computes `continue(k, add(v1, v2))`, while the fourth computes `v1 match ...`.

As explained in Chapter 5, an ADT is the best way to implement a type that consists of values of various shapes. Thus, `Cont` can be newly defined with a `sealed trait` and `case classes` as follows:

```
sealed trait Cont
case class AddSecondK(e2: Expr, env: Env, k: Cont) extends Cont
case class DoAddK(v1: Value, k: Cont) extends Cont
case class AppArgK(e2: Expr, env: Env, k: Cont) extends Cont
case class DoAppK(v1: Value, k: Cont) extends Cont
case object MtK extends Cont
```

The names of the classes do not matter, though they are named carefully so that the names can reflect what they are for. The important things are data carried by each continuation. Following the Scala convention, the last sort, which can be represented by the empty tuple, is now represented by a singleton object. One may use `case class MtK() extends Cont` instead without changing the semantics, but the singleton object is more efficient than the case class from the implementation perspective. Now, the implementation of continuations does not require first-class functions.

Now, we need to revise the `continue` function. The previous implementation is `def continue(k: Cont, v: Value): Value = k(v)`. It works because `Cont` is a function type before our change. However, `Cont` is not a function now, and `continue` needs a fix. In fact, we already know everything to make a correct fix. Previously, `continue` applies `k` to `v` when `k` and `v` are given. Now, it should check `k` and do the correct computation according to the data in `k`. Below is the repetition of the previous explanations, but with the names of the case classes and object.

- `AddSecondK(e2, env, k)` and `v1` are given. Then, evaluate `interp(e2, env, v2 => continue(k, add(v1, v2)))` with `e2`, `env`, `k`, and `v1`.
- `DoAddK(v1, k)` and `v2` are given. Then, evaluate `continue(k, add(v1, v2))` with `v1`, `k`, and `v2`.
- `AppArgK(e2, env, k)` and `v1` are given. Then, evaluate `interp(e2, env, v2 => v1 match ...)` with `e2`, `env`, `k`, and `v1`.
- `DoAppK(v1, k)` and `v2` are given. Then, evaluate `v1 match ...` with `v1`, `k`, and `v2`.
- `MtK` and `v` are given. Then, evaluate `v` with `v`.

The first and third explanations still pass functions to `interp` even though continuations are not functions anymore. They need small changes. Now, `DoAddK(v1, k)` represents `v2 => continue(k, add(v1, v2))`, and `DoAppK(v1, k)` represents `v2 => v1 match ...`.

- `AddSecondK(e2, env, k)` and `v1` are given. Then, evaluate `interp(e2, env, DoAddK(v1, k))` with `e2`, `env`, `k`, and `v1`.
- `AppArgK(e2, env, k)` and `v1` are given. Then, evaluate `interp(e2, env, DoAppK(v1, k))` with `e2`, `env`, `k`, and `v1`.

The new implementation of `continue` is as follows:

```
def continue(k: Cont, v: Value): Value = k match {
  case AddSecondK(e2, env, k) => interp(e2, env, DoAddK(v, k))
  case DoAddK(v1, k) => continue(k, add(v1, v))
  case AppArgK(e2, env, k) => interp(e2, env, DoAppK(v, k))
  case DoAppK(v1, k) => v1 match {
    case CloV(x, e, fenv) =>
      interp(e, fenv + (x -> v), k)
    case ContV(nk) => continue(nk, v)
  }
  case MtK => v
}
```

The code is straightforward since it is exactly the same as the explanation.

The `interp` function also needs a fix to follow the new definition of `Cont`:

```
def interp(e: Expr, env: Env, k: Cont): Value = e match {
  case Num(n) => continue(k, NumV(n))
  case Id(x) => continue(k, lookup(x, env))
  case Fun(x, b) => continue(k, CloV(x, b, env))
  case Add(e1, e2) => interp(e1, env, AddSecondK(e2, env, k))
  case App(e1, e2) => interp(e1, env, AppArgK(e2, env, k))
  case Vcc(x, b) => interp(b, env + (x -> ContV(k)), k)
}
```

Only the `Add` and `App` cases are different from before. The `Add` case uses `AddSecondK(e2, env, k)` to represent `v1 => interp(e2, env, v2 => continue(k, add(v1, v2)))`; the `App` case uses `AppArgK(e2, env, k)` to represent `v1 => interp(e2, env, v2 => v1 match ...)`.

Note that `MtK` does not appear in `interp`. `MtK` is used to call `interp` in the beginning. One should write `interp(`*e*`, Map(), MtK)` to evaluate *e*.

16.2 Big-Step Semantics of KFAE

Representing continuations without first-class functions additionally gives us a clue to define the big-step semantics of KFAE. This section defines the big-step semantics of KFAE from the new interpreter implementation.

First, we can define continuations inductively. It is straightforward from the implementation.

```
sealed trait Cont
case class AddSecondK(e2: Expr, env: Env, k: Cont) extends Cont
case class DoAddK(v1: Value, k: Cont) extends Cont
case class AppArgK(e2: Expr, env: Env, k: Cont) extends Cont
case class DoAppK(v1: Value, k: Cont) extends Cont
case object MtK extends Cont
```

The above ADT can be formalized as below:

$$\kappa \ ::= \ [\square + (e, \sigma)] :: \kappa \mid [v + \square] :: \kappa \mid [\square \ (e, \sigma)] :: \kappa \mid [v \ \square] :: \kappa \mid [\square]$$

where the metavariable κ ranges over continuations. Note that $\square$ in the definition is different from $\square$ used by the small-step semantics. While $\square$ denotes the empty computation stack in the small-step semantics, $\square$ in the above definition is just a part of the notation. It does not have any formal meaning but conceptually represents a hole in an intuitive way.

Previous chapters define only one sort of a proposition for the big-step semantics because their interpreters have only `interp`. On the other hand, the interpreter of this chapter has both `interp` and `continue`, and each of them has a different role from the other. Therefore, we formalize each function with a different sort of a proposition. The first sort is $\sigma, \kappa \vdash e \Rightarrow v$, which denotes that the result of `interp(`e`, `σ`, `κ`)` is v. The other is $v_1 \mapsto \kappa \Downarrow v_2$, which denotes that the result of `continue(`κ`, `v_1`)` is v_2.

Let us define inference rules related to `interp` first.

```
def interp(e: Expr, env: Env, k: Cont): Value = e match {
  case Num(n) => continue(k, NumV(n))
  case Id(x) => continue(k, lookup(x, env))
  case Fun(x, b) => continue(k, CloV(x, b, env))
  case Add(e1, e2) => interp(e1, env, AddSecondK(e2, env, k))
  case App(e1, e2) => interp(e1, env, AppArgK(e2, env, k))
  case Vcc(x, b) => interp(b, env + (x -> ContV(k)), k)
}
```

Each case of the `interp` function produces a single inference rule.

$$\frac{n \mapsto \kappa \Downarrow v}{\sigma, \kappa \vdash n \Rightarrow v} \quad \text{[Interp-Num]}$$

The conclusion, $\sigma, \kappa \vdash n \Rightarrow v$, denotes that the result of `interp(`n`, `σ`, `κ`)` is v. The result of `interp(`n`, `σ`, `κ`)` is the same as that of `continue(`κ`, NumV(`n`))`. $n \mapsto \kappa \Downarrow v$ denotes that the result of `continue(`κ`, NumV(`n`))` is v.

$$\frac{x \in \mathit{Domain}(\sigma) \qquad \sigma(x) \mapsto \kappa \Downarrow v}{\sigma, \kappa \vdash x \Rightarrow v} \quad \text{[Interp-Id]}$$

$$\frac{\langle \lambda x.e, \sigma \rangle \mapsto \kappa \Downarrow v}{\sigma, \kappa \vdash \lambda x.e \Rightarrow v} \quad \text{[Interp-Fun]}$$

The rules for variables and functions are similar to the rule for integers.

$$\frac{\sigma, [\square + (e_2, \sigma)] :: \kappa \vdash e_1 \Rightarrow v}{\sigma, \kappa \vdash e_1 + e_2 \Rightarrow v} \quad \text{[Interp-Add]}$$

The conclusion, $\sigma, \kappa \vdash e_1{+}e_2 \Rightarrow v$, denotes that the result of `interp(Add(`e_1`,`e_2`), `σ`, `κ`)` is v. The result of `interp(Add(`e_1`,`e_2`), `σ`, `κ`)` is the same as that of `interp(`e_1`, `σ`, AddSecondK(`e_2`, `σ`, `κ`))`. Note that $[\square + (e_2, \sigma)] :: \kappa$ denotes `AddSecondK(`e_2`, `σ`, `κ`)`. $\sigma, \kappa' \vdash e_1 \Rightarrow v$ denotes that the result of `interp(`e_1`, `σ`, `κ'`)` is v, where κ' is $[\square + (e_2, \sigma)] :: \kappa$.

$$\frac{\sigma, [\square\ (e_2, \sigma)] :: \kappa \vdash e_1 \Rightarrow v}{\sigma, \kappa \vdash e_1\ e_2 \Rightarrow v} \quad \text{[Interp-App]}$$

The rule for function application is similar to the rule for addition.

$$\frac{\sigma[x \mapsto \kappa], \kappa \vdash e \Rightarrow v}{\sigma, \kappa \vdash \mathsf{vcc}\ x;\ e \Rightarrow v} \quad \text{[Interp-Vcc]}$$

The conclusion, $\sigma, \kappa \vdash \mathsf{vcc}\ x;\ e \Rightarrow v$, denotes that the result of `interp(Vcc(`x`, `e`), `σ`, `κ`)` is v. The result of `interp(Vcc(`x`, `e`), `σ`, `κ`)` is the same as that of `interp(`e`, `$\sigma[x \mapsto \kappa]$`, `κ`)`. $\sigma[x \mapsto \kappa], \kappa \vdash e \Rightarrow v$ denotes that the result of `interp(`e`, `$\sigma[x \mapsto \kappa]$`, `κ`)` is v.

Now, we define inference rules related to `continue`.

```
def continue(k: Cont, v: Value): Value = k match {
  case AddSecondK(e2, env, k) => interp(e2, env, DoAddK(v, k))
  case DoAddK(v1, k) => continue(k, add(v1, v))
  case AppArgK(e2, env, k) => interp(e2, env, DoAppK(v, k))
  case DoAppK(v1, k) => v1 match {
    case CloV(xv1, ev1, sigmav1) =>
      interp(ev1, sigmav1 + (xv1 -> v), k)
    case ContV(k) => continue(k, v)
  }
  case MtK => v
}
```

Each case of the `continue` function also produces a single inference rule. The only exception is the `DoAppK` case because it requires two rules: one for the `CloV` case and the other for the `ContV` case.

$$\frac{\sigma, [v_1 + \square] :: \kappa \vdash e_2 \Rightarrow v_2}{v_1 \mapsto [\square + (e_2, \sigma)] :: \kappa \Downarrow v_2} \quad \text{[Continue-AddSecondK]}$$

The conclusion, $v_1 \mapsto [\square + (e_2, \sigma)] :: \kappa \Downarrow v_2$, denotes that the result of `continue(AddSecondK(`e_2`, `σ`, `κ`), `v_1`)` is v_2. The result of `continue(AddSecondK(`e_2`, `σ`, `κ`), `v_1`)` is the same as that of `interp(`e_2`, `σ`, DoAddK(`v_1`, `κ`))`. Note that $[v_1 + \square]$ denotes `DoAddK(`v_1`, `κ`)`. $\sigma, \kappa' \vdash e_2 \Rightarrow v_2$ denotes that the result of `interp(`e_2`, `σ`, `κ'`)` is v_2 where κ' is $[v_1 + \square]$.

$$\frac{n_1 + n_2 \mapsto \kappa \Downarrow v}{n_2 \mapsto [n_1 + \square] :: \kappa \Downarrow v} \quad \text{[Continue-DoAddK]}$$

The conclusion, $n_2 \mapsto [n_1 + \square] :: \kappa \Downarrow v$, denotes that the result of `continue(DoAddK(NumV(`n_1`), `κ`), NumV(`n_2`))` is v. The result of `continue(DoAddK(NumV(`n_1`), `κ`), NumV(`n_2`))` is the same as that of `continue(`κ`, add(NumV(`n_1`),`

`NumV(`n_2`)))`. Note that `add(NumV(`n_1`), NumV(`n_2`))` equals `NumV(`$n_1 + n_2$`)`.

$n_1 + n_2 \mapsto \kappa \Downarrow v$ denotes that the result of `continue(`κ`, NumV(`$n_1 + n_2$`))` is v.

$$\frac{\sigma, [v_1\ \square] :: \kappa \vdash e \Rightarrow v_2}{v_1 \mapsto [\square\ (e, \sigma)] :: \kappa \Downarrow v_2} \quad \text{[Continue-AppArgK]}$$

This rule is similar to the rule when the continuation is $[\square + (e_2, \sigma)]$.

$$\frac{\sigma[x \mapsto v_2], \kappa \vdash e \Rightarrow v}{v_2 \mapsto [\langle \lambda x.e, \sigma \rangle\ \square] :: \kappa \Downarrow v} \quad \text{[Continue-DoAppK-CloV]}$$

The conclusion, $v_2 \mapsto [\langle \lambda x.e, \sigma \rangle\ \square] :: \kappa \Downarrow v$, denotes that the result of `continue(DoAppK(CloV(`x`, `e`, `σ`), `κ`), `v_2`)` is v. The result of `continue(DoAppK(CloV(`x`, `e`, `σ`), `κ`), `v_2`)` is the same as that of `interp(`e`, `$\sigma[x \mapsto v_2]$`, `κ`)`. $\sigma[x \mapsto v_2], \kappa \vdash e \Rightarrow v$ denotes that the result of `interp(`e`, `$\sigma[x \mapsto v_2]$`, `κ`)` is v.

$$\frac{v_2 \mapsto \kappa_1 \Downarrow v}{v_2 \mapsto [\kappa_1\ \square] :: \kappa \Downarrow v} \quad \text{[Continue-DoAppK-ContV]}$$

The conclusion, $v_2 \mapsto [\kappa_1\ \square] :: \kappa \Downarrow v$, denotes that the result of `continue(DoAppK(ContV(`κ_1`), `κ`), `v_2`)` is v. The result of `continue(DoAppK(ContV(`κ_1`), `κ`), `v_2`)` is the same as that of `continue(`κ_1`, `v_2`)`. $v_2 \mapsto \kappa_1 \Downarrow v$ denotes that the result of `continue(`κ_1`, `v_2`)` is v.

$$v \mapsto [\square] \Downarrow v \quad \text{[Continue-MtK]}$$

The conclusion, $v \mapsto [\square] \Downarrow v$, denotes that the result of `continue(MtK, `v`)` is v. The result of `continue(MtK, `v`)` is actually v.

Nameless Representation of Expressions

17

In previous chapters, languages distinguish different variables by naming them with different identifiers. For example, λx.λy.x is a function that takes an argument twice and returns the first argument. Since two parameters have different names, one can easily conclude that the first argument is the result. The first parameter is named x, and the second parameter is named y. Therefore, x in the function body denotes the first parameter.

Naming variables is an intuitive and practically useful way to represent variables. However, it becomes problematic in some cases like formalizing the semantics of languages and implementing interpreters and compilers, which take source code as input.

First, two variables may not be distinguished when their names are the same. Environments can easily deal with variables of the same name well, but substitution is often used instead of environments to define the semantics of languages. For instance, defining the semantics of function applications with substitutions is as follows: evaluating (λx.x + x) 1 is the same as evaluating 1 + 1, which is obtained by substituting x with 1 in the function body x + x.[1] In fact, it is difficult to define the semantics correctly with substitutions. Consider the expression (λf.λx.f) λy.x. By applying the same principle, evaluating the expression is the same as evaluating λx.λy.x, which is obtained by substituting f with λy.x in λx.f. Alas, it is wrong. x in the original argument λy.x is a free identifier, while x in λx.λy.x is a bound occurrence. The meaning of x before and after the substitution is completely different. This example shows that the current semantics is incorrect, and the root cause of the problem is two different variables of the same name x.

1: Since the main purpose of mentioning substitutions is explaining the problem of naming, we do not formally define substitutions. To find more about substitution, see Exercise 8 of Chapter 9.

Second, names hinder us from checking the semantic equivalence of expressions. For example, both λx.x and λy.y are identity functions. However, a naïve syntactic check cannot prove the semantic equivalence of them, i.e. that their behaviors are the same, because the first expression names the parameter x, while the second expression names the parameter y. The ability to check semantic equivalence is valuable in many places. Consider optimization of expressions.

```
val f = λx.x;
val g = λy.y;
(f 1) + (g 2)
```

The above expression defined the functions f and g and, then, evaluate (f 1) + (g 2). f and g are semantically equivalent, but the names of their parameters are different. If a compiler is aware of their equivalence, it can reduce the size of the program by modifying the expression like below:

```
val f = λx.x;
(f 1) + (f 2)
```

As the example shows, comparing the semantic equivalence of expressions is an important problem, but naming variables is not a good way for this purpose.

For these reasons, names are often problematic in programming languages. Multiple solutions have been proposed to resolve the issue. This chapter introduces de Bruijn indices, which are one of those solutions. *De Bruijn indices* represent variables with indices, not names. This chapter shows how de Bruijn indices can be used in FAE. Note that FAE is just one possible use case of de Bruijn indices. De Bruijn indices can be used anywhere names lead to a problem.

17.1 De Bruijn Indices

De Bruijn indices represent variables with indices, which are natural numbers. The number of λ between a bound occurrence and the corresponding binding occurrence represents the binding occurrence. For instance, $\lambda.\underline{0}$ is the nameless version of λx.x. $\lambda.\underline{0}$ is a function with one parameter. Its body is $\underline{0}$, which differs from a natural number 0. $\underline{0}$ denotes a variable whose distance from its definition is zero. The distance means the number of λ. Therefore, the parameter of $\lambda.\underline{0}$ is the one that $\underline{0}$ is bound to. In a similar fashion, $\lambda.\lambda.\underline{1}$ is the nameless version of λx.λy.x. $\lambda.\lambda.\underline{1}$ is a function with one parameter and the body expression $\lambda.\underline{1}$. $\lambda.\underline{1}$ also is a function with one parameter. Its body is $\underline{1}$, which is a variable whose distance from the definition is one. Thus, the parameter of $\lambda.\underline{1}$ cannot be denoted by $\underline{1}$. There is no λ between the parameter and $\underline{1}$. $\underline{1}$ denotes the parameter of $\lambda.\lambda.\underline{1}$ because there is one λ in between. The following table shows various examples of de Bruijn indices.

With names	Without names
λx.x	$\lambda.\underline{0}$
λx.λy.x	$\lambda.\lambda.\underline{1}$
λx.λy.y	$\lambda.\lambda.\underline{0}$
λx.λy.x + y	$\lambda.\lambda.\underline{1} + \underline{0}$
λx.λy.x + y + 42	$\lambda.\lambda.\underline{1} + \underline{0} + 42$
λx.(x λy.(x y))	$\lambda.(\underline{0}\ \lambda.(\underline{1}\ \underline{0}))$
λx.((λy.x) (λz.x))	$\lambda.((\lambda.\underline{1})\ (\lambda.\underline{1}))$

It is important to notice that different indices can denote the same variable, and the same indices can denote different variables. Consider the second example from the bottom. The first $\underline{0}$ in $\lambda.(\underline{0}\ \lambda.(\underline{1}\ \underline{0}))$ denotes x of the original expression. At the same time, $\underline{1}$ also denotes x of the original expression. On the other hand, the second $\underline{0}$ denotes y of the original expression. The distance from the definition depends on the location of a variable. Since de Bruijn indices represent variables with their distances, the indices of a single variable can vary among places.

Note that expressions should be treated as trees, not strings, to calculate the distances. Consider the last example. There are two λ's between

the last x and its definition when the expression is written as a string. However, when the abstract syntax tree representing the expression is considered, there is only one λ in between. Therefore, the index of the last x is $\underline{1}$, not $\underline{2}$. We usually write expressions as strings for convenience, but they always have tree structures in fact.

De Bruijn indices successfully resolve the issues arising from names. Consider the comparison of expressions. $\lambda \mathsf{x}.\mathsf{x}$ and $\lambda \mathsf{y}.\mathsf{y}$ are semantically equivalent but syntactically different expressions. Both become $\lambda.\underline{0}$ when de Bruijn indices are used. By the help of de Bruijn indices, a simple syntactic check will find out that two expressions are equal.

Now, let us define the procedure that transforms named expressions into nameless expressions. It helps readers understand de Bruijn indices. At the same time, the procedure is practically valuable. Use of names is the best way to denote variables for programmers. Therefore, expressions written by programmers have names. On the other hand, programs like interpreters and compilers sometimes need to use de Bruijn indices to represent variables. In such cases, the procedure is a part of the interpreter/compiler implementation.

First, we define indices as follows:

$$i \in \mathbb{N}$$

where the metavariable i ranges over indices.

Then, we can define nameless expressions as follows: [2]

$$e ::= \underline{i} \mid \lambda.e \mid e\ e \mid n \mid e + e$$

In nameless expressions, natural numbers represent variables. Those numbers have underlines and, therefore, cannot be confused with integers. A lambda abstraction $\lambda.e$ lacks the name of its parameter. Note that $\lambda.e$ does have a single parameter. It is not a function with zero parameters.

A context, which is a finite partial function from names to natural numbers, takes an important role during the transformation. A context gives the distance between a variable and its definition.

$$\chi \in Id \xrightarrow{\text{fin}} \mathbb{N}$$

The metavariable χ ranges over contexts.

Let $[e]\chi$ be a nameless expression representing e under a context χ. The definition of $[e]\chi$ is as follows:

$$\begin{aligned}
[x]\chi &= \underline{i} \text{ if } \chi(x) = i \\
[\lambda x.e]\chi &= \lambda.[e]\chi' \text{ where } \chi' = (\uparrow \chi)[x \mapsto 0] \\
[e_1\ e_2]\chi &= [e_1]\chi\ [e_2]\chi \\
[n]\chi &= n \\
[e_1 + e_2]\chi &= [e_1]\chi + [e_2]\chi
\end{aligned}$$

$[x]\chi$ is the result of transforming x. A natural number represents a variable, and the natural number can be found in χ. Therefore, when

2: This chapter uses e to denote both named expressions and nameless expressions. Strictly speaking, two different metavariables should be introduced to denote each sort of an expression separately. However, for brevity, we abuse the notation and use e for both sorts of an expression.

$\chi(x)$ is i, x is transformed into $\underline{i}$.

$[\lambda x.e]\chi$ is the result of transforming $\lambda x.e$ and should look like $\lambda.e$. However, e uses names and, thus, needs to be transformed. χ is not the correct context for the transformation of e. First, it lacks the information of x. If x appears in e without any function definitions, there is no λ between the use and the definition. The context must know that the index of x is 0. In addition, indices in χ need changes. Suppose that x' is in χ and its index is 0. If x' occurs in e, its index is not 0 anymore. Since e is the body of $\lambda x.e$, there is one λ between x' and its definition. During the transformation of e, the index of x' is 1, not 0. Similarly, if there is a variable whose index is 1 in χ, its index must be 2 during the transformation of e. In conclusion, every index in χ has to increase by one. $\uparrow \chi$ denotes the context same as χ but whose indices are one larger. The context used during the transformation of e is $(\uparrow \chi)[x \mapsto 0]$. $[\lambda x.e]\chi$ is $\lambda.[e]\chi'$ where χ' is $(\uparrow \chi)[x \mapsto 0]$.

The remaining cases are straightforward. The transformations of $e_1\ e_2$ and $e_1 + e_2$ are recursively defined. Since n does not contain variables, n itself is the result.

Below shows how $\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x} + \mathsf{y}$ is transformed by the procedure. In the beginning, the context is empty because there is no variable yet.

$$
\begin{aligned}
& [\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x} + \mathsf{y}]\emptyset \\
= \ & \lambda.[\lambda \mathsf{y}.\mathsf{x} + \mathsf{y}][\mathsf{x} \mapsto 0] \\
= \ & \lambda.\lambda.[\mathsf{x} + \mathsf{y}][\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 0] \\
= \ & \lambda.\lambda.[\mathsf{x}][\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 0] + [\mathsf{y}][\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 0] \\
= \ & \lambda.\lambda.\underline{1} + [\mathsf{y}][\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 0] \\
= \ & \lambda.\lambda.\underline{1} + \underline{0}
\end{aligned}
$$

Now, let us implement the procedure in Scala. For named expressions, we can reuse the previous implementation. The following code defines nameless expressions:

```
object Nameless {
  sealed trait Expr
  case class Num(n: Int) extends Expr
  case class Add(l: Expr, r: Expr) extends Expr
  case class Id(i: Int) extends Expr
  case class Fun(e: Expr) extends Expr
  case class App(f: Expr, a: Expr) extends Expr
}
```

`Id(`i`)` represents $\underline{i}$, and `Fun(`e`)` represent $\lambda.e$.

Note that nameless expressions are defined in the `Nameless` singleton object. Therefore, outside the object, `Expr` denotes the type of named expressions, while `Nameless.Expr` denotes the type of nameless expressions. Similarly, `Id` represents a variable represented by a name, while `Nameless.Id` represents a variable represented by an index.

```
type Ctx = Map[String, Int]
```

`Ctx`, the type of a context, is a map from strings to integers.

The following `transform` function recursively transforms a named expression into a nameless expression:

```
def transform(e: Expr, ctx: Ctx): Nameless.Expr = e match {
  case Id(x) => Nameless.Id(ctx(x))
  case Fun(x, e) =>
    val nctx = ctx.map{ case (x, i) => x -> (i + 1) } + (x -> 0)
    Nameless.Fun(transform(e, nctx))
  case App(f, a) =>
    Nameless.App(transform(f, ctx), transform(a, ctx))
  case Num(n) => Nameless.Num(n)
  case Add(l, r) =>
    Nameless.Add(transform(l, ctx), transform(r, ctx))
}
```

The function exactly looks like its mathematical definition, so it is easy to understand the code.

Lists can replace maps in the implementation. A context is a list of names, and the index of a name is the location of the name in the list. Lists simplify the implementation. When a name is added to a context, its index is always zero. It means that the name is the head of the list. Adding a name is the same as making the head of the list be the name. Increasing every index by one is the same as moving each name backward by one slot. Therefore, if a context is a list, prepending a new name in front of the list does everything we need to extend the context. For example, consider a context containing `x` and `y`. Let the indices of `x` and `y`, respectively, be 0 and 1. The context is represented by `List("x", "y")`. It is enough to prepend `z` to the list to add `z` to the context. The resulting list is `List("z", "x", "y")`—`z` at index 0, `x` at index 1, and `y` at index 2. Since `z` is the new name, its index should be 0. At the same time, the indices of `x` and `y` should be greater by one than before. The new list does represent the new context well.

To use lists instead, we change the definition of `Ctx`.

```
type Ctx = List[String]
```

Now, `Ctx` is a list of strings. Then, we can revise `transform` accordingly.

```
def transform(e: Expr, ctx: Ctx): Nameless.Expr = e match {
  case Id(x) => Nameless.Id(ctx.indexOf(x))
  case Fun(x, e) => Nameless.Fun(transform(e, x :: ctx))
  case App(f, a) =>
    Nameless.App(transform(f, ctx), transform(a, ctx))
  case Num(n) => Nameless.Num(n)
  case Add(l, r) =>
    Nameless.Add(transform(l, ctx), transform(r, ctx))
}
```

The `Id` case needs to calculate the location of a given variable in a given context. It is enough to use `indexOf`. In the `Fun` case, `x :: ctx` is everything we need to add `x` to `ctx`.

17.2 Evaluation of Nameless Expressions

Evaluation of nameless expressions is similar to evaluation of named expressions. The definition of a value has a minor difference:

$$v ::= n \mid \langle \lambda.e, \sigma \rangle$$

As lambda abstractions lack parameter names, closures also lack parameter names.

The definition of an environment also has an insignificant difference:

$$\sigma \in \mathbb{N} \xrightarrow{\text{fin}} V$$

Environments are finite partial functions from indices, which are natural numbers, to values.

Now, let us define the inference rules.

Rule Id
If i is in the domain of σ,
then $\underline{i}$ evaluates to $\sigma(i)$ under σ.

$$\frac{i \in \mathit{Domain}(\sigma)}{\sigma \vdash \underline{i} \Rightarrow \sigma(i)} \quad [\textsc{Id}]$$

The value of a variable can be found in a given environment.

Rule Fun
$\lambda.e$ evaluates to $\langle \lambda.e, \sigma \rangle$ under σ.

$$\sigma \vdash \lambda.e \Rightarrow \langle \lambda.e, \sigma \rangle \quad [\textsc{Fun}]$$

A lambda abstraction evaluates to a closure without evaluating anything.

Rule App
If
e_1 evaluates to $\langle \lambda.e, \sigma' \rangle$ under σ,
e_2 evaluates to v_2 under σ, and
e evaluates to v under $(\uparrow \sigma')[0 \mapsto v_2]$,
then
$e_1\ e_2$ evaluates to v under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow \langle \lambda.e, \sigma' \rangle \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad (\uparrow \sigma')[0 \mapsto v_2] \vdash e \Rightarrow v}{\sigma \vdash e_1\ e_2 \Rightarrow v} \quad [\textsc{App}]$$

Evaluation of $e_1\ e_2$ evaluates both e_1 and e_2. Then, the body of the closure is evaluated under the environment captured by the closure with the value of the argument. If the parameter is used in the body, there is no λ

between the use and the definition. Its index is 0. Therefore, the value of the argument has the index 0 in the new environment. In addition, every index in the environment of the closure needs a change. Let a value v correspond to the index 0. The value is not the value of the argument, so it cannot correspond to the index 0 anymore. As λ from the closure exists between the use and the definition, the index should increase by one. By the same principle, every index in the environment increases by one. Since $\uparrow \sigma'$ denotes the context same as σ' but whose indices are one larger, the body of the closure is evaluated under $(\uparrow \sigma')[0 \mapsto v_2]$.

The rules for integers and addition are omitted because they are the same as those of FAE.

This new semantics for nameless expressions is equivalent to the previous semantics for named expressions. Let e be a named expression. The result of evaluating e is the same as evaluating e' where e' is the nameless expression obtained by transforming e. [3] Mathematically, the following proposition is true:

$\forall e. \forall v. (\emptyset \vdash e \Rightarrow v) \leftrightarrow (\emptyset \vdash [e]\emptyset \Rightarrow v)$.

Let us implement an interpreter of nameless expressions in Scala. Below is the definitions of values and environments.[4]

```
type Env = List[Value]

sealed trait Value
case class NumV(n: Int) extends Value
case class CloV(e: Expr, env: Env) extends Value
```

An environment is a list of values. As shown by the implementation of `transform`, lists are simpler than maps from integers to values.

```
def interp(e: Expr, env: Env): Value = e match {
  case Id(i) => env(i)
  case Fun(e) => CloV(e, env)
  case App(f, a) =>
    val CloV(b, fenv) = interp(f, env)
    interp(b, interp(a, env) :: fenv)
  case Num(n) => NumV(n)
  case Add(l, r) =>
    val NumV(n) = interp(l, env)
    val NumV(m) = interp(r, env)
    NumV(n + m)
}
```

The App case is the only interesting case. The others are the same as before. Since a closure lacks its parameter name and an environment does not need the name, it is enough to prepend the value of the argument in front of the list.

3: Assume that the equality of closures is defined properly.

4: At this point, we do not consider named expressions, so we omit the `Nameless` singleton object.

Typed Languages

18 Type Systems

This chapter is the first chapter about typed languages. This chapter explains the motivation of type checking and introduces a simple type system by defining TFAE, a typed variant of FAE.

18.1 Run-Time Errors

In FAE, expressions can be classified into three groups according to their behaviors. Let us see what those three groups are. Note that in most languages, expressions can be classified into three groups in the same manner. Thus, the discussion of this section can be applied to various real-world languages. Just for brevity, this section uses FAE.

The first group includes every expression that evaluates to a value. For example, $(1 + 2) - 3$ and $(\lambda x.\lambda y.x + y)\ 1\ 2$ belong to the first group because $(1 + 2) - 3$ evalutes to 0, and $(\lambda x.\lambda y.x + y)\ 1\ 2$ evaluates to 3. Expressions in this group correspond to programs that terminate without any problem. When we write a program, the program usually belongs to the first group.

The second group includes every expression that never teminates. For instance, $(\lambda x.x\ x)\ (\lambda x.x\ x)$ belongs to the second group. The expression is function application. The first $\lambda x.x\ x$ is a function, and the seceond $\lambda x.x\ x$ is an argument. To evaluate the function application, the body, $x\ x$, is evaluated under the environment that maps x to $\lambda x.x\ x$. Following the content of the environment, evaluating $x\ x$ is equivalent to evaluating $(\lambda x.x\ x)\ (\lambda x.x\ x)$, which is the original expression. Thus, we can say that the evaluation of $(\lambda x.x\ x)\ (\lambda x.x\ x)$ leads to the evaluation of the exactly same expression. The evaluation runs forever and never terminates. There are many nonterminating programs in real world. If a language supports recursive functions or loops, writing nonterminating programs becomes much easier. Some of them are created by programmers' mistakes. Wrong use of recursive functions or loops makes programs run forever, contrary to the expectation of the programmers. However, programmers sometimes intentionally write nonterminating programs. Consider operating systems, web servers, and shells. They do not finish their execution unless a user inputs a termination command. If an operating system terminates although a user has not given any commands, such a behavior should be considered as a bug. These examples clearly show the necessity of writing nonterminating programs.

The third group includes every expression that terminates but fails to produce a result. For example, $(\lambda x.x) + 1$, $1\ 0$, and $2 - x$ belong to the third group. The first example, $(\lambda x.x) + 1$ adds a function to an integer. Since such addition is impossible, the evaluation cannot proceed beyond the addition. Thus, the evaluation terminates at the middle of the computation rather than reaching the final stage and producing a result.

The second example, 1 0, applies an integer to a value. Functions can be applied to values, but intgers cannot. Such an application expression also makes the evaluation terminate abnormally. In the last example, x is a free identifier. Its value is unknown, so there is no way to subtract the value of x from 2. Expressions in this group correspond to programs that incur *run-time errors*.

Run-time errors are always unintentional. The only reason to write expressions that incur run-time errors is programmers' mistakes. Run-time errors terminate programs abnormally before the programs produce meaningful results. Programmers write programs to achieve their goals: getting particular results or performing certain tasks forever. Run-time errors hinder programmers from achieving the goals. Run-time errors are problematic not only to programmers but also to other people. In commercial software, run-time errors are unpleasant experiences for users and harm the profits and reputations of the company. Moreover, people use programs for various purposes these days, so run-time errors can cause much more serious problems. Programs control cars, airplanes, and medical devices. Improper operations of such devices may kill or hurt people. A device will operate in a weird way if the program controlling the device terminates abnormally. Programmers surely need a way to check the existence of run-time errors before they deploy programs.

18.2 Detecting Run-Time Errors

The simplest way to detect run-time errors is to run a program. This strategy is called *dynamic analysis*. The term dynamic means "with execution" (or "during exeuction"). In general, analysis of a program means to determine whether the program satisfies a certain property. In this chapter, we are interested in existence of run-time errors, so analysis means to detect run-time errors. It is straightforward to find run-time errors in a program with dynamic analysis. If execution finishes due to a run-time error, the program needs revision. Otherwise, the program may be usable without any problems.

However, dynamic analysis often fails to detect run-time errors. Execution can take a long time or run forever. Programmers want to deploy their programs; they cannot wait for the execution forever to finish. The dynamic analysis must stop at some point. It makes complete prevention of run-time errors impossible. Even though a program runs one hundred hours without any run-time errors, it can incur a run-time error after one more hour of execution. Moreover, most programs take inputs from users, and there are infinitely many possible inputs. Dynamic analysis cannot cover all the cases. Even if a program runs fine for every input being tried, the program can result in a run-time error for another input. In addition, some programs are nondeterministic. For example, multithreaded programs can produce different results among multiple runs because the execution of threads is interleaved arbitrarily. Even when run-time errors are not found during a few trials, the absence of run-time errors cannot be guaranteed. Dynamic analysis is a simple and popular way to find run-time errors but cannot ensure the nonexistence of run-time errors. To rule out run-time errors in programs, we need a better way than dynamic analysis.

Since executing programs cannot prove the absence of run-time errors, we should explore a way to detect run-time errors without executing programs. This approach is called *static analysis*. The term static means "without execution" (or "before execution"). We want to make a program that automatically checks whether a given program can cause a run-time error. More precisely, we want a program P that satisfies the following conditions:

- For any program I given as an input, P outputs OK or NOT OK in a finite time.
- If I never incurs run-time errors, $P(I) = \mathsf{OK}$.[1]
- If I can incur a run-time error, $P(I) = \mathsf{NOT\ OK}$.

1: $P(I)$ denotes the output of P when I is an input.

The first property implies that P always terminates, which is different from dynamic analysis. The second property is called *completeness*, which implies that P never detects run-time errors by mistake, i.e., there is no false positive—a *false positive* is to incorrectly say OK. If $P(I)$ is NOT OK, I must be able to incur a run-time error.[2] The third property is called *soundness*, which implies that P never misses run-time errors, i.e., there is no false negative—a *false negative* is to incorrectly say NOT OK. If $P(I)$ is OK, I must be free from run-time errors.[3] P can liberate programmers from the burden of detecting run-time errors. If P says OK, then the programmers do not need to worry about run-time errors at all. If P says NOT OK, then the program is certainly wrong, and the programmers should fix the problem.

2: It is the contrapositive of the second property.

3: It is the contrapositive of the third property.

Alas, such a program P does not exist. It has not existed so far and will not exist in the future as well. In other words, the problem of deciding whether a certain program can incur a run-time error is proven to be undecidable. The undecidability can be proved in a similar fashion to Turing's proof of the undecidability of the halting problem. We do not explain the proof because the proof is outside the scope of this book.

Fortunately, there is a tolerable solution. If we give up either completenss or soundness, we can find such a program P. The most common choice is to forgo completeness. Dynamic analysis is complete because a run-time error found during execution always indicates a real bug. Dynamic analysis does not suffer from false positives. The limitation of dynamic analysis is its unsoundness; it can miss run-time errors. It would be better to design static analysis as a complementary technique. P, which performs static analysis, should be sound at the cost of losing completeness. Now, P satisfies the following conditions:

- For any given program I as an input, P outputs OK or NOT OK in a finite time.
- If I never incurs run-time errors, both $P(I) = \mathsf{OK}$ and $P(I) = \mathsf{NOT\ OK}$ are possible.
- If I can incur a run-time error, $P(I) = \mathsf{NOT\ OK}$.

If P says OK, it still guarantees the absence of run-time errors; there is no false negative. However, if P says NOT OK, we cannot get any information. False positives are possible, so P can say NOT OK even when a given program never causes a run-time error in fact.

After giving up completness, P should satisfy two more conditions in order to be practically useful. First, the number of false positives must be modest. If not, programmers cannot get useful information from

P. For example, we can design P to output NOT OK in any case. Such P surely satisfies the above conditions because producing NOT OK is allowed when a given program never incurs run-time errors. Of course, programmers cannot get any help from such P. Therefore, the number of false positives must be modest. Second, when P says NOT OK, it must provide additional information to let programmers know why it says so. The information can help programmers decide whether NOT OK is a false positive or not.

Sadly, it is difficult to make P that satisfies the original three conditions and the new two conditions. It is still possible but extremely challenging. Therefore, people forgo the detection of all the run-time errors and try to catch a subset of them. They classify run-time errors into two categories: type errors and the others. *Type errors* are run-time errors due to use of values of wrong types. The other run-time errors are irrelevant to types. Now, the goal of P is to detect every type error. P satisfies the following conditions:

- For any given program I as an input, P outputs OK or NOT OK in a finite time.
- If I never incurs type errors, both $P(I) =$ OK and $P(I) =$ NOT OK are possible.
- If I can incur a type error, $P(I) =$ NOT OK.
- The number of false positives is modest.
- When $P(I) =$ NOT OK, P provides additional information about its decision.

Currently, there is no notion of a type. To distinguish type errors from the others, we first need to define what a type is.

18.3 Type Errors

A *type* is a set of values. We use types to categorize values according to their ability and structures. Values with the same ability and structure belong to the same type, and values with different ability or structures belong to different types. There are various values in each programming language, and each value has its own ability and structure. For instance, integers can be used for addition and subtraction, while functions can be applied to values. Thus, it is natural to classify values according to their characteristics. In TFAE, the easiest way to categorize values is to split them into numbers and functions. For example, 1, 42, 0, and -1 are numbers and belong to the type num. On the other hand, λx.x, λx.x + x, and λx.x 1 are functions and belong to the type fun. Actually, this classification is too coarse, and we will refine it later. For now, num and fun are quite enough to introduce the notion of a type.

Now, we can explain what a type error is. When a value of an unexpected type appears, a type error happens. More precisely, if a value of the fun type appears where a value of the num type is required, or vice versa, then a type error happens. Consider (λx.x) + 1. The first operand belongs to fun, and the second operand belongs to num. Addition expects both operands to be num. Since a value of fun appears where a value of num is expected, evaluation of the expression incurs a type error. As another example, consider 1 0. The first operand belongs to num. However,

function application expects the first operand to be fun. A type error happens during the evalutaion because a value of num appears where a value of fun is expected.

Sometimes, it is unclear to determine whether a certain run-time error is a type error or not. The definition of a type error can vary among people. For example, recall the expression $2 - \mathsf{x}$. The expression incurs a run-time error because x is a free identifier. One may say this error is irrelevant to types. From this perspective, such an error is just a free identifier error, not a type error. However, another may say the error is relevant to types because x, whose value is unknown, cannot belong to any type and, therefore, is not a value of num although subtraction requires both operands to be num. From this perspective, free identifiers are just a subset of type errors. There is no single correct answer; both perspectives make sense. This book follows the latter, i.e., that free identifiers are type errors, because it fits the purpose of our discussion better.

Even if we classify free identifier errors as type errors, not all run-time errors are type errors. Some run-time errors happen even when the types of values are correct, so they cannot be classified as type errors. We cannot find such examples in FAE. Type errors are all of possible run-time errors in FAE. However, in many real-world languages, which provide various features FAE excludes, we can find run-time errors irrelevant to types. One of the most famous examples is `NullPointerException` of Java. In Java, `null` is a value that belongs to any type.[4] Thus, `null` is a value of the `String` type. Java strings provide various methods, including `length`, which computes the length of the string. However, the following code incurs `NullPointerException`, which is one sort of a run-time error in Java, because computing the length of `null` is impossible:

4: In fact, Java classifies types into primitive types and reference types, and `null` is a value of any reference type.

```
String s = null;
s.length();
```

`NullPointerException` is not a type error since a value of `String` is expected in front of `.length()`, and `s`, which denotes `null`, does belong to `String`.

18.4 Type Checking

P, which detects type errors in a given program, is called a *type checker*. A type error happens when a value of an unexpected type occurs. Therefore, to find type errors, a type checker predicts the type of the result of an expression and compares the predicted type with the expected type.

For example, consider $e_1 + e_2$. To evaluate $e_1 + e_2$ without type errors, the following conditions must be satisfied:

- e_1 does not incur a type error.
- e_1 evaluates to a value of num or does not terminate.
- e_2 does not incur a type error.
- e_2 evaluates to a value of num or does not terminate.

When the conditions are true, not only the absence of type errors in $e_1 + e_2$ is guaranteed, but also we can predict the result of $e_1 + e_2$: it evaluates to a value of num or does not terminate.

Now, let us say that "the type of e is τ" when the following conditions are true:

- e does not incur a type error.
- e evaluates to a value of τ or does not terminate.

where the metavariable τ ranges over types. Then, we can restate the finding of the above paragraph: when the type of e_1 is num and the type of e_2 is num, the type of $e_1 + e_2$ is num.

This example shows what a type checker does. A type checker computes the type of an expression. When the type is successfully computed, it ensures that the expression does not incur type errors. In this case, we say that the expression is *well-typed*. Then, the type can be used to check whether an expression containing the previously checked expression can cause type errors. This process is repeated until the whole program is checked. We call this process *type checking*.

A type checker requires different strategies to predict the types of different sorts of an expression. In the above example, addition requires both subexpressions to have num as their types. However, it is clear that function application requires different types. It requires the first subexpression to have fun as its type because only functions can be applied to values. These examples show that a type checker needs a separate rule for each sort of an expression to predict the type of the expression. We call such rules *typing rules*.

There are multiple typing rules in a single language, and we call the collection of all the typing rules in a language the *type system* of the language. *Static semantics* is another name of a type system since type systems explain the behaviors of expressions by predicting their types without execution. To distinguish the semantics so far, which explains the behaviors of expressions by defining their values from execution, from static semantics, we use the term *dynamic semantics*.

The following table compares dynamic semantics and static semantics:

	Dynamic semantics	Static semantics
What it is for	Evaluation	Type checking
Which program implements it	Interpreter	Type checker
Result	Value	Type

Dynamic semantics defines how expressions are evaluated. By evaluation, expressions result in values. An interpreter is a program that takes an expression and computes its result. Static semantics defines how expressions are type-checked. By type checking, the types of expressions are computed. A type checker is a program that takes an expresion, predicts its type, and checks whether run-time errors are possible. We can consider static semantics as overapproximation of dynamic semantics. For example, dynamic semantics lets us know that $1 + 2$ results in 3, while static semantics lets us know that $1 + 2$ results in an integer without any run-time errors or does not terminate.

As mentioned before, the goal of a type checker, P, is soundness. Therefore, the most important property of type systems is *type soundness*, or simply, just soundness. If a type checker says OK for a given program, then the program must never incur type errors. In this case, we say that the program passes type checking or that the type checker accepts the

program. On the other hand, if a type checker says NOT OK for a given program, we cannot conclude anything, but the program might incur a type error. In this case, we say that the type checker rejects the program.

It is nontrivial to design a sound type system for a given language. Proving the soundness of a type system is more challenging. Proving type soundness is beyond the scope of this book. This book introduces various type systems whose type soundness has been proved by researchers already.

Since designing a type system and implementing a type checker are difficult tasks, those tasks are the jobs of language designers, not language users in most cases. Some languages come out with type systems. We call such languages *typed languages* or *statically typed languages*. The terms imply that the languages have the notion of a type whose correct use is verified statically. In such languages, only programs that pass type checking can be executed. Programs rejected by the type checker are disallowed to be exectued because their safety is not ensured. Therefore, any execution is guaranteed to be type error free. Java, Scala, and Rust are well-known statically typed languages in real world.

On the other hand, some languages do not provide type systems. We call such languages *untyped languages* or *dynamically typed languages*. The term untyped languages implies that they do not have type checking. The term dynamically typed languages implies that they have the notion of a type only at run time. Note that a type is a natural concept that exists anywhere because values can be classified according to their characteristics in any languages. However, in dynamically typed languages, types exist only during execution since there are no static type checking. In such languages, programs may incur type errors during execution. Python and JavaScript are well-known dynamically typed languages in real world.

Statically typed languages and dynamically typed languages have their own pros and cons. Statically typed languages have the following advantages:

- Errors can be detected early. Programmers can find errors before execution.
- Static type checking gives type information to compilers, and the compilers can optimize programs with the information. For these reasons, programs in statically typed languages usually outperform programs in dynamically typed languages.
- Some statically typed languages require programmers to write types explicitly on the source code. Such types on the code are called *type annotations*. Type checkers verify the correctness of the type annotations. Thus, type annotations are automatically verified comments, which never become outdated, and help programmers understand the programs easily.

On the other hand, statically typed languages have the following disadvantages:

- Statically typed languages attain type soundness by giving up completeness. Type checkers may reject programs that never incur type errors. Therefore, programmers may waste their time in making type checkers agree that given programs do not result in type errors.

- Type annotations make code unnecessarily verbose despite their usefulness. In addition, programmers spend their time on writing correct type annotations.

Due to these characteristics, statically typed languages are attractive when one writes complex programs whose error detection is difficult. In addition, programs that have to be highly trustworthy or require high performance are typical use cases of statically typed languages. Programs that need long-term maintenance also are good clients of statically typed languages.

The characteristics of dynamically typed languages are the opposite of statically typed languages. Due to the lack of static type checking, errors are discovered during execution, and programs lose some chances of optimization. However, inconvenience due to the incompleteness of type systems disappears. Dynamically typed languages liberate programmers from the burden of fighting against type checking and allow them to save their time.

Therefore, dynamically-typed languages are ideal for the early stage of development. Programmers can easily make prototypes of their programs and try various changes in the prototypes. They do not waste their time arguing with type checkers. Also, programs that are very small and used only a few times are where dynamically typed languages should be used. In such applications, the advantages of statically typed languages are worthless.

18.5 TFAE

This section defines TFAE, a statically typed version of FAE.

Syntax

The syntax of TFAE is as follows:

$$e \ ::= \ n \mid e + e \mid e - e \mid x \mid \lambda x{:}\tau.e \mid e \ e$$

The only difference from FAE is the type annotation of a lambda abstraction. $\lambda x{:}\tau.e$ is a function whose parameter is x, parameter type is τ, and body is e. The parameter type annotation is required during type checking, which will be explained soon.

Now, we need to define types. Classifying values into num and fun like so far is too imprecise. We need more fine-grained types for functions for a few reasons. First, functions require arguments to belong to specific types. Consider λx.x + x. When the function is applied to a value, the value must be a number to avoid a type error. If a function is given as an argument, the evaluation of the body incurs a type error. Each function has its own requirement. Therefore, the type of a function must describe the type of an argument expected by the function. Second, different functions return different values. Some functions returns numbers, while others return functions. To predict the type of a function application expression, the type checker must be able to predict the type of the return

value. Thus, the type of a function must describe the type of the return value as well.

Based on the above observations, we define types as follows:

$$\tau \ ::= \ \mathsf{num} \mid \tau \rightarrow \tau$$

The type num is the type of every number. A type $\tau_1 \rightarrow \tau_2$ is the type of a function that takes a value of τ_1 as an argument and returns a value of τ_2. For example, λx:num.x takes a value of num and returns the value. Its type is num $\rightarrow$ num. λx:num.λy:num.x + y takes a value of num and returns λy:num.x + y. λy:num.x + y also takes a value of num. Since both x and y are numbers, x + y also is a number, whose type is num. Therefore, the type of λy:num.x + y is num $\rightarrow$ num, and the type of λx:num.λy:num.x + y is num $\rightarrow$ (num $\rightarrow$ num). Because arrows in function types are right associative, we can write num $\rightarrow$ num $\rightarrow$ num instead.

A type is either num or $\tau_1 \rightarrow \tau_2$ for some τ_1 and τ_2. Every value belongs to at most one type. No value is an integer and a function at the same time. No function takes an integer as an argument and a function as an argument at the same time. In this chapter, every value has at most one type, and, therefore, every expression has at most one type as well. However, in some type systems, a single value or a single expression can have multiple types. Chapter 22 shows such an example.

Dynamic Semantics

The dynamic semantics of TFAE is similar to that of FAE. The only difference is type annotations in lambda abstractions. Since type annotations are used only for type checking and do not have any role at run time, they are simply ignored when closures are constructed.

Rule Fun
$\lambda x{:}\tau.e$ evaluates to $\langle \lambda x.e, \sigma \rangle$ under σ.

$$\sigma \vdash \lambda x{:}\tau.e \Rightarrow \langle \lambda x.e, \sigma \rangle \quad [\textsc{Fun}]$$

Interpreter

An interpreter of TFAE is similar to that of FAE. Since lambda abstractions have type annotations, the Fun case class needs a change.

```
sealed trait Expr
...
case class Fun(x: String, t: Type, b: Expr) extends Expr
```

Fun(x, τ, e) represents $\lambda x{:}\tau.e$.

In addition, we define types as an ADT.

```
sealed trait Type
```

```
case object NumT extends Type
case class ArrowT(p: Type, r: Type) extends Type
```

`NumT` represents num, and `ArrowT(`τ_1`, `τ_2`)` represents $\tau_1 \to \tau_2$.

The `interp` function needs only one fix in the `Fun` case.

```
case Fun(x, _, b) => CloV(x, b, env)
```

Type annotations are ignored.

Static Semantics

Now, we define the static semantics of TFAE. One naïve approach is to define the static semantics as a relation over expressions and types because static semantics defines the type of each expression. However, this approach does not work. Recall that the dynamic semantics is a relation over environments, expressions, and values. An environment stores the values of variables. Since variables exist both before and at run time, the static semantics needs information about variables. While the dynamic semantics requires the values of variables, the static semantics requires the types of variables. To fulfill this requirement, we introduce a type environment, which is a finite partial function from identifiers to types. Let T be the set of every type and $TEnv$ be the set of every type environment.

$$TEnv = Id \xrightarrow{\text{fin}} \mathrm{T}$$

$$\Gamma \in TEnv$$

The metavariable Γ ranges over type environments.

The static semantics defines a relation over type environments, expressions, and types.

$$:\subseteq TEnv \times E \times T$$

$\Gamma \vdash e : \tau$ denotes that the type of an expression e under a type environment Γ is τ. If $\emptyset \vdash e : \tau$ is true for some τ, then e is well-typed, and the type system accepts the expression. If $\emptyset \vdash e : \tau$ is false for every τ, then e is *ill-typed*, i.e. not well-typed, and the type system rejects the expression.

Let us define the typing rule for each sort of an expression.

Rule Typ-Num
The type of n is num under Γ.

$$\Gamma \vdash n : \mathsf{num} \quad [\textsc{Typ-Num}]$$

The type of a number is num.

Rule Typ-Add

If the type of e_1 is num under Γ and the type of e_2 is num under Γ,
then the type of $e_1 + e_2$ is num under Γ.

$$\frac{\Gamma \vdash e_1 : \mathsf{num} \qquad \Gamma \vdash e_2 : \mathsf{num}}{\Gamma \vdash e_1 + e_2 : \mathsf{num}} \quad [\textsc{Typ-Add}]$$

If the types of e_1 and e_2 are both num, then the type of $e_1 + e_2$ is num.

Rule Typ-Sub
If the type of e_1 is num under Γ and the type of e_2 is num under Γ,
then the type of $e_1 - e_2$ is num under Γ.

$$\frac{\Gamma \vdash e_1 : \mathsf{num} \qquad \Gamma \vdash e_2 : \mathsf{num}}{\Gamma \vdash e_1 - e_2 : \mathsf{num}} \quad [\textsc{Typ-Sub}]$$

The rule for subtraction is similar to that for addition.

Rule Typ-Id
If x is in the domain of Γ,
then the type of x is $\Gamma(x)$ under Γ.

$$\frac{x \in \mathit{Domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad [\textsc{Typ-Id}]$$

The dynamic semantics finds the value of a variable from an environment. Similarly, the static semantics finds the type of a variable from a type environment. This rule allows the type system to detect free identifier errors.

Rule Typ-Fun
If the type of e is τ_2 under $\Gamma[x : \tau_1]$,[5]
then the type of $\lambda x{:}\tau_1.e$ is $\tau_1 \rightarrow \tau_2$ under Γ.

$$\frac{\Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \rightarrow \tau_2} \quad [\textsc{Typ-Fun}]$$

The rule for a lambda abstraction needs to compute the type of a closure created by the lambda abstraction. The type of an argument is given as τ_1 by the type annotation. The rule should determine the type of the return value of the function as well. The return type equals the type of e, the function body. The value of an argument is unknown, but the type is known as τ_1. It shows why a lambda abstraction needs a parameter type annotation. It gives information to compute the type of the body. Since a closure captures the environment when it is created, evaluation of its body can use variables in the environment. Thus, computation of the type of e needs every information in Γ and that the type of x is τ_1. The computation uses $\Gamma[x : \tau_1]$. If the type of e is τ_2, the return type of the function also is τ_2. Finally, the type of the lambda abstraction becomes $\tau_1 \rightarrow \tau_2$.

5: Since $\mapsto$ looks similar to arrows in types, we use : instead of $\mapsto$ to prevent confusion.

Rule Typ-App
If the type of e_1 is $\tau_1 \to \tau_2$ under Γ and the type of e_2 is τ_1 under Γ, then the type of $e_1\ e_2$ is τ_2 under Γ.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ [\text{Typ-App}]$$

A function application expression $e_1\ e_2$ is well-typed only if e_1 is a function. Let the type of e_1 be $\tau_1 \to \tau_2$. The type of the argument, e_2 must be τ_1. The type of the return value is τ_2, so the type of $e_1\ e_2$ is τ_2.

The following proof tree proves that the type of $(\lambda \mathsf{x}{:}\mathsf{num}.\lambda \mathsf{y}{:}\mathsf{num}.\mathsf{x}+\mathsf{y})\ 1\ 2$ is num:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}(\Gamma_2)}{\Gamma_2 \vdash \mathsf{x} : \mathsf{num}} \quad \dfrac{\mathsf{y} \in \mathit{Domain}(\Gamma_2)}{\Gamma_2 \vdash \mathsf{y} : \mathsf{num}}}{\Gamma_2 \vdash \mathsf{x} + \mathsf{y} : \mathsf{num}}}{\Gamma_1 \vdash \lambda \mathsf{y}{:}\mathsf{num}.\mathsf{x} + \mathsf{y} : \mathsf{num} \to \mathsf{num}}}{\emptyset \vdash e : \mathsf{num} \to \mathsf{num} \to \mathsf{num}} \quad \emptyset \vdash 1 : \mathsf{num}}{\emptyset \vdash e\ 1 : \mathsf{num} \to \mathsf{num}} \quad \emptyset \vdash 2 : \mathsf{num}}{\emptyset \vdash e\ 1\ 2 : \mathsf{num}}$$

where

$e = \lambda \mathsf{x}{:}\mathsf{num}.\lambda \mathsf{y}{:}\mathsf{num}.\mathsf{x} + \mathsf{y}$
$\Gamma_1 = [\mathsf{x} : \mathsf{num}]$
$\Gamma_2 = [\mathsf{x} : \mathsf{num}, \mathsf{y} : \mathsf{num}]$

We call a proof tree that proves the type of an expression a *type derivation*.

This type system is sound; it rejects every expression producing a type error. For example, consider $(\lambda \mathsf{x}{:}\mathsf{num} \to \mathsf{num}.\mathsf{x}\ 1)\ 1$. Evaluation of the expression results in evaluation of 1 1, which causes a type error. Since the type of x 1 is num, the type of the function is $(\mathsf{num} \to \mathsf{num}) \to \mathsf{num}$. The function takes an argument of type $\mathsf{num} \to \mathsf{num}$. However, 1, the argument, has the type num, which differs from $\mathsf{num} \to \mathsf{num}$. Therefore, the type checker rejects the expression, which is a correct decision.

Any sound type system is incomplete. Therefore, this type system is incomplete. The type system can reject a type-error-free expression. Various such expressions exist. Consider $(\lambda \mathsf{x}{:}\mathsf{num}.\mathsf{x})\ (\lambda \mathsf{x}{:}\mathsf{num}.\mathsf{x})$. The expression evaluates to $\langle \lambda \mathsf{x}.\mathsf{x}, \emptyset \rangle$ without any type error. However, the type system rejects the expression. $\lambda \mathsf{x}{:}\mathsf{num}.\mathsf{x}$ takes an argument of the type num. However, $\lambda \mathsf{x}{:}\mathsf{num}.\mathsf{x}$, the argument, has the type $\mathsf{num} \to \mathsf{num}$, which differs from num. As a result, the type system rejects the expression even though it evaluates to a value without any type error.

Type Checker

To implement a type checker of TFAE, we first define the TEnv type, which is the type of a type environment.

```
type TEnv = Map[String, Type]
```

`TEnv` is a map from strings to TFAE types.

The following `mustSame` function compares given two types:

```
def mustSame(t1: Type, t2: Type): Unit =
  if (t1 != t2)
    throw new Exception
```

If the types are different, it throws an exception.

The following `typeCheck` function type-checks a given expression under a given type environment.

```
def typeCheck(e: Expr, env: TEnv): Type = e match {
  case Num(n) => NumT
  case Add(l, r) =>
    mustSame(typeCheck(l, env), NumT)
    mustSame(typeCheck(r, env), NumT)
    NumT
  case Sub(l, r) =>
    mustSame(typeCheck(l, env), NumT)
    mustSame(typeCheck(r, env), NumT)
    NumT
  case Id(x) => env(x)
  case Fun(x, t, b) =>
    ArrowT(t, typeCheck(b, env + (x -> t)))
  case App(f, a) =>
    val ArrowT(t1, t2) = typeCheck(f, env)
    mustSame(t1, typeCheck(a, env))
    t2
}
```

If type checking succeeds, the function returns the type of the expression. Otherwise, it throws an exception. Therefore, if the function throws an exception for a given expression, the expression is ill-typed. If the function terminates without throwing an exception, the expression is well-typed.

Each case of the pattern matching coincides with the corresponding typing rule. In the `Num` case, the type is `NumT`. In the `Add` and `Sub` cases, the subexpressions of the expression must have the type `NumT`. The type of the expression also is `NumT`. The `Fun` case checks the type of the function body under the extended type environment. The type of the function is a function type. The parameter type is the same as the type annotation, and the return type is the type of the body. The `App` case checks the types of the function and the argument positions. The parameter type of the function position must equal the type of the argument position. The type of the application expression is the return type of the function position.

18.6 Extending Type Systems

Type system designers extend type systems to enhance their usability. Type systems can be extended in various ways. Adding new sorts of an expression or a type is a typical way. On the other hand, it is possible to refine typing rules without changing the syntax of a language. Chapter 22 illustrates an example of refining typing rules by adding subtyping.

There are multiple reasons to extend type systems. First, people extend type systems to make them less incomplete. Incompleteness is an inherent limitation of type systems. Type systems reject some expressions that do not incur type errors. False positives cannot be eliminated completely. However, we can extend type systems to reduce the number of false positives. By doing so, programmers can suffer less from the misfortune that type-error-free programs are rejected. Reducing false positives is the most common reason of extending type systems. The subsequent four chapters show famous and practically useful type system extensions of this kind.

Second, type systems are extended just for the convenience of programmers. Extensions of this kind do not reduce the number of false positives. However, by adding useful language constructs, programmers become easily able to express their high-level ideas in source code. It is the same as extensions of dynamically typed languages described by previous chapters. For example, it is possible to define recursive functions in FAE. However, defining recursive functions in FAE is difficult and complex. To resolve the problem, we define RFAE by extending FAE with primitive support for recursive functions. Recursive functions can be defined much easily in RFAE than FAE. Statically typed languages can be improved in similar ways.

Third, more run-time errors can be considered as type errors by extending type systems. In most real-world languages, some run-time errors are not type errors. For example, recall `NullPointerException` of Java. Since Java does not classify `NullPointerException` as a type error, the Java type checker does not detect `NullPointerException`, and every Java program suffers from the possibility of `NullPointerException`. `NullPointerException` is one of the most commonly occuring run-time errors in Java. Kotlin introduces the notion of an explicit null type by extending the type system of Java. In Kotlin, `null` is not a value of `String`, and `NullPointerException` is considered as a type error. If a Kotlin program passes type checking, it is free of `NullPointerException`. Extending type systems to detect more run-time errors is valuable but can increase false positives. Thus, type system designers always consider the tradeoff between enlarging the set of type errors and reducing false positives.

Now, let us consider two simple extensions of TFAE: local variable definitions and pairs.

Local Variable Definitions

Local variable definitions (val expressions) are the second kind of an extension. Even without local variable definitions, programmers can write

the equivalent code with functions. However, local variable definitions help them write code concisely.

The syntax and dynamic semantics of local variable definitions follow VAE. The static semantics is as follows:

Rule Typ-Val
If the type of e_1 is τ_1 under Γ and the type of e_2 is τ_2 under $\Gamma[x : \tau_1]$, then the type of val x=e_1 in e_2 is τ_2 under Γ.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{val}\ x{=}e_1\ \mathsf{in}\ e_2 : \tau_2} \quad [\text{Typ-Val}]$$

Note that local variable definitions do not require type annotations, while lambda abstractions do. Therefore, local variable definitions are more convenient then lambda abstractions for binding.

Pairs

Pairs are the first kind of an extension. In FAE, we can desugar pairs to functions:

- (e_1, e_2), which creates a new pair, is desugared to $\lambda \mathtt{f}.\mathtt{f}\ e_1\ e_2$.[6]
- $e.1$, which acquires the first element of a pair, is desugared to $e\ \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}$.
- $e.2$, which acquires the second element of a pair, is desugared to $e\ \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}$.

6: Strictly speaking, the correct desugaring is $(\lambda \mathtt{x}.\lambda \mathtt{y}.\lambda \mathtt{f}.\mathtt{f}\ \mathtt{x}\ \mathtt{y})\ e_1\ e_2$ in eager languages like FAE, but we use the simpler one here.

However, such expressions are ill-typed in TFAE. When the type of e_1 is num and the type of e_2 is num $\to$ num, $\lambda \mathtt{f}.\mathtt{f}\ e_1\ e_2$ is a function that returns num in some cases and num $\to$ num in some other cases. There is no way to represent the type of such a function. Thus, programs using pairs cannot be written in TFAE.

To overcome the limitation, we extend TFAE to support pairs. The syntax and dynamic semantics of pairs follow Exercise 7 of Chapter 9. We add pair types as follows:

$$\tau ::= \cdots \mid \tau \times \tau$$

A type $\tau_1 \times \tau_2$ is the type of (v_1, v_2) if the type of v_1 is τ_1 and the type of v_2 is τ_2. The following rules define the static semantics:

Rule Typ-Pair
If the type of e_1 is τ_1 under Γ and the type of e_2 is τ_2 under Γ, then the type of (e_1, e_2) is $\tau_1 \times \tau_2$ under Γ.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad [\text{Typ-Pair}]$$

Rule Typ-Fst
If the type of e is $\tau_1 \times \tau_2$ under Γ,

then the type of $e.1$ is τ_1 under Γ.

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.1 : \tau_1} \quad [\textsc{Typ-Fst}]$$

Rule Typ-Snd
If the type of e is $\tau_1 \times \tau_2$ under Γ,
then the type of $e.2$ is τ_2 under Γ.

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.2 : \tau_2} \quad [\textsc{Typ-Snd}]$$

18.7 Exercises

1. This exercise extends TFAE with lists.

   $$e ::= \cdots \mid \mathsf{nil}[\tau] \mid \mathsf{cons}\ e\ e \mid \mathsf{head}\ e \mid \mathsf{tail}\ e$$
   $$\tau ::= \cdots \mid \mathsf{list}\ \tau$$

   Write the typing rules of the added expressions.
2. This exercise extends TFAE with boxes.

   $$e ::= \cdots \mid \mathsf{box}\ e \mid !e \mid e{:=}e \mid e; e$$
   $$\tau ::= \cdots \mid \mathsf{box}\ \tau$$

   The dynamic semantics of boxes is the same as BFAE.

   a) Write the typing rules of the added expressions. Assignments should not change the types of the values at given locations.
   b) Draw the type derivation of the following expression:

   ```
   val x=box 3 in
   val y=!x + 7 in
   x:=8;
   y+!x
   ```

3. This exercise extends TFAE with mutable variables.

   $$e ::= \cdots \mid x{:=}e$$

   The dynamic semantic of mutable variables is the same as MFAE.

   a) Write the typing rule of the added expression.

   Now, we extend the language again with pointers.

   $$e ::= \cdots \mid *e \mid \&x \mid *e{:=}e$$
   $$\tau ::= \cdots \mid \tau*$$

   The dynamic semantic of pointers is the same as Exercise 1 of Chapter 12. A type $\tau*$ denotes the address type of a type τ. For example, for a given address a, if the value at a is a number, then the type of a is num$*$.

   b) Write the typing rules of the added expressions.

19 Typing Recursive Functions

In FAE, recursive functions can be considered as syntactic sugar. For example, a function that computes the sum from one to a given integer can be implemented as the following:

$Z\ (\lambda \mathsf{f}.\lambda \mathsf{v}.\mathsf{if0}\ \mathsf{v}\ 0\ (\mathsf{v} + \mathsf{f}\ (\mathsf{v} - 1)))$

where

$Z = \lambda \mathsf{f}.(\lambda \mathsf{x}.\mathsf{f}\ (\lambda \mathsf{v}.\mathsf{x}\ \mathsf{x}\ \mathsf{v}))\ (\lambda \mathsf{x}.\mathsf{f}\ (\lambda \mathsf{v}.\mathsf{x}\ \mathsf{x}\ \mathsf{v}))$

Thus, RFAE has the exactly same expressive power as FAE.

However, we cannot implement recursive functions in TFAE. Why cannot we use the same approach as FAE? See the body of Z. We can find x x. Unfortunately, such an expression is ill-typed in TFAE. Let us try to find the type of x. It is sure that the type cannot be num because x is applied to a value. Therefore, the type must be $\tau_1 \to \tau_2$ for some τ_1 and τ_2. Since a function of $\tau_1 \to \tau_2$ is applied to x, the type of x must be τ_1. It implies that the type of x is $\tau_1 \to \tau_2$ and τ_1 at the same time. Since a value has at most one type in TFAE, $\tau_1 \to \tau_2$ must be the same as τ_1. However, it is impossible. In TFAE, such a type does not exist. A type cannot be the same as a part of itself. Since the fixed point combinator is ill-typed, we cannot desugar recursive functions to nonrecursive functions in TFAE.

More interestingly, not only recursive functions, but also any nonterminating programs cannot be written in TFAE. In other words, every well-typed TFAE expression evaluates to a value in a finite time. This is called the *normalization* property of TFAE. The normalization property of TFAE was proved by Tait in 1967 [Tai67].[1]

This chpater defines TRFAE, which extends TFAE with recursive functions. By adding recursive functions to the language, programmers become able to implement recursive functions and nonterminating programs, which are impossible in TFAE. Thus, while the extension from FAE to RFAE does not increase the expressivity, the extension from TFAE to TRFAE dose.

1: Actually, the Tait's proof is about the lazy version of TFAE, but the same technique can be applied to TFAE, which is eager, as Pierce discussed [Pie02].

19.1 Syntax

The following is the syntax of TRFAE:

$$e\ ::=\ \cdots \mid \mathsf{if0}\ e\ e\ e \mid \mathsf{def}\ x(x{:}\tau){:}\tau{=}e\ \mathsf{in}\ e$$

def $x_1(x_2{:}\tau_1){:}\tau_2{=}e_1$ in e_2 defines a recursive function. It is similar to a recursive function in RFAE but additionally has type annotations τ_1 and τ_2. τ_1 denotes the parameter type of the function, and τ_2 denotes the return type of the function. They are used for type checking, just like type annotations in TFAE.

19.2 Dynamic Semantics

The dynamic semantics of TRFAE is similar to that of RFAE, but type annotations are ignored during closure construction.

Rule Rec
If e_2 evaluates to v under σ', where $\sigma' = \sigma[x_1 \mapsto \langle \lambda x_2.e_1, \sigma' \rangle]$,
then def $x_1(x_2{:}\tau_1){:}\tau_2{=}e_1$ in e_2 evaluates to v under σ.

$$\frac{\sigma' = \sigma[x_1 \mapsto \langle \lambda x_2.e_1, \sigma' \rangle] \qquad \sigma' \vdash e_2 \Rightarrow v}{\sigma \vdash \textsf{def } x_1(x_2{:}\tau_1){:}\tau_2{=}e_1 \textsf{ in } e_2 \Rightarrow v} \quad [\textsc{Rec}]$$

19.3 Interpreter

The following Scala code implements the syntax:

```
sealed trait Expr
...
case class If0(c: Expr, t: Expr, f: Expr) extends Expr
case class Rec(
  f: String, x: String, p: Type, r: Type, b: Expr, e: Expr
) extends Expr
```

`If0(`e_1`, `e_2`, `e_3`)` represents if0 e_1 e_2 e_3, and `Rec(`x_1`, `x_2`, `τ_1`, `τ_2`, `e_1`, `e_2`)` represents def $x_1(x_2{:}\tau_1){:}\tau_2{=}e_1$ in e_2.

The `interp` function is similar to that of RFAE, but type annotations are ignored during closure construction.

```
def interp(e: Expr, env: Env): Value = e match {
  ...
  case Rec(f, x, _, _, b, e) =>
    val cloV = CloV(x, b, env)
    val nenv = env + (f -> cloV)
    cloV.e = nenv
    interp(e, nenv)
}
```

19.4 Static Semantics

We need to define typing rules for conditional expressions and recursive functions. Let us consider conditional expressions first.

Rule Typ-If0
If
  the type of e_1 is num under Γ,
  the type of e_2 is τ under Γ, and
  the type of e_3 is τ under Γ,
then
  the type of if0 e_1 e_2 e_3 is τ under Γ

$$\frac{\Gamma \vdash e_1 : \mathsf{num} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if0}\ e_1\ e_2\ e_3 : \tau} \quad [\textsc{Typ-If0}]$$

The condition of a conditional expression must be a value of num. The rule cannot determine which branch will be evaluated at run time. Since every expression has at most one type, e_2 and e_3 must have the same type, τ. The type of the whole expression also is τ.

Actually, the condition can have any type. If the result of the condition is 0, then the true branch is evaluated. If the result is a nonzero integer or a closure, then the false branch is evaluated. A value of any type can be safely used as the condition. Therefore, the type system may use the following rule instead of Rule Typ-If0:

Rule Typ-If0′
If
the type of e_1 is τ' under Γ,
the type of e_2 is τ under Γ, and
the type of e_3 is τ under Γ,
then
the type of if0 e_1 e_2 e_3 is τ under Γ

$$\frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if0}\ e_1\ e_2\ e_3 : \tau} \quad [\textsc{Typ-If0}']$$

Both rules make the type system sound, but they are different from each other. Rule Typ-If0 rejects more expressions than Rule Typ-If0′ because the former allows only integers to be conditions, while the latter allows functions as well. Therefore, from the perspective of reducing false positives, Rule Typ-If0′ is better than Rule Typ-If0. However, if the type of the condition is a function type, it is highly likely to be a mistake of the programmer. When the condition evaluates to a function, the conditional expression always evaluates its false branch. The use of a conditional expression is totally meaningless. Thus, from the perspective of detecting programmers' mistakes, Rule Typ-If0 is better than Rule Typ-If0′.

Now, we define the typing rule of a recursive function.

Rule Typ-Rec
If
the type of e_1 is τ_2 under $\Gamma[x_1 : \tau_1 \to \tau_2, x_2 : \tau_1]$ and
the type of e_2 is τ under $\Gamma[x_1 : \tau_1 \to \tau_2]$,
then
the type of def $x_1(x_2{:}\tau_1){:}\tau_2{=}e_1$ in e_2 is τ under Γ.

$$\frac{\Gamma[x_1 : \tau_1 \to \tau_2, x_2 : \tau_1] \vdash e_1 : \tau_2 \qquad \Gamma[x_1 : \tau_1 \to \tau_2] \vdash e_2 : \tau}{\Gamma \vdash \mathsf{def}\ x_1(x_2{:}\tau_1){:}\tau_2{=}e_1\ \mathsf{in}\ e_2 : \tau} \quad [\textsc{Typ-Rec}]$$

The principle of the rule is the same as the typing rule of a lambda abstraction: given type annotations are used during the type checking of the function body. Since the function itself can be used in the body, the type checking of the body requires the type of the function. This is the

reason that the expression needs the return type annotation in addition to the parameter type annotation. To type-check the body, the return type must be known. For the type checking of the body, the type environment is extended with the type of the function, $\tau_1 \rightarrow \tau_2$, and the type of the parameter, τ_1. Since the type of the body is the return type, it must be τ_2. After the type checking of the body, e_2 is type-checked. For e_2, only the type of the function is required; the parameter can be used only in the body.

The following proof trees prove that the type of def f(x:num):num=if0 x 0 (x+ f (x − 1)) in f 3 is num:

$$\dfrac{\dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}(\Gamma_1)}{\Gamma_1 \vdash \mathsf{x} : \mathsf{num}} \quad \dfrac{\mathsf{f} \in \mathit{Domain}(\Gamma_1)}{\Gamma_1 \vdash \mathsf{f} : \mathsf{num} \rightarrow \mathsf{num}} \quad \dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}(\Gamma_1)}{\Gamma_1 \vdash \mathsf{x} : \mathsf{num}} \quad \Gamma_1 \vdash 1 : \mathsf{num}}{\Gamma_1 \vdash \mathsf{x} - 1 : \mathsf{num}}}{\Gamma_1 \vdash \mathsf{f}\ (\mathsf{x} - 1) : \mathsf{num}}}{\Gamma_1 \vdash \mathsf{x} + (\mathsf{f}\ (\mathsf{x} - 1)) : \mathsf{num}}$$

$$\dfrac{\dfrac{\mathsf{x} \in \mathit{Domain}(\Gamma_1)}{\Gamma_1 \vdash \mathsf{x} : \mathsf{num}} \quad \Gamma_1 \vdash 0 : \mathsf{num} \quad \Gamma_1 \vdash \mathsf{x} + (\mathsf{f}\ (\mathsf{x} - 1)) : \mathsf{num}}{\Gamma_1 \vdash \mathsf{if0\ x\ 0\ (x + f\ (x - 1))} : \mathsf{num}}$$

$$\dfrac{\Gamma_1 \vdash \mathsf{if0\ x\ 0\ (x + f\ (x - 1))} : \mathsf{num} \quad \dfrac{\dfrac{\mathsf{f} \in \mathit{Domain}(\Gamma_2)}{\Gamma_2 \vdash \mathsf{f} : \mathsf{num} \rightarrow \mathsf{num}} \quad \Gamma_2 \vdash 3 : \mathsf{num}}{\Gamma_2 \vdash \mathsf{f}\ 3 : \mathsf{num}}}{\emptyset \vdash \mathsf{def\ f(x:num):num=if0\ x\ 0\ (x + f\ (x - 1))\ in\ f\ 3} : \mathsf{num}}$$

where $\Gamma_1 = [\mathsf{f} : \mathsf{num} \rightarrow \mathsf{num}, \mathsf{x} : \mathsf{num}]$ and $\Gamma_2 = [\mathsf{f} : \mathsf{num} \rightarrow \mathsf{num}]$.

19.5 Type Checker

To implement a type checker, we need to add the `If0` and `Rec` cases to the `typeCheck` function for TFAE.

```
case If0(c, t, f) =>
  mustSame(typeCheck(c, env), NumT)
  val tt = typeCheck(t, env)
  val tf = typeCheck(f, env)
  mustSame(tt, tf)
  tt
```

The condition of an expression must belong to num. The type of `c` is computed with `typeCheck` and compared to `NumT` with `mustSame`. The types of the branches must be the same. The `typeCheck` function computes the types of `t` and `f`, and the `mustSame` function compares them. If they are the same, then the type is the type of the whole expression.

```
case Rec(f, x, p, r, b, e) =>
  val t = ArrowT(p, r)
  val nenv = env + (f -> t)
  mustSame(typeCheck(b, nenv + (x -> p)), r)
  typeCheck(e, nenv)
```

The parameter type is `p`, and the return type is `r`. Thus, the type of `f` is the function type from `p` to `r`. The type of `x` is `p`. To type-check `b`, the type environment must have the types of `f` and `x`. The type of `b` must equal `r`. The `mustSame` function compares the types. The function can be used not only in `b`, which is the body of the function, but also in `e`. On the other hand, the parameter `x` cannot be used in `e`. Therefore, it is enough to add only the type of `f` to the type environment used to type-check `e`. The type of the whole expression is equal to the type of `e`.

19.6 Exercises

1. Write a TRFAE expression e such that only one of e and λx:num.(e x) terminates, while both e and λx:num.(e x) are well-typed.
2. Consider the following language:

$$
\begin{array}{lllllll}
e ::= & n & c ::= & \mathsf{skip} & b ::= & \mathsf{true} \\
\mid & b & \mid & x\mathsf{:=}e & \mid & \mathsf{false} \\
\mid & x & \mid & \mathsf{if}\ e\ c\ c & v ::= & n \\
\mid & e + e & \mid & \mathsf{while}\ e\ c & \mid & b \\
\mid & e < e & \mid & c; c & \tau ::= & \mathsf{num} \\
 & & & & \mid & \mathsf{bool}
\end{array}
$$

   The metavariable c ranges over commands.
   Under a given environment σ, evaluation of an expression yields a value and does not change σ. The following is the operational semantics of the expressions:

$$\sigma \vdash n \Rightarrow n \qquad \sigma \vdash b \Rightarrow b \qquad \frac{x \in \mathit{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2}$$

   a) Write the typing rules of the form $\boxed{\Gamma \vdash e : \tau}$.

   Evaluation of a command produces a new environment. The following is the operational semantics of the commands:

$$\sigma \vdash \mathsf{skip} \Rightarrow \sigma \qquad \frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash x\mathsf{:=}e \Rightarrow \sigma[x \mapsto v]}$$

$$\frac{\sigma \vdash e \Rightarrow \mathsf{true} \qquad \sigma \vdash c_1 \Rightarrow \sigma_1}{\sigma \vdash \mathsf{if}\ e\ c_1\ c_2 \Rightarrow \sigma_1} \qquad \frac{\sigma \vdash e \Rightarrow \mathsf{false} \qquad \sigma \vdash c_2 \Rightarrow \sigma_1}{\sigma \vdash \mathsf{if}\ e\ c_1\ c_2 \Rightarrow \sigma_1}$$

$$\frac{\sigma \vdash e \Rightarrow \mathsf{true} \qquad \sigma \vdash c \Rightarrow \sigma_1 \qquad \sigma_1 \vdash \mathsf{while}\ e\ c \Rightarrow \sigma_2}{\sigma \vdash \mathsf{while}\ e\ c \Rightarrow \sigma_2}$$

$$\frac{\sigma \vdash e \Rightarrow \mathsf{false}}{\sigma \vdash \mathsf{while}\ e\ c \Rightarrow \sigma} \qquad \frac{\sigma \vdash c_1 \Rightarrow \sigma_1 \qquad \sigma_1 \vdash c_2 \Rightarrow \sigma_2}{\sigma \vdash c_1; c_2 \Rightarrow \sigma_2}$$

   b) Write the typing rules of the form $\boxed{\Gamma \vdash c : \Gamma}$. The following command must be well-typed: `x := 1; x := 2`. However, the following command must be ill-typed: `x := 1; x := true`.

20 Algebraic Data Types

Algebraic Data Types (ADTs) are ubiquitous in functional programming. Chapter 5 explains the concept of an ADT and how programmers can use ADTs in Scala. ADTs are useful when a single type includes values of different structures. Such types are common in computer science. For example, lists and options, which almost every programming language provides, are typically implemented as ADTs. In addition, ASTs, which all the compilers, interpreters, and static analyzers need, can be easily implemented as ADTs as well.

Nonrecursive ADTs can be considered as syntactic sugar in TFAE.[1] Let us see how they can be desugared with an example. Consider the following Scala code:

```
sealed trait Fruit
case class Apple(radius: Int) extends Fruit
case class Banana(radius: Int, height: Int) extends Fruit
```

A fruit is either an apple or a banana. In this example, we are interested in the sizes of fruits. An apple is approximated as a sphere and, therefore, parametrized by its radius. A banana is approximated as a cylinder and, therefore, parametrized by its radius and height.

We can easily create values that represent apples and bananas like below.

```
val apple = Apple(5)
val banana = Banana(2, 6)
```

In the above code, `apple` represents an apple whose radius is `5`, and `banana` represents a banana whose radius is `2` and height is `6`.

In TFAE, we can represent a fruit as a value of $\mathsf{num} \times (\mathsf{num} \times (\mathsf{num} \times \mathsf{num}))$. Thus, a fruit value is a pair. The first value of the pair indicates which fruit it is. If the value is 0, it is an apple. Otherwise, it is a banana. The second value of the pair is another pair, which represents the size of the fruit. If the fruit is an apple, only the first value of the second pair is meaningful. The value denotes the radius of the apple. Therefore, the following value represents an apple whose radius is 5:

$(0, (5, (0, 0)))$

Note that $(0, 0)$ can be replaced with any pair of integers. On the other hand, if the fruit is a banana, only the second value, which is a pair again, of the pair is meaningful. The pair consists of the radius and height of the banana. The following value represents a banana whose radius is 2 and height is 6:

$(1, (0, (2, 6)))$

1: Here, TFAE means the extended version defined in Section 18.6.

Note that 1 can be replaced with any nonzero integer, and 0 can be replaced with any integer.

It is tedious and error-prone to make fruit values like the above, while Scala provides a simple way to construct fruit values. In TFAE, we can define functions to mimic constructors in Scala.

val `Apple`=λx:num.$(0, (x, (0, 0)))$ in
val `Banana`=λx:(num $\times$ num).$(1, (0, x))$ in
...

`Apple` is a function that takes an integer as an argument and returns an apple whose radius is the given integer. Similarly, `Banana` is a function that takes a pair of integers as an argument and returns a banana whose size is represented by the given pair. We can now easily create fruit values with `Apple` and `Banana`.

val `apple`=`Apple` 5 in
val `banana`=`Banana` $(6, 2)$ in
...

In Scala, a typical way to use a value of an ADT is pattern matching. For instance, consider a function that computes the radius of a given fruit. The function can be implemented like below.

```
def radius(f: Fruit): Int = f match {
  case Apple(r) => r
  case Banana(r, _) => r
}
```

TFAE does not have pattern matching, but we can exploit the fact that the first value of a given pair indicates which fruit it is. We use a conditional expression to perform a certain operation when the fruit is an apple, i.e. the first value is 0, and another opertaion when the fruit is a banana, i.e. the first value is nonzero. The following expression defines the `radius` function:

val `radius`=λx:(num $\times$ (num $\times$ (num $\times$ num))).if0 x.1 x.2.1 x.2.2.1 in ...

This example shows that we can desugar ADTs and pattern matching to pairs, functions, and conditional expressions in TFAE. The ADT of the example has only two variants, which have one or two parameters. ADTs can have any number of variants, and vairants can have any number of parameters. The same strategy can be used to desugar ADTs with more variants and variants with more parameters.

Although nonrecursive ADTs can be desugared in TFAE, there are a few flaws. First, desugared programs have unnecessary values. Even when we make an apple, we need $(0, 0)$, which is a pair for the size of a banana. Similarly, a banana value requires the size of an apple. They add unessential complexity and computation to the code. Second, a single type may represent conceptually different types when a single program uses multiple ADTs. In practice, it is common to use multiple ADTs in a single program. Recall that the type of a fruit is num$\times$(num$\times$(num$\times$num)). The same type may represent other types as well. For example, the type of an electronic product can also be num $\times$ (num $\times$ (num $\times$ num)). In this case, the type system allows a function intended to take a fruit to take an electronic product as an argument. It does not incur any type errors at

run time but can cause undesirable behaviors.

These flaws can be resolved by adding primitive support for ADTs to the language. After adding ADTs, programs do not require unnecessary values to construct values of ADTs. In addition, each ADT can be defined as a separate type, so conceptually different types can be correctly distinguished even when they share the same structure.

The most critical limitation of the current desugaring strategy is the missing support for recursive ADTs. Consider the following Scala code:

```
sealed trait List
case object Nil extends List
case class Cons(h: Int, t: List) extends List
```

which implements an integer list type. A list is one of the most famous recursive types. Look at the definition of `Cons`. `Cons` is one variant of `List`, so it defines `List`. At the same time, the definition uses `List` as the type of the second parameter. Thus, `List` is a recursively defined type, whose definition depends on itself.

Can we desugar the definition of a list in TFAE? For desugaring, the first thing to do is to determine the type of a list. Let the type of a list be τ. Then, τ equals (num, τ') for some τ'. The first element is an integer that indicates which variant the value denotes. When the integer is 0, the value is `Nil`; otherwise, the value is `Cons`. When the value is `Nil`, no other data is required since `Nil` does not have any parameters. Thus, the second element of a type τ' is for `Cons`. Since `Cons` has two parameters, an integer and a list, τ' equals (num, τ). Then, we obtain the equation $\tau = (\mathsf{num}, (\mathsf{num}, \tau))$. However, as discussed in the previous chapter, no type in TFAE can be the same as a part of itself. Therefore, there is no such τ. We can conclude that we cannot desugar lists in TFAE. In general, recursive ADTs cannot be expressed in TFAE.

This chapter defines TVFAE by extending TFAE [2] with ADTs, each of which can be either nonrecursive or recursive. It allows programmers to represent ADTs efficiently and concisely. In addition, many interesting recursive data types become able to be used in programs.

2: For the rest of the chapter, local variable definitions and types are not parts of TFAE. However, we may keep using them in examples.

20.1 Syntax

First, we introduce type identifiers, which are the names of types defined by programmers. For example, `Fruit` of the previous example is a type identifier. Let *TId* be the set of every type identifier.

$$t \in \mathit{TId}$$

The metavariable t ranges over type identifiers. Since *TId* includes only the names of user-defined types, num is not a member of *TId*.

The names of user-defined types can be used as types. For example, `Fruit`, which is a type name, is used as a type in `def radius(f: Fruit): Int = ...`. Therefore, we extend the syntax of types as follows:

$$\tau ::= \cdots \mid t$$

where t is a type that includes every value of an ADT whose name is t.

Now, we define the syntax of expressions:

$$e ::= \cdots \mid \mathsf{type}\ t = x@\tau + x@\tau\ \mathsf{in}\ e \mid e\ \mathsf{match}\ x(x) \to e, x(x) \to e$$

- $\mathsf{type}\ t = x_1@\tau_1 + x_2@\tau_2\ \mathsf{in}\ e$ is an expression that defines a new type. The name of the type is t, and the type has two variants. The name of the first variant is x_1, and it has a single parameter whose type is τ_1. Similarly, the name of the second variant is x_2, and it has a single parameter whose type is τ_2. The names of the variants serve as constructors in e. The type name t can be used in e as a type. In addition, since types can be recursively defined, t can appear also in τ_1 and τ_2.
- $e\ \mathsf{match}\ x_1(x_3) \to e_1, x_2(x_4) \to e_2$ is a pattern matching expression. e is the target of the pattern matching. x_1 is the name of the variant handled by the first case, and x_2 is the name of the variant handled by the second case. In the first case, x_3 denotes the value held by the match target, and e_1 determines the result. Similarly, in the second case, x_4 denotes the value held by the match target, and e_2 determines the result.

Note that each type can have only two variants, and each variant can have only one parameter. This restriction can be easily removed. For brevity, this chapter keeps the restriction.

Let us see some example expressions. The following expression defines the `Fruit` type and the `radius` function:

```
type Fruit = Apple@num + Banana@(num × num) in
val radius=λx:Fruit.x match Apple(y) → y, Banana(y) → y.1 in
...
```

The following expression defines the `List` type:

```
type List = Nil@num + Cons@(num × List) in ...
```

Note that `Nil` has one parameter since every variant of TVFAE must have a parameter. `Nil` can have any value because the value is not used at all anyway.

Recursive data types are typically used with recursive functions. If we add recursive functions of TRFAE to the language, we can implement the following `sum` function, which calculates the sum of every integer in a given list:

```
def sum(x:List):num=x match Nil(y) → 0, Cons(y) → y.1+(sum y.2) in ...
```

20.2 Dynamic Semantics

We should introduce two new sorts of a value: a variant value and a constructor.

$$v ::= \cdots \mid x(v) \mid \langle x \rangle$$

- $x(v)$ is a value of a variant named x. It contains one value v. A variant value can be the target of pattern matching.
- $\langle x \rangle$ is a constructor of a variant named x. A constructor can be considered as a special kind of a function because it can be applied to a value. When a constructor is applied to a value, a variant value is constructed.

For example, `Apple`(5) is an apple value whose radius is 5, and `Banana`$((2, 6))$ is a banana value whose radius is 2 and height is 6. Both $\langle$`Apple`$\rangle$ and $\langle$`Banana`$\rangle$ are constructors. When $\langle$`Apple`$\rangle$ is applied to 5, the result is `Apple`(5), and when $\langle$`Banana`$\rangle$ is applied to $(2, 6)$, the result is `Banana`$((2, 6))$.

Now, let us define the dynamic semantics of the added expressions. First, consider an expression that defines a new type.

Rule TypeDef
If e evaluates to v under $\sigma[x_1 \mapsto \langle x_1 \rangle, x_2 \mapsto \langle x_2 \rangle]$,
then $\mathsf{type}\ t = x_1@\tau_1 + x_2@\tau_2\ \mathsf{in}\ e$ evaluates to v under σ.

$$\frac{\sigma[x_1 \mapsto \langle x_1 \rangle, x_2 \mapsto \langle x_2 \rangle] \vdash e \Rightarrow v}{\sigma \vdash \mathsf{type}\ t = x_1@\tau_1 + x_2@\tau_2\ \mathsf{in}\ e \Rightarrow v} \quad [\textsc{TypeDef}]$$

The result of $\mathsf{type}\ t = x_1@\tau_1 + x_2@\tau_2\ \mathsf{in}\ e$ equals the result of e. The constructors of the variants have to be available during the evaluation of e. For example, if the expression defines `Fruit`, then e should be able to construct values with the constructors, $\langle$`Apple`$\rangle$ and $\langle$`Banana`$\rangle$. Programmers can use the names of the variants to denote the constructors. They can write code like `Apple` 5 and `Banana` $(2, 6)$. It shows that the identifier `Apple` must denote the value $\langle$`Apple`$\rangle$ and that the identifier `Banana` must denote the value $\langle$`Banana`$\rangle$. For this reason, the environment used for the evaluation of e contains a mapping from x_1 to $\langle x_1 \rangle$ and a mapping from x_2 to $\langle x_2 \rangle$.

The other new expression is a pattern matching expression. Like the dynamic semantics of a conditional expression, we define two rules: one for when the target is handled by the first case and the other for when the target is handled by the second case.

Rule Match-L
If e evaluates to $x_1(v')$ under σ and e_1 evaluates to v under $\sigma[x_3 \mapsto v']$,
then $e\ \mathsf{match}\ x_1(x_3) \to e_1, x_2(x_4) \to e_2$ evaluates to v under σ.

$$\frac{\sigma \vdash e \Rightarrow x_1(v') \qquad \sigma[x_3 \mapsto v'] \vdash e_1 \Rightarrow v}{\sigma \vdash e\ \mathsf{match}\ x_1(x_3) \to e_1, x_2(x_4) \to e_2 \Rightarrow v} \quad [\textsc{Match-L}]$$

For pattern matching, the target has to be evaluated first. Therefore, e is the first expression to be evaluated. If the result of e is $x_1(v')$, then it matches the first case. Thus, e_1 is the next expression to be evaluated. During the evaluation of e_1, x_3 denotes v'. Therefore, the environment has a mapping from x_3 to v'. The result of e_1 is the result of the whole

pattern matching expression.

Rule Match-R
If e evaluates to $x_2(v')$ under σ and e_2 evaluates to v under $\sigma[x_4 \mapsto v']$, then e match $x_1(x_3) \to e_1, x_2(x_4) \to e_2$ evaluates to v under σ.

$$\frac{\sigma \vdash e \Rightarrow x_2(v') \qquad \sigma[x_4 \mapsto v'] \vdash e_2 \Rightarrow v}{\sigma \vdash e \text{ match } x_1(x_3) \to e_1, x_2(x_4) \to e_2 \Rightarrow v} \quad \text{[Match-R]}$$

On the other hand, if e evaluates to $x_2(v')$, it matches the second case. Then, e_2 is evaluated under the environment that contains a mapping from x_4 to v'. The result of e_2 is the result of the whole pattern matching expression.

In addition, we should define a new rule for function application. In TFAE, closures are the only values that can be applied to values. However, TVFAE has constructors, which also can be applied to values. We need a rule to handle such cases.

Rule App-Cnstr
If e_1 evaluates to $\langle x \rangle$ under σ and e_2 evaluates to v under σ, then $e_1\ e_2$ evaluates to $x(v)$ under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow \langle x \rangle \qquad \sigma \vdash e_2 \Rightarrow v}{\sigma \vdash e_1\ e_2 \Rightarrow x(v)} \quad \text{[App-Cnstr]}$$

If e_1 evaluates to the constructor of a variant named x, the application expression constructs a value of the variant.

Let e be `type Fruit = Apple@num+Banana@(num×num) in (Apple 5) match Apple(y) → y, Banana(z) → z.1`. The following proof tree proves that e evaluates to 5:

$$\dfrac{\dfrac{\dfrac{\texttt{Apple} \in \mathit{Domain}(\sigma_1)}{\sigma_1 \vdash \texttt{Apple} \Rightarrow \langle \texttt{Apple} \rangle} \qquad \sigma_1 \vdash 5 \Rightarrow 5}{\sigma_1 \vdash \texttt{Apple}\ 5 \Rightarrow \texttt{Apple}(5)} \qquad \dfrac{\texttt{y} \in \mathit{Domain}(\sigma_2)}{\sigma_2 \vdash \texttt{y} \Rightarrow 5}}{\sigma_1 \vdash (\texttt{Apple}\ 5)\ \texttt{match Apple(y)} \to \texttt{y, Banana(z)} \to \texttt{z.1} \Rightarrow 5}}{\emptyset \vdash e \Rightarrow 5}$$

where

$\sigma_1 = [\texttt{Apple} \mapsto \langle \texttt{Apple} \rangle, \texttt{Banana} \mapsto \langle \texttt{Banana} \rangle]$
$\sigma_2 = \sigma_1[\texttt{y} \mapsto 5]$

20.3 Interpreter

The following code implements expressions of TVFAE:

```
sealed trait Expr
...
case class TypeDef(
```

```
  t: String, v1: String, vt1: Type,
  v2: String, vt2: Type, b: Expr
) extends Expr
case class Match(
  e: Expr, v1: String, x1: String, e1: Expr,
  v2: String, x2: String, e2: Expr
) extends Expr
```

`TypeDef(`t`, `x_1`, `τ_1`, `x_2`, `τ_2`, `e`)` represents type $t = x_1@\tau_1 + x_2@\tau_2$ in e, and `Match(`e`, `x_1`, `x_3`, `e_1`, `x_2`, `x_4`, `e_2`)` represents e match $x_1(x_3) \to e_1, x_2(x_4) \to e_2$.

The following code implements values of TVFAE:

```
sealed trait Value
...
case class VariantV(x: String, v: Value) extends Value
case class ConstructorV(x: String) extends Value
```

`VariantV(`x`, `v`)` represents $x(v)$, and `ConstructorV(`x`)` represents $\langle x \rangle$.

```
case TypeDef(_, v1, _, v2, _, b) =>
  interp(b, env + (v1 -> ConstructorV(v1)) + (v2 -> ConstructorV(v2)))
```

An expression defining a type evaluates its body under the environment with the constructors.

```
case Match(e, v1, x1, e1, v2, x2, e2) =>
  interp(e, env) match {
    case VariantV(‘v1‘, v) => interp(e1, env + (x1 -> v))
    case VariantV(‘v2‘, v) => interp(e2, env + (x2 -> v))
  }
```

A pattern matching expression evaluates the target expression first. If the result is a variant value and its name is the same as `v1`, then `e1` is evaluated under the environment with the value of `x1`. If the name is the same as `v2`, then `e2` is evaluated under the environment with the value of `x2`.

```
case App(f, a) =>
  val fv = interp(f, env)
  val av = interp(a, env)
  fv match {
    case CloV(x, b, fEnv) =>
      interp(b, fEnv + (x -> av))
    case ConstructorV(x) =>
      VariantV(x, av)
  }
```

Function application allows a constructor to occur at the function position. If a constructor appears, the result is a variant value that contains the value denoted by the argument.

20.4 Static Semantics

To define the static semantics, the definition of a type environment should be revised first. In TFAE, type environments store the types of variables. They are finite partial functions from identifiers to types. In TVFAE, type environments have more things to do. They have to store the information about user-defined types. The type checking process uses the information. Let Γ be the type environment when $\mathsf{type}\ t = x_1@\tau_1 + x_2@\tau_2\ \mathsf{in}\ e$ is type-checked. Adding the information of t to Γ yields $\Gamma[t : \{(x_1, \tau_1), (x_2, \tau_2)\}]$. The variants are elements of a set since the order between them is unimportant. We will write $t = x_1@\tau_1 + x_2@\tau_2$ to denote $t : \{(x_1, \tau_1), (x_2, \tau_2)\}$ just for the sake of intuitive notation. The domain of a type environment now needs to include t, which is a type identifier. Also, the codomain has to contain $x_1@\tau_1 + x_2@\tau_2$. Below is the revised definition. Note that $\mathcal{P}(A)$ denotes the power set of A.

$$TEnv = (Id \cup TId) \overset{\text{fin}}{\twoheadrightarrow} (T \cup \mathcal{P}(Id \times T))$$

Well-Formed Types

An arbitrary type identifier can be a type in TVFAE. For example, `Fruit` is a type. It is true regardless of whether the type `Fruit` is bound by a type definition. Programmers can write λx:Fruit.x without defining `Fruit`. Such a function does not make sense at all since `Fruit` is a free type identifier, whose information is missing. Expressions with free type identifiers are weird and useless. Furthermore, they can break the type soundness of the type system.

To prevent free type identifiers, we introduce the notion of a well-formed type. A *well-formed* type is a type that does not contain any free type identifiers. The opposite of a well-formed type is an *ill-formed* type, which contains a free type identifier.

When a type appears in an expression, the type must be well-formed. Any expression that contains an ill-formed type is rejected by the type system. In this way, weird programs like λx:Fruit.x can be effectively prevented by type checking.

Now, we defined well-formed types. The well-formedness relation is a relation over type environments and types because we need type information in a type environment to decide whether a certain type identifier is free or not.

$$\vdash \subseteq TEnv \times T$$

$\Gamma \vdash \tau$ denotes that τ is well-formed under Γ.

Well-formedness rules prescribe which types are well-formed.

Rule WF-NUMT
num is well-formed under Γ.

$$\Gamma \vdash \mathsf{num} \quad [\text{WF-NUMT}]$$

The first well-formedness rule states that num is always well-formed. num is neither a type identifier nor the name of a user-defined type. It is a built-in type, which always exists. Thus, num is well-formed under any type environment.

Rule Wf-ArrowT
If τ_1 is well-formed under Γ and τ_2 is well-formed under Γ,
then $\tau_1 \to \tau_2$ is well-formed under Γ.

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \quad \text{[Wf-ArrowT]}$$

If both τ_1 and τ_2 are well-formed, then $\tau_1 \to \tau_2$ also is well-formed. The reason is clear: if both τ_1 and τ_2 lack free type identifiers, then $\tau_1 \to \tau_2$ also does.

Rule Wf-IdT
If t in the domain of Γ,
then t is well-formed under Γ.

$$\frac{t \in \mathit{Domain}(\Gamma)}{\Gamma \vdash t} \quad \text{[Wf-IdT]}$$

If a type identifier can be found in the type environment, the type identifier is a well-formed type. For example, if λx:Fruit.x is the whole expression, Fruit is ill-formed since there is no Fruit in the type environment. However, in type Fruit = Apple@num + Banana@(num × num) in λx:Fruit.x, Fruit is well-formed since the expression puts the definition of Fruit into the type environment.

Typing Rules

Before defining the typing rules, we need to classify newly introduced values. The type of a variant value is the type that defines the variant. For example, if x is a variant of t, then $x(v)$ is a value of t. The type of a constructor is a function type because constructors can be applied to values. Each constructor takes a value of the type that is specified in the type definition and returns a value of the type that constructor belongs to. For instance, if x is a variant of t and t defines the parameter type of x to be τ, then the type of x is $\tau \to t$.

Now, let us define the typing rules. First, consider expressions that define types.

Rule Typ-TypeDef
If
  Γ' denotes $\Gamma[t = x_1@\tau_1 + x_2@\tau_2, x_1 : \tau_1 \to t, x_2 : \tau_2 \to t]$,
  τ_1 is well-formed under Γ',
  τ_2 is well-formed under Γ', and
  the type of e is τ under Γ',
then
  the type of type $t = x_1@\tau_1 + x_2@\tau_2$ in e is τ under Γ.

$$\frac{\Gamma' = \Gamma[t = x_1@\tau_1 + x_2@\tau_2, x_1 : \tau_1 \to t, x_2 : \tau_2 \to t] \quad \Gamma' \vdash \tau_1 \quad \Gamma' \vdash \tau_2 \quad \Gamma' \vdash e : \tau}{\Gamma \vdash \mathsf{type}\ t = x_1@\tau_1 + x_2@\tau_2\ \mathsf{in}\ e : \tau} \quad [\textsc{Typ-TypeDef}]$$

Rule Typ-TypeDef defines the type of an expression that defines a new type. First, the definition of t must be added to the type environment. In addition, since e can use the constructors, the type environment should also contain the types of x_1 and x_2. Γ' denotes the type environment after adding the type definition and the constructors. Since τ_1 and τ_2 are user-written type annotations, they can be ill-formed types. To rule out ill-formed types, the well-formedness of τ_1 and τ_2 is checked. For the well-formedness checking, Γ' is used instead of Γ. The use of Γ' allows recursively defined types. Since Γ' contains the definition of t, τ_1 and τ_2 can be well-formed even when t occurs in them. Finally, e is type-checked under Γ'. The type of e is the type of the whole type-defining expression. Note that the type of e does not need well-formedness checking since it is the result of type checking, not a user-written type annotation. The same principle can be applied to all the other typing rules in TVFAE.

Rule Typ-Match
If
the type of e is t under Γ,
t is in the domain of Γ,
$\Gamma(t)$ equals $x_1@\tau_1 + x_2@\tau_2$,
the type of e_1 is τ under $\Gamma[x_3 : \tau_1]$, and
the type of e_2 is τ under $\Gamma[x_4 : \tau_2]$,
then
the type of e match $x_1(x_3) \to e_1, x_2(x_4) \to e_2$ is τ under Γ.

$$\frac{\Gamma \vdash e : t \quad t \in \mathit{Domain}(\Gamma) \quad \Gamma(t) = x_1@\tau_1 + x_2@\tau_2 \quad \Gamma[x_3 : \tau_1] \vdash e_1 : \tau \quad \Gamma[x_4 : \tau_2] \vdash e_2 : \tau}{\Gamma \vdash e\ \mathsf{match}\ x_1(x_3) \to e_1, x_2(x_4) \to e_2 : \tau} \quad [\textsc{Typ-Match}]$$

Rule Typ-Match defines the type of a pattern matching expression. First, the type of e, which is the target, is computed. Since pattern matching in TVFAE can be used for only user-defined types, the type of the target must be t, which is a type identifier. In addition, its definition must be found in the type environment. Since x_1 and x_2 in the cases denote the names of variants, the names of t's variants must be x_1 and x_2. Note that the order does not need to be the same. The pattern matching expression can place x_1 first while the type definition places x_2 first. The fact that the order does not matter is reflected in the rule by representing variant information as $x_1@\tau_1 + x_2@\tau_2$, which actually denotes a set $\{(x_1, \tau_1), (x_2, \tau_2)\}$. Finally, e_1 and e_2 are type-checked. The type of x_3 is τ_1 during the type checking of e_1, and the type of x_4 is τ_2 during the type checking of e_2. The type of e_1 must be the same as the type of e_2 because an expression can have at most one type. It is similar to the typing rule of a conditional expression. The common type to e_1 and e_2 is the type of the whole pattern matching expression.

In addition, one typing rule from TFAE needs a revision: Rule Typ-Fun.

Rule Typ-Fun
If τ_1 is well-formed under Γ and the type of e is τ_2 under $\Gamma[x : \tau_1]$, then the type of $\lambda x{:}\tau_1.e$ is $\tau_1 \to \tau_2$ under Γ.

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \quad [\text{Typ-Fun}]$$

A lambda abstraction also has a user-written type annotation, which might be ill-formed. To rule out ill-formed types, the well-formedness of the type annotation must be checked.

Let e be type `Fruit = Apple@num+Banana@(num×num)` in `(Apple 5) match Apple(y) → y, Banana(z) → z.1`. The following proof trees prove that the type of e is `num`:

$$\frac{\dfrac{\texttt{Apple} \in \mathit{Domain}(\Gamma_1)}{\Gamma_1 \vdash \texttt{Apple} : \text{num} \to \texttt{Fruit}} \qquad \Gamma_1 \vdash 5 : \text{num}}{\Gamma_1 \vdash \texttt{Apple}\ 5 : \texttt{Fruit}}$$

$$\frac{\text{y} \in \mathit{Domain}(\Gamma_2)}{\Gamma_2 \vdash \text{y} : \text{num}}$$

$$\frac{\dfrac{\text{z} \in \mathit{Domain}(\Gamma_3)}{\Gamma_3 \vdash \text{z} : \text{num} \times \text{num}}}{\Gamma_3 \vdash \text{z.1} : \text{num}}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \texttt{Apple}\ 5 : \texttt{Fruit} \\ \texttt{Fruit} \in \mathit{Domain}(\Gamma_1) \qquad \Gamma_1(\texttt{Fruit}) = \texttt{Apple@num} + \texttt{Banana@}(\text{num} \times \text{num}) \\ \Gamma_2 \vdash \text{y} : \text{num} \qquad \Gamma_3 \vdash \text{z.1} : \text{num}\end{array}}{\Gamma_1 \vdash (\texttt{Apple}\ 5)\ \texttt{match Apple(y)} \to \text{y}, \texttt{Banana(z)} \to \text{z.1} : \text{num}}$$

$$\frac{\begin{array}{c}\Gamma_1 = \Gamma_1 \qquad \Gamma_1 \vdash \text{num} \qquad \dfrac{\Gamma_1 \vdash \text{num} \qquad \Gamma_1 \vdash \text{num}}{\Gamma_1 \vdash \text{num} \times \text{num}} \\ \Gamma_1 \vdash (\texttt{Apple}\ 5)\ \texttt{match Apple(y)} \to \text{y}, \texttt{Banana(z)} \to \text{z.1} : \text{num}\end{array}}{\emptyset \vdash e : \text{num}}$$

where

$\Gamma_1 = [\texttt{Fruit} = \texttt{Apple@num} + \texttt{Banana@}(\text{num} \times \text{num}), \texttt{Apple} : \text{num} \to \texttt{Fruit}, \texttt{Banana} : (\text{num} \times \text{num}) \to \texttt{Fruit}]$
$\Gamma_2 = \Gamma_1[\text{y} : \text{num}]$
$\Gamma_3 = \Gamma_1[\text{z} : \text{num} \times \text{num}]$

20.5 Type Checker

First, we extend the definition of a type since TVFAE has a new sort of a type.

```
sealed trait Type
...
case class IdT(t: String) extends Type
```

`IdT(`t`)` represents t.

In TVFAE, type environments store type definitions and the types of variables. Thus, they cannot be represented by simple maps any longer. Now, they are represented by `TEnv` instances, each of which contains two maps.

```
case class TEnv(
  vars: Map[String, Type],
  tbinds: Map[String, Map[String, Type]]
) {
  def add(x: String, t: Type): TEnv =
    TEnv(vars + (x -> t), tbinds)

  def add(x: String, m: Map[String, Type]): TEnv =
    TEnv(vars, tbinds + (x -> m))

  def contains(x: String): Boolean =
    tbinds.contains(x)
}
```

`TEnv` has two fields: `vars` and `tbinds`. The field `vars`, which is a map from strings to TVFAE types, contains the types of variables. The field `tbinds`, which is a map from strings to maps, contains type definitions. Each map in `tbinds` maps strings, which are the names of variants, to TVFAE types, which are the parameter types of variants. For example, `tbinds` containing the `Fruit` type is as follows:

```
Map("Fruit" -> Map("Apple" -> NumT, "Banana" -> PairT(NumT, NumT)))
```

For the ease of adding type definitions and variables to type environments, the `TEnv` class has two methods named `add`. Adding that the type of a variable `x` is `num` to `env` can be written like below.

```
env.add("x", NumT)
```

Adding the `Fruit` type to `env` can be written like below.

```
env.add("Fruit", Map("Apple" -> NumT, "Banana" -> PairT(NumT, NumT)))
```

The `contains` method of the `TEnv` class checks whether a particular type identifier is a bound type identifier. For instance, the following code checks whether `Fruit` is bound:

```
env.contains("Fruit")
```

Now let us define a function that checks the well-formedness of a type. The following `wfType` function checkes whether a given type is well-formed under a given type environment:

```
def wfType(t: Type, env: TEnv): Unit = t match {
  case NumT =>
```

```
  case ArrowT(p, r) =>
    wfType(p, env)
    wfType(r, env)
  case IdT(t) =>
    if (!env.contains(t))
      throw new Exception
}
```

If the type is ill-formed under the type environment, the function throws an exception.

Now, we add the `TypeDef` and `Match` cases to the `typeCheck` function.

```
 case TypeDef(t, v1, vt1, v2, vt2, b) =>
  val nenv = env
    .add(t, Map(v1 -> vt1, v2 -> vt2))
    .add(v1, ArrowT(vt1, IdT(t)))
    .add(v2, ArrowT(vt2, IdT(t)))
  wfType(vt1, nenv)
  wfType(vt2, nenv)
  typeCheck(b, nenv)
```

First, the function adds the type definition and the constructors to the type environment. Then, it checks the well-formedness of the parameter types of the variants under the new type environment. If both are well-formed, it type-checks the body expression. The type of the body is the type of the whole type-defining expression.

```
case Match(e, v1, x1, e1, v2, x2, e2) =>
  val IdT(t) = typeCheck(e, env)
  val tdef = env.tbinds(t)
  val t1 = typeCheck(e1, env.add(x1, tdef(v1)))
  val t2 = typeCheck(e2, env.add(x2, tdef(v2)))
  mustSame(t1, t2)
  t1
```

First, the function type-checks the target expression. The type must be a type identifier. The definition of the type should be found in the type environment. The definition gives the parameter type of each variant. The function type-checks `e1` and `e2` under the type environments with the type of `x1` and with the type of `x2`, respectively. The types must be the same, and if it is the case, the common type is the type of the whole pattern mathcing expression.

```
case Fun(x, t, b) =>
  wfType(t, env)
  ArrowT(t, typeCheck(b, env.add(x, t)))
```

As discussed already, the `Fun` case needs a revision. The well-formedness of the parameter type annotation needs to be checked.

```
case Id(x) => env.vars(x)
```

The `Id` case also has a small change. Due to the new definition of a type environment, a way to find the type of a variable is a bit different.

20.6 Type Soundness of TVFAE

Actually, TVFAE is not type sound. If an expression defines multiple types of the same name, the expression can be well-typed and incur a run-time error at the same time. Consider the following expression:

```
type Fruit = Apple@num + Banana@(num × num) in (
  (λf:Fruit → num.
    type Fruit = Cherry@num + Durian@(num × num) in
    f (Cherry 1)
  ) (λx:Fruit.x match
    Apple(y) → y,
    Banana(z) → z.1
  )
)
```

The expression defines `Fruit` twice in a nested manner. The outer `Fruit` has `Apple` and `Banana` as variants, and the inner `Fruit` has `Cherry` and `Durian` as variants. The expression applies a function of (`Fruit` → num) → num to a value of `Fruit` → num, so it is well-typed. However, the expression causes a run-time error. The function of (`Fruit` → num) → num applies a given function to a value of the `Cherry` variant because `Fruit` has `Cherry` and `Durian` inside the function. However, the inner definition of `Fruit` is unavailable outside the function, so the argument given to the function is a function that expects a value of `Apple` or `Banana`. Thus, at run time, the pattern matching fails and incurs a run-time error.

The reason of broken type soundness is that the language allows multiple different types of the same name, while its type checking depends solely on the names of types to distinguish different types. Two different types may incorrectly considered as the same type when they have the same name.

There are multiple ways to fix the problem. The first solution is to prohibit local type definitions. Every type definition should be at top level, just like functions in F1VAE. Then, types cannot be nested, and every type must have a different name from each other. Since there cannot be different types of the same name, the problem is resolved.

The second solution is to prevent interaction between different types of the same name. It can be achieved by changing Rule Typ-TypeDef like below.

Rule Typ-TypeDef′
If
  t is not in the domain of Γ,
  Γ' denotes $\Gamma[t = x_1@\tau_1 + x_2@\tau_2, x_1 : \tau_1 \to t, x_2 : \tau_2 \to t]$,
  τ_1 is well-formed under Γ',
  τ_2 is well-formed under Γ',
  the type of e is τ under Γ', and
  τ is well-formed under Γ,
then
  the type of type $t = x_1@\tau_1 + x_2@\tau_2$ in e is τ under Γ.

$$\frac{t \notin \mathit{Domain}(\Gamma) \qquad \Gamma' = \Gamma[t = x_1@\tau_1 + x_2@\tau_2, x_1 : \tau_1 \to t, x_2 : \tau_2 \to t] \qquad \Gamma' \vdash \tau_1 \qquad \Gamma' \vdash \tau_2 \qquad \Gamma' \vdash e : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash \mathsf{type}\ t = x_1@\tau_1 + x_2@\tau_2\ \mathsf{in}\ e : \tau} \quad [\textsc{Typ-TypeDef}']$$

The rule has two new premises:

- $t \notin \mathit{Domain}(\Gamma)$
- $\Gamma \vdash \tau$

The first one prevents nested types from having the same name. The second one prevents each locally defined type from escaping its scope. In this way, we can effectively solve the issue. A program still can have different types of the same name, but different types of the same name cannot meet each other, i.e. they cannot be used in the same place.

The third solution is to rename types before type checking in order to remove any duplication in type names. Since the bound-bind relation between identifiers can be easily determined with simple syntactic checking, it is possible to rename types without changing the semantics of a given program. For instance, the above example becomes the following expression after renaming:

```
type Fruit = Apple@num + Banana@(num × num) in (
  (λf:Fruit → num.
    type Fruit1 = Cherry@num + Durian@(num × num) in
    f (Cherry 1)
  ) (λx:Fruit.x match
    Apple(y) → y,
    Banana(z) → z.1
  )
)
```

This expression is certainly ill-typed because `Cherry` 1 is a value of `Fruit1`, while `f`, which is applied to `Cherry` 1, is a function that expects a value of `Fruit`. This solution is desirable in the sense that it does not need any change in the language. It only requires one additional step of transformation, which can be easily implemented. On the other hand, the first solution changes the language at syntax level, and the second solution changes the static semantics of the language.

20.7 Exercises

1. What does each of the following expressions evaluate to? If it is a run-time error, describe where the error occurs.

   a) `type Fruit = Apple@num + Banana@num in`
   `type Animal = Apple@(num → num) + Banana@(num → num) in`
   `(λx:Fruit.x match Apple(y) → y, Banana(y) → y) (Banana 10)`

   b) `type Fruit = Apple@num + Banana@num in`
   `type Fruit = Apple@(num → num) + Banana@(num → num) in`
   `(λx:Fruit.x match Apple(y) → y, Banana(y) → y) (Banana 10)`

2. Consider the following expression:

```
type Fruit = Apple@num + Banana@num in
(type Color = Apple@num + Banana@num in
λx:Fruit.x match
  Apple(y) → Apple (y + 1),
  Banana(y) → Banana (y + 1)
) (Banana 10)
```

What is the result of type-checking the expression? If the result is `Fruit`, explain why in detail. Otherwise, revise the typing rules to make it `Fruit`.

3. This exercise extends TVFAE to have
   - functions with multiple parameters
   - types with more than two variants
   - variants with multiple parameters

   $$\begin{aligned} e &::= \cdots \\ &\mid \lambda(x{:}\tau, \ldots, x{:}\tau).e \\ &\mid e(e, \ldots, e) \\ &\mid \mathsf{type}\ t = x@(\tau, \ldots, \tau) + \ldots + x@(\tau, \ldots, \tau)\ \mathsf{in}\ e \\ &\mid e\ \mathsf{match}\ x(x, \ldots, x) \to e, \ldots, x(x, \ldots, x) \to e \\ \tau &::= \cdots \\ &\mid (\tau, \ldots, \tau) \to \tau \end{aligned}$$

   a) Write the well-formedness rule for the added type.
   b) Write the typing rules for the added expressions.
   c) Draw the type derivation of the following expression:

   ```
   type Fruit = Apple@() + Banana@(Fruit → num, Fruit) + Cherry@(num) in
   Apple() match
     Apple() → 42,
     Banana(f, x) → f(x),
     Cherry(x) → x
   ```

4. Suppose that Rule Typ-TypeDef has been changed as the following:

   $$\frac{\Gamma' = \Gamma[t = x_1@\tau_1 + x_2@\tau_2, x_1 : \tau_1 \to t, x_2 : \tau_2 \to t] \qquad \Gamma' \vdash \tau_2 \qquad \Gamma' \vdash e : \tau}{\Gamma \vdash \mathsf{type}\ t = x_1@\tau_1 + x_2@\tau_2\ \mathsf{in}\ e : \tau}$$

   It lacks the well-formedness check of τ_1. Write a well-typed expression whose evaluation results in a run-time error. Your expression cannot define multiple types of the same name whose scopes are overlapping.

21 Parametric Polymorphism

A function in TFAE is more restrictive than a function in FAE. Consider λx.x in FAE. It is an identity function, which takes a value as an argument and returns the value without changing it. Any value can be an argument for this function. Since the body of the function does nothing with the argument, the evaluation of the body never causes a run-time error. On the other hand, λx:num.x in TFAE is an identity function that takes only an integer. The parameter type annotation restricts the type of an argument to be only num. The type system rejects a program that passes a value other than an integer to the function. However, since the body simply returns the argument, a run-time error never happens even when the argument is not an integer. This example shows that a single identity function can be applied to a value of every type in FAE, while a different identity function is required for each type in TFAE.

Because of this restrictiveness, programmers have to define more functions in TFAE than in FAE to implement the same program. For example, the following FAE expression does not incur run-time errors:[1]

```
val f=λx.x in
val y=f 1 in
f true
```

1: Suppose that we extends FAE with booleans.

On the other hand, the following TFAE expression is rejected by the type system:

```
val f=λx:num.x in
val y=f 1 in
f true
```

The expression is ill-typed because `f true` on the last line applies a function of num $\rightarrow$ num to a value that does not belong to num. To make an equivalent and well-typed expression, we should define one more function like below.[2]

```
val f=λx:num.x in
val g=λx:bool.x in
val y=f 1 in
g true
```

2: Suppose that the type of a boolean is bool.

Now, the expression is well-typed since `g`, whose type is `bool` $\rightarrow$ `bool`, is applied to a value of `bool`.

Because of this problem, writing programs in TFAE is inconvenient. Programmers should define multiple functions of the same functionality to convince the type checker that their programs do not incur any run-time errors. Such functions of an overlapping role lead to duplicated code, which increases the code length and harms maintainability. When a program has plenty of functions and types, the amount of duplicated code will become huge.

Polymorphism can resolve the problem. *Polymorphism* is to use a single

entity as multiple types. For example, it may allow λx.x to have multiple types: num $\rightarrow$ num and bool $\rightarrow$ bool. There are three widely-used ways to realize polymorphism in a language: parametric polymorphism, subtype polymorphism, and ad-hoc polymorphism. The topic of this chapter is parametric polymorphism. Chapter 22 introduces subtype polymorphism, and ad-hoc polymorphism is beyond the scope of this book.

To introduce parametric polymorphism, we first need to discuss what parameterization is. Functions are well-known examples of parameterizing entities. Each function parameterizes an expression with a value (or an expression in the case of lazy languages). Consider λx.x + x. In this function, x is the parameter. The body, x + x is parameterized by x. This function is the most general form of adding a value to the same value. By applying the function, we can express any expresion that adds a value to the same value. For example, 1 + 1 is equivalent to (λx.x + x) 1, and 42 + 42 is equivalent to (λx.x + x) 42. A function abstracts an expression by replacing some portion of the expression with a parameter. By applying a function to values, multiple expressions can be expressed without repeating the common constituents. Only different parts should be written as an argument in each case.

Parametric polymorphism allows entities to be parameterized by types. It is a new form of parameterization, which functions do not provide. Parametric polymorphism allows parameterizing an expression with a type, instead of a value. To distinguish this new notion of parameterization from functions, we use the term *type abstraction*. While functions are applied to values to replace their parameters with real values, type abstractions are applied to types to replace their *type parameters* with real types. To differentiate application of type abstractions from application of functions, we use the term *type application*. Since type abstractions parameterize expressions, the results of type application are values, just like functions. The following table compares functions and type abstractions:

	Function	Type abstraction
What is parameterized?	Expression	Expression
With what?	Value	Type
Applied to what?	Value	Type
Result of application	Value	Value

Consider λx:num.x and λx:bool.x. The only difference is the type annotation: num and bool. We can parameterize both expressions with a type by introducing a type parameter α. By replacing num with α in λx:num.x, we obtain λx:α.x. Similarly, by replacing bool with α in λx:bool.x, we obtain λx:α.x. The results are exactly identical to each other. We can make a type abstraction that takes a type τ as a *type argument* and returns λx:τ.x as a result. This book uses Λ to denote type abstractions. Thus, the type abstraction we want is $\Lambda\alpha.\lambda$x:α.x. The type abstraction can be applied to types to recover the original expressions. This book uses [] to denote type application. Then, ($\Lambda\alpha.\lambda$x:α.x)[num] is equivalent to λx:num.x, and ($\Lambda\alpha.\lambda$x:α.x)[bool] is equivalent to λx:bool.x.

After adding parametric polymorphism, we can make the previous example well-typed while defining a function only once.

```
val f=Λα.λx:α.x in
val y=f[num] 1 in
f[bool] true
```

It is still more complex than the FAE version but defines a function only once, unlike the TFAE version.

Traditionally, parametric polymorphism was supported by only functional languages. For example, OCaml and Haskell have been well-known for their support for parametric polymorphism. On the other hand, object-oriented languages provided only subtype polymorphism. For instance, Java lacked parametric polymorphism until Java 4. However, programmers in these days require languages to provide more advanced features because their programs become more complicated. For this reason, Java has been supporting parametric polymorphism since Java 5. Many recent languages, such as Scala, provide both parametric and subtype polymorphism. In the context of object-oriented programming, parametric polymorphism is often called *generics* since it allows generic programming.

This chapter defines PTFAE by extending TFAE with parametric polymorphism. PTFAE is known as System F in the programming language community. System F was first discoverd by Girard in the context of logic in 1972 [Gir72]. Later, Reynolds independently discoverd the equivalent system in the context of computer science in 1974 [Rey74]. System F, or PTFAE, is the most foundatinoal formulation of parametric polymorphism, and its metatheory and variants are widely studied even in these days.

21.1 Syntax

First, we introduce type identifiers like in TVFAE. Type identifiers are used as type parameters. Let *TId* be the set of every type identifier. The metavariable α ranges over type identifiers.

$$\alpha \in TId$$

We add two sorts of an expression: type abstraction and type application.

$$e ::= \cdots \mid \Lambda\alpha.e \mid e[\tau]$$

- $\Lambda\alpha.e$ is a type abstraction. α is the type parameter, and e is the body. α in $\Lambda\alpha.e$ is a binding occurrence whose scope is e.
- $e[\tau]$ is a type application expression. e denotes the type abstraction applied to τ.

In addition, we add two sorts of a type.

$$\tau ::= \cdots \mid \alpha \mid \forall\alpha.\tau$$

- α is a type identifier as a type. Like in TVFAE, a type identifier can be used as a type. For example, in $\Lambda\alpha.\lambda x{:}\alpha.x$, the second α is the type of x.

- $\forall\alpha.\tau$ is a *universally quantified type*. It is the type of a type abstraction that takes a type as a type argument and returns a value of the type obtained by replacing every α with the given type in τ. α in $\forall\alpha.\tau$ is a binding occurrence whose scope is τ. For instance, the type of $\Lambda\alpha.\lambda x{:}\alpha.x$ is $\forall\alpha.\alpha \to \alpha$ since applying the type abstraction to num results in a function of num $\to$ num and applying to bool results in a function of bool $\to$ bool.

21.2 Dynamic Semantics

The addition of type abstractions adds a new sort of a value to the language.

$$v ::= \cdots \mid \langle \Lambda\alpha.e, \sigma \rangle$$

$\langle \Lambda\alpha.e, \sigma \rangle$ is a type abstraction value. It is similar to a closure. A closure is a function value, which contains the definition of a function and the environment of when the function is defined. Similarly, a type abstraction value contains the definition of a type abstraction and the environment of when the type abstraction is defined.

Now, we define the dynamic semantics of PTFAE. We should define the rules for the added expressions.

Rule TyAbs
$\Lambda\alpha.e$ evaluates to $\langle \Lambda\alpha.e, \sigma \rangle$ under σ.

$$\sigma \vdash \Lambda\alpha.e \Rightarrow \langle \Lambda\alpha.e, \sigma \rangle \quad [\textsc{TyAbs}]$$

A type abstraction evaluates to a type abstraction value. It is almost the same as Rule Fun.

Rule TyApp
If
e evaluates to $\langle \Lambda\alpha.e', \sigma' \rangle$ under σ and
$e'[\alpha \leftarrow \tau]$ evaluates to v under σ',
then
$e[\tau]$ evaluates to v under σ.

$$\frac{\sigma \vdash e \Rightarrow \langle \Lambda\alpha.e', \sigma' \rangle \qquad \sigma' \vdash e'[\alpha \leftarrow \tau] \Rightarrow v}{\sigma \vdash e[\tau] \Rightarrow v} \quad [\textsc{TyApp}]$$

To evaluate a type application expression, its only subexpression is evaluated. The result must be a type abstraction value. $e'[\alpha \leftarrow \tau]$ denotes an expression obtained by substituting every free occurrence of α with τ in e'.[3] Thus, the rule states that every occurrence of the type parameter is replaced with the given type argument in the body of the type abstraction. Finally, the expression obtained by the substitution is evaluated under the environment held by the type abstraction value. Its result is the result of the whole type application expression.

One may wonder why Rule TyApp needs substitution because types do

3: We do not discuss substitution in detail here. Interested readers can refer to Exercise 8 of Chapter 9, which introduces substition of an identifier with a value in an expression. The same principle applies to substition of a type identifier with a type in an expression.

not have any roles at run time. We can omit substitution in the dynamic semantics of PTFAE. However, if we extend the language to support any form of dynamic type testing, substitution is mandatory. In addition, if we want to prove type soundness, we should prove that every expression of a certain type evaluates to a value of the same type when the evaluation terminates. This property is called type preservation, and evaluation will not preserve the type of an expression if the rule omits substitution. For these reasons, Rule TyApp requires substitution.

The following proof trees prove that $(\Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x})[\mathsf{num}]\ 1$ evaluates to 1:

$$\frac{\emptyset \vdash \Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x} \Rightarrow \langle \Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x}, \emptyset\rangle \qquad \emptyset \vdash \lambda \mathsf{x}{:}\mathsf{num}.\mathsf{x} \Rightarrow \langle \lambda \mathsf{x}.\mathsf{x}, \emptyset\rangle}{\emptyset \vdash (\Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x})[\mathsf{num}] \Rightarrow \langle \lambda \mathsf{x}.\mathsf{x}, \emptyset\rangle}$$

$$\frac{\emptyset \vdash (\Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x})[\mathsf{num}] \Rightarrow \langle \lambda \mathsf{x}.\mathsf{x}, \emptyset\rangle \qquad \emptyset \vdash 1 \Rightarrow 1 \qquad \dfrac{\mathsf{x} \in \mathit{Domain}(\sigma_1)}{\sigma_1 \vdash \mathsf{x} \Rightarrow 1}}{\emptyset \vdash (\Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x})[\mathsf{num}]\ 1 \Rightarrow 1}$$

where $\sigma_1 = [\mathsf{x} \mapsto 1]$.

21.3 Static Semantics

Like in TVFAE, the definition of a type environment needs to be revised. Type environments should be able to store type identifiers in addition to the types of variables. Unlike type definitions in TVFAE, type identifiers in PTFAE do not have any further information. However, the type checking procedure needs to know whether a certain type identifier is free or not to determine the well-formedness of types. Thus, type environments store type identifiers to record their existence.

$$\mathit{TEnv} = (\mathit{Id} \cup \mathit{TId}) \xrightarrow{\text{fin}} (T \cup \{\cdot\})$$

Now, the codomain of a type environment contains $\cdot$, which is a meaningless mathematical object. For brevity, let $\Gamma[\alpha]$ denote $\Gamma[\alpha : \cdot]$.

Well-Formed Types

The well-formedness checking of PTFAE is similar to that of TVFAE. The following three rules are the same as those of TVFAE:

Rule Wf-NumT
num is well-formed under Γ.

$$\Gamma \vdash \mathsf{num} \quad [\text{Wf-NumT}]$$

Rule Wf-ArrowT
If τ_1 is well-formed under Γ and τ_2 is well-formed under Γ,
then $\tau_1 \to \tau_2$ is well-formed under Γ.

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \quad [\text{Wf-ArrowT}]$$

Rule Wf-IdT
If α is in the domain of Γ,
then α is well-formed under Γ.

$$\frac{\alpha \in \mathit{Domain}(\Gamma)}{\Gamma \vdash \alpha} \quad [\text{Wf-IdT}]$$

One remaining sort of a type is a universally quantified type, which is new in PTFAE.

Rule Wf-ForallT
If τ is well-formed under $\Gamma[\alpha]$,
then $\forall\alpha.\tau$ is well-formed under Γ.

$$\frac{\Gamma[\alpha] \vdash \tau}{\Gamma \vdash \forall\alpha.\tau} \quad [\text{Wf-ForallT}]$$

In $\forall\alpha.\tau$, α is a binding occurrence and, thus, can appear in τ. Therefore, α must be in the type environment when the well-formedness of τ is checked. For example, $\forall\alpha.\alpha$ is well-formed under the empty type environment, while $\forall\alpha.\beta$ is ill-formed under the same type environment.

Typing Rules

Now, let us define the typing rules of PTFAE.

Rule Typ-TyAbs
If the type of e is τ under $\Gamma[\alpha]$,
then the type of $\Lambda\alpha.e$ is $\forall\alpha.\tau$ under Γ.

$$\frac{\Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \quad [\text{Typ-TyAbs}]$$

The type of $\Lambda\alpha.e$ is $\forall\alpha.\tau$ if the type of e is τ. Since α is a binding occurrence whose scope is e, e should be type-checked under the type environment containing α.

Rule Typ-TyApp
If τ is well-formed under Γ and the type of e is $\forall\alpha.\tau'$ under Γ,
then the type of $e[\tau]$ is $\tau'[\alpha \leftarrow \tau]$ under Γ.

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash e : \forall\alpha.\tau'}{\Gamma \vdash e[\tau] : \tau'[\alpha \leftarrow \tau]} \quad [\text{Typ-TyApp}]$$

If the type of e is $\forall\alpha.\tau'$, the type of $e[\tau]$ is $\tau'[\alpha \leftarrow \tau]$, which is a type obtained by substituting α with τ in τ'. Since τ is a user-written type annotation, the well-formeness of τ must be checked.

In addition, like in TVFAE, Rule Typ-Fun has to check the well-formedness of the parameter type annotation.

Rule Typ-Fun
If τ_1 is well-formed under Γ and the type of e is τ_2 under $\Gamma[x : \tau_1]$, then the type of $\lambda x{:}\tau_1.e$ is $\tau_1 \rightarrow \tau_2$ under Γ.

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \rightarrow \tau_2} \quad [\text{Typ-Fun}]$$

The following proof tree proves that the type of $(\Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x})[\mathsf{num}]\ 1$ is num:

$$\dfrac{\dfrac{\emptyset \vdash \mathsf{num} \qquad \dfrac{\dfrac{\dfrac{\alpha \in \mathit{Domain}([\alpha])}{[\alpha] \vdash \alpha} \qquad \dfrac{\mathsf{x} \in \mathit{Domain}([\alpha, \mathsf{x} : \alpha])}{[\alpha, \mathsf{x} : \alpha] \vdash \mathsf{x} : \alpha}}{[\alpha] \vdash \lambda \mathsf{x}{:}\alpha.\mathsf{x} : \alpha \rightarrow \alpha}}{\emptyset \vdash \Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x} : \forall\alpha.\alpha \rightarrow \alpha}}{\emptyset \vdash (\Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x})[\mathsf{num}] : \mathsf{num} \rightarrow \mathsf{num}} \qquad \emptyset \vdash 1 : \mathsf{num}}{\emptyset \vdash (\Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x})[\mathsf{num}]\ 1 : \mathsf{num}}$$

The current type system of PTFAE has two problems. First, multiple type parameters of the same name breaks the type soundness, as multiple type definitions of the same name does in TVFAE. Second, syntactic comparison of types makes the type checking too restrictive. For example, if $\Lambda\beta.\lambda \mathsf{x}{:}\beta.\mathsf{x}$ is given to a function that expects a value of $\forall\alpha.\alpha \rightarrow \alpha$, syntactically comparing $\forall\alpha.\alpha \rightarrow \alpha$ and $\forall\beta.\beta \rightarrow \beta$ judges them to be different and makes the type checking reject the program. However, $\forall\alpha.\alpha \rightarrow \alpha$ and $\forall\beta.\beta \rightarrow \beta$ denote the same type semantically.

The best solution to both of the problems is de Bruijn indices, introduced in Chapter 17. Chapter 17 shows use of de Bruijn indices for expressions. However, de Bruijn indices are not limited to expressions; they can be applied to types. For instance, both $\forall\alpha.\alpha \rightarrow \alpha$ and $\forall\beta.\beta \rightarrow \beta$ can be represented with $\Lambda.\underline{0} \rightarrow \underline{0}$, so their semantic equivalence can be verified with syntactic comparison. In addition, de Bruijn indices prevent multiple types from being considered as the same type because of their names.

De Bruijn indices seem to be the best solution, but, still, other solutions can be used. The three solutions described in Chapter 20 can be applied to PTFAE in the same manner to resolve the first issue. The second issue can be fixed by renaming type parameters before the comparison. For example, simple syntactic transformation can transform $\forall\alpha.\alpha \rightarrow \alpha$ into $\forall\beta.\beta \rightarrow \beta$.

21.4 Exercises

1. Draw the type derivation of each of the following expressions:
   a) $(\lambda \mathsf{f}{:}\forall\alpha.\alpha \to \alpha.\mathsf{f}[\mathsf{num}]\ 10)\ (\Lambda\alpha.\lambda \mathsf{x}{:}\alpha.\mathsf{x})$
   b) $(\Lambda\alpha.\Lambda\beta.\lambda \mathsf{f}{:}\alpha \to \beta.\lambda \mathsf{x}{:}\alpha.\mathsf{f}\ \mathsf{x})[\mathsf{num}][\mathsf{num}]\ (\lambda \mathsf{y}{:}\mathsf{num}.17 - \mathsf{y})\ 9$
2. Rewrite the following code with type abstractions and type applications to replace all the occurrences of ? with types and to make function calls take explicit type arguments.
   $\mathsf{val}\ \mathsf{f}{=}\lambda \mathsf{g}{:}?.\lambda \mathsf{x}{:}?.\mathsf{g}\ \mathsf{x}\ \mathsf{in}$
   $\mathsf{val}\ \mathsf{g}{=}\lambda \mathsf{x}{:}?.\mathsf{x}\ \mathsf{in}$
   $\mathsf{f}\ \mathsf{g}\ 10$
3. Consider the following language:
   Syntax

$$
\begin{array}{llllll}
e ::= n & \mid \lambda x.e & \tau ::= \mathsf{num} & \sigma ::= \tau & \sigma \in \mathit{TScheme} \\
\quad \mid b & \mid e\ e & \quad \mid \mathsf{bool} & \quad \mid \forall\alpha.\sigma & \Gamma \in \mathit{Id} \xrightarrow{\text{fin}} \mathit{TScheme} \\
\quad \mid x & \mid \mathsf{val}\ x = e\ \mathsf{in}\ e & \quad \mid \tau \to \tau & b ::= \mathsf{true} & \\
\quad \mid e; e & & \quad \mid \alpha & \quad \mid \mathsf{false} &
\end{array}
$$

   Typing rules

$$\Gamma \vdash n : \mathsf{num} \quad \Gamma \vdash b : \mathsf{bool} \quad \frac{x \in \mathit{Domain}(\Gamma) \qquad \Gamma(x) \succ \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1; e_2 : \tau_2} \quad \frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \quad \frac{\Gamma \vdash e_1 : \tau \qquad \tau \prec_\Gamma \sigma \qquad \Gamma[x : \sigma] \vdash e_2 : \tau'}{\Gamma \vdash \mathsf{val}\ x = e_1\ \mathsf{in}\ e_2 : \tau'}$$

   Miscellaneous definitions

$$\boxed{\sigma \succ \tau} \quad \forall\alpha_1.\cdots\forall\alpha_n.\tau \succ \tau[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]$$

$$\boxed{\tau \prec_\Gamma \sigma} \quad \frac{\mathit{FTV}(\tau) \setminus \mathit{FTV}(\Gamma) = \{\alpha_1, \ldots, \alpha_n\}}{\tau \prec_\Gamma \forall\alpha_1.\cdots\forall\alpha_n.\tau}$$

$$
\begin{aligned}
\mathit{FTV}(\mathsf{num}) &= \emptyset \\
\mathit{FTV}(\mathsf{bool}) &= \emptyset \\
\mathit{FTV}(\tau_1 \to \tau_2) &= \mathit{FTV}(\tau_1) \cup \mathit{FTV}(\tau_2) \\
\mathit{FTV}(\alpha) &= \{\alpha\} \\
\mathit{FTV}(\forall\alpha.\sigma) &= \mathit{FTV}(\sigma) \setminus \{\alpha\} \\
\mathit{FTV}(\Gamma) &= \cup_{x \in \mathit{Domain}(\Gamma)} \mathit{FTV}(\Gamma(x))
\end{aligned}
$$

   For each of the following expressions, write whether the expression is well-typed. If so, draw its type derivation. Otherwise, explain why.
   a) $(\lambda \mathsf{x}.(\mathsf{x}\ 42))\ \lambda \mathsf{z}.\mathsf{z}$
   b) $(\lambda \mathsf{x}.((\mathsf{x}\ 42); (\mathsf{x}\ \mathsf{true})))\ \lambda \mathsf{z}.\mathsf{z}$
   c) $\mathsf{val}\ \mathsf{x} = \lambda \mathsf{z}.\mathsf{z}\ \mathsf{in}\ ((\mathsf{x}\ 42); (\mathsf{x}\ \mathsf{true}))$
4. Type annotations do not take any roles during evaluation. Therefore, this exercise defines type-erasure semantics, which removes type annotations from an expression. Consider the following Scala code, which implements type erasure:

```
sealed trait Expr
case class Num(n: Int) extends Expr
case class Add(l: Expr, r: Expr) extends Expr
```

```
case class Sub(l: Expr, r: Expr) extends Expr
case class Id(x: String) extends Expr
case class Fun(p: String, t: Type, b: Expr) extends Expr
case class App(f: Expr, a: Expr) extends Expr
case class TyFun(a: String, e: Expr) extends Expr
case class TyApp(e: Expr, t: Type) extends Expr

sealed trait Type
case object NumT extends Type
case class ArrowT(p: Type, r: Type) extends Type
case class VarT(a: String) extends Type
case class Forall(a: String, t: Type) extends Type

object Expr {
  sealed trait Expr
  case class Num(n: Int) extends Expr
  case class Add(l: Expr, r: Expr) extends Expr
  case class Sub(l: Expr, r: Expr) extends Expr
  case class Id(x: String) extends Expr
  case class Fun(p: String, b: Expr) extends Expr
  case class App(f: Expr, a: Expr) extends Expr
}

def erase(e: Expr): FAE.Expr = e match {
  case Num(n) => FAE.Num(n)
  case Add(l, r) => FAE.Add(erase(l), erase(r))
  case Sub(l, r) => FAE.Sub(erase(l), erase(r))
  case Id(x) => FAE.Id(x)
  case Fun(p, t, b) => FAE.Fun(p, erase(b))
  case App(f, a) => FAE.App(erase(f), erase(a))
  case TyFun(a, e) => erase(e)
  case TyApp(e, t) => erase(e)
}
```

Applying the type erasure to a PTFAE expression results in an FAE expression. The following is the syntax of FAE:

$$E ::= n \mid E + E \mid E - E \mid x \mid \lambda x.E \mid E\ E$$

a) Write the formal definition of the type erasure $\boxed{erase(e) = E}$ according to the Scala code.
b) Write the result of applying the type erasure to each of the following expressions:
   i. $(\Lambda\alpha.\lambda x{:}\alpha.x)[\mathsf{num}]\ 1$
   ii. $(\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.y)[\mathsf{num}][\mathsf{num}]\ 1\ 2$
c) Write a well-typed expression that becomes the following expression by the type erasure:
   $\lambda x.(x\ (\lambda x.x)\ (x\ 1))$

5. The following language adds polymorphic non-recursive type definitions to PTFAE (the omitted parts are the same as PTFAE):

$$e ::= \cdots \mid \mathsf{type}\ t[\alpha] = x@\tau + x@\tau;\ e \mid e\ \mathsf{match}\ x(x) \to e,\ x(x) \to e$$
$$v ::= \cdots \mid \Lambda\alpha.\langle x \rangle \mid \langle x \rangle \mid x(v)$$
$$\tau ::= \cdots \mid t[\tau]$$
$$\Gamma \in (Id \cup TId \cup TN) \xrightarrow{\text{fin}} (T \cup \{\cdot\} \cup (TId \times Id \times T \times Id \times T))$$

$$\frac{\sigma[x_1 \mapsto \Lambda\alpha.\langle x_1 \rangle, x_2 \mapsto \Lambda\alpha.\langle x_2 \rangle] \vdash e \Rightarrow v}{\sigma \vdash \mathsf{type}\ t[\alpha] = x_1@\tau_1 + x_2@\tau_2;\ e \Rightarrow v} \qquad \frac{\sigma \vdash e \Rightarrow \Lambda\alpha.\langle x \rangle}{\sigma \vdash e[\tau] \Rightarrow \langle x \rangle}$$

$$\frac{\sigma \vdash e_1 \Rightarrow \langle x \rangle \qquad \sigma \vdash e_2 \Rightarrow v}{\sigma \vdash e_1\ e_2 \Rightarrow x(v)}$$

$$\frac{\sigma \vdash e \Rightarrow x_1(v') \qquad \sigma[x_3 \mapsto v'] \vdash e_1 \Rightarrow v}{\sigma \vdash e\ \mathsf{match}\ x_1(x_3) \to e_1, x_2(x_4) \to e_2 \Rightarrow v}$$

$$\frac{\sigma \vdash e \Rightarrow x_2(v') \qquad \sigma[x_4 \mapsto v'] \vdash e_2 \Rightarrow v}{\sigma \vdash e\ \mathsf{match}\ x_1(x_3) \to e_1, x_2(x_4) \to e_2 \Rightarrow v}$$

For example, programmers can write the following code, which defines a polymorphic option type, in this language:

```
type option[α] = None@num + Some@α;
val getOrElse = Λα.λx:option[α].λy:α.(
  x match
    None(z) → y,
    Some(z) → z
);
getOrElse[num] (Some[num] 1) 2
```

On the other hand, the following code is not well-typed since types are not recursive in this language:

```
type foo[α] = bar@num + baz@foo[α];
...
```

Note that `foo` appears in the definition of itself, which implies that `foo` is a recursive type.

a) Write the typing rules of the form $\boxed{\Gamma \vdash e : \tau}$ of $\mathsf{type}\ t[\alpha] = x_1@\tau_1 + x_2@\tau_2;\ e$ and $e\ \mathsf{match}\ x_1(x_3) \to e_1,\ x_2(x_4) \to e_2$.
b) Write the well-formedness rule of the form $\boxed{\Gamma \vdash \tau}$ of $t[\tau]$.
c) Write the type of `getOrElse` in the above example.

22 Subtype Polymorphism

This chapter introduces subtype polymorphism. Subtype polymorphism is often found in object-oriented languages, but some functional languages also support subtype polymorphism these days. For example, OCaml, which stands for Objective Caml, is a functional language that provides subtype polymorphism. Understanding subtype polymorphism is important in many real-world languages.

This chapter defines STFAE by extending TFAE with subtype polymorphism. To illustrate the need of subtype polymorphism, we start with adding records to TFAE. We can add subtype polymorphism to TFAE without adding records together, but examples with records can clearly show the benefits of subtype polymorphism.

22.1 Records

A record is a value that consists of multiple values. Records are similar to tuples, but users can designate the names of the elements in a record. When we talk about the elements of records, we often use the term *fields*. From the perspective of their expressivity and roles in programming, records are the same as tuples. However, records help programmers express their high-level ideas in the code more directly than tuples and prevent mistakes.

For example, consider a tuple representing the height, score, and scholarship state of a student. The tuple $(180, 91, \mathsf{true})$ represents a student who is 180-centimeter-tall, got 91 points from the recent exam, and is currently receiving a scholarship. One disadvantage of using tuples is that programmers may use wrong elements by mistake. When `st` denotes the previously mentioned tuple, one should write `st.1` to get the height. However, there is no reason to associate the first element with the height. If he or she writes `st.2` instead `st.1`, then the wrong conclusion—the student is 91-centimeter-tall—will be obtained.

Records effectively prevent such mistakes. We can represent the same student with the record $\{\texttt{height} = 180, \texttt{score} = 91, \texttt{scholarship} = \mathsf{true}\}$. This record has three fields: `height`, `score`, and `scholarship`. Then, the height can be found by `st.height`. In this case, there is a logical reason to associate the field whose name is `height` with the height of the student. It is clear that `st.score` does not denote the height of the student.

Syntax

First, we introduce labels, which are the names of fields in records. Let $\mathscr{L}$ be the set of every label and the metavariable l ranges over labels.

$$l \in \mathscr{L}$$

Now, we define the syntax of expressions related to records.

$$e \ ::= \ \cdots \mid \{l = e, \cdots, l = e\} \mid e.l$$

- $\{l_1 = e_1, \cdots, l_n = e_n\}$ is an expression that creates a record. A record has zero or more fields. l's are the names of the fields. We assume that the fields of a single record have distinct names. Since a record can have zero fields, $\{\}$ is an expression.
- $e.l$ is an expression that gives the value of a certain field from a record. It is usually called a *projection*. e determines the record, and l is the name of the field whose value is acquired.

Dynamic Semantics

To make a language support records, we should add record values to the language.

$$v \ ::= \ \cdots \mid \{l = v, \cdots, l = v\}$$

A value $\{l_1 = v_1, \cdots, l_n = v_n\}$ is a record value. l's are the names of the fields. v_i is the value of the field l_i. For example, the result of $\{\mathsf{a} = 1 + 2, \mathsf{b} = 3 + 4\}$ is $\{\mathsf{a} = 3, \mathsf{b} = 7\}$. The record has two fields: `a` and `b`. The value of `a` is 3, and the value of `b` is 7. The result of $\{\}$ is $\{\}$, which is the empty record value.

Now, we add rules to define the dynamic semantics of the added expressions.

Rule Record
If e_1 evaluates to v_1 under σ, $\cdots$, and e_n evaluates to v_n under σ,
then $\{l_1 = e_1, \cdots, l_n = e_n\}$ evaluates to $\{l_1 = v_1, \cdots, l_n = v_n\}$ under σ.

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \cdots \qquad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash \{l_1 = e_1, \cdots, l_n = e_n\} \Rightarrow \{l_1 = v_1, \cdots, l_n = v_n\}} \ [\textsc{Record}]$$

Every e_i has to be evaluated for the evaluation of $\{l_1 = e_1, \cdots, l_n = e_n\}$. If the value of e_i is v_i, the value of the field l_i is v_i. The result is $\{l_1 = v_1, \cdots, l_n = v_n\}$.

Rule Proj
If e evaluates to $\{\cdots, l = v, \cdots\}$ under σ,
then $e.l$ evaluates to v under σ.

$$\frac{\sigma \vdash e \Rightarrow \{\cdots, l = v, \cdots\}}{\sigma \vdash e.l \Rightarrow v} \ [\textsc{Proj}]$$

For the evaluation of $e.l$, e has to be evaluated first. If the result of e is a record that contains a field named l, then $e.l$ results in the value of the field. If e evaluates to a nonrecord value or a record value that does not contain l, $e.l$ incurs a run-time error.

Static Semantics

Since records are a new sort of a value, exsiting types cannot include records. We need to add new types that records can belong to.

$$\tau \ ::= \ \cdots \mid \{l : \tau, \cdots, l : \tau\}$$

A type $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ is a record type that includes records whose fields follow the designated names and types. In other words, if a record value has fields named l_1 from l_n and the value of a field l_i is a value of τ_i, then the value belongs to $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$. For example, the type of the value $\{\mathsf{a} = 3, \mathsf{b} = 7\}$ is $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$. In addition, the type of the expression $\{\mathsf{a} = 1 + 2, \mathsf{b} = 3 + 4\}$, which evaluates to $\{\mathsf{a} = 3, \mathsf{b} = 7\}$, also is $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$. Similarly, the type of the empty record is $\{\}$.

Let us define the typing rules for the added expressions.

Rule Typ-Record
If the type of e_1 is τ_1 under Γ, $\cdots$, the type of e_n is τ_n under Γ,
then the type of $\{l_1 = e_1, \cdots, l_n = e_n\}$ is $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ under Γ.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1, \cdots, l_n = e_n\} : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}} \quad [\text{Typ-Record}]$$

Let the type of e_i be τ_i. Then, e_i evaluates to a value of τ_i. Thus, $\{l_1 = e_1, \cdots, l_n = e_n\}$ evaluates to a value of $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$, and the type of $\{l_1 = e_1, \cdots, l_n = e_n\}$ is $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$.

Rule Typ-Proj
If the type of e is $\{\cdots, l : \tau, \cdots\}$ under Γ,
then the type of $e.l$ is τ under Γ.

$$\frac{\Gamma \vdash e : \{\cdots, l : \tau, \cdots\}}{\Gamma \vdash e.l : \tau} \quad [\text{Typ-Proj}]$$

$e.l$ can be evaluated without an error only if e evaluates to a record containing a field named l. Therefore, the type of e has to be a record type that contains a field l. Then, the type of $e.l$ is the type of l.

For example, the type of $\{\mathsf{a} = 1 + 2, \mathsf{b} = 3 + 4\}.\mathsf{a}$ is num since the type of $\{\mathsf{a} = 1 + 2, \mathsf{b} = 3 + 4\}$ is $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$. On the other hand, $\{\mathsf{a} = 1 + 2, \mathsf{b} = 3 + 4\}.\mathsf{c}$ is ill-typed because $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$ lacks a field named c.

22.2 Subtype Polymorphism

The current type system is sound but not expressive enough. It rejects many expressions that do not cause any run-time errors. Consider the following expression:

$(\lambda$x:{a : num}.x.a) {a = 1, b = 2}

The expression evaluates {a = 1, b = 2}.a, which yields 1 without any error. However, the type system rejects the expression. The type of {a = 1, b = 2} is {a : num, b : num}, while the parameter type of the function is {a : num}. Since the argument type is different from the designated parameter type, the application expression is ill-typed.

Currently, the type {a : num} denotes the set of every record that has only the integer-valued field a. However, this definition is too restrictive. The type implies that its value can be used for any place that accesses the field a and expects the value of the field to be an integer. Thus, the type does not need to exclude values that have other fields in addition to the field a.

To resolve the problem, we extend the meaning of {a : num}. Now, the type includes any records that have an integer-valued field a. Records that have additional fields also can be values of {a : num}. This change can be attained by modifying Rule Typ-Record like below.

Rule Typ-Record'
If the type of e_1 is τ_1 under $\Gamma, \cdots$, the type of e_n is τ_n under $\Gamma, \cdots$, the type of e_{n+m} is τ_{n+m} under Γ,
then the type of $\{l_1 = e_1, \cdots, l_n = e_n, \cdots, l_{n+m} = e_{n+m}\}$ is $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ under Γ.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n \quad \cdots \quad \Gamma \vdash e_{n+m} : \tau_{n+m}}{\Gamma \vdash \{l_1 = e_1, \cdots, l_n = e_n, \cdots, l_{n+m} = e_{n+m}\} : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}} \quad \text{[Typ-Record']}$$

The rule allows forgetting the types of some fields if they are unnecessary. Now, {a = 1, b = 2} is a value of {a : num}. Thus, the previous example, $(\lambda$x:{a : num}.x.a) {a = 1, b = 2}, is well-typed.

Does this fix solve all the problems? Unfortunately, no. Consider the following expression:

```
val x={a = 1, b = 2} in
val y=(λx:{a : num}.x.a) x in
(λx:{a : num, b : num}.x.a + x.b) x
```

This expression is still ill-typed though it does not incur any run-time errors. If we say the type of x is {a : num}, the first function application is well-typed. However, the second function application is ill-typed. If we say the type of x is {a : num, b : num} instead, the second function application becomes well-typed. However, the first function application becomes ill-typed. There is no way to make both application expressions well-typed. We need a way to consider x as an expression of {a : num} and as an expression of {a : num, b : num} at the same time. In other words, we should assign multiple types to a single entity, and this is the

notion of polymorphism.

Subtype polymorphism is one way of polymorphism, which is based on the notion of subtyping. Recall that a type is a set of values. Sometimes, one type is a subset of another type. For example, any values that belong to $\{\mathtt{a} : \mathsf{num}, \mathtt{b} : \mathsf{num}\}$ are values of $\{\mathtt{a} : \mathsf{num}\}$, so $\{\mathtt{a} : \mathsf{num}, \mathtt{b} : \mathsf{num}\}$ is a subset of $\{\mathtt{a} : \mathsf{num}\}$. When τ_1 is a subset of τ_2, we say that τ_1 is a *subtype* of τ_2 and τ_2 is a *supertype* of τ_1. For example, $\{\mathtt{a} : \mathsf{num}, \mathtt{b} : \mathsf{num}\}$ is a subtype of $\{\mathtt{a} : \mathsf{num}\}$ and $\{\mathtt{a} : \mathsf{num}\}$ is a supertype of $\{\mathtt{a} : \mathsf{num}, \mathtt{b} : \mathsf{num}\}$. This is the notion of subtyping.

Roughly speaking, subtyping is an "A is a B" relation betwen types. As an example, consider `Animal` and `Cat`, which denote the type of every animal and the type of every cat, respectively. We know that a cat is an animal. Then, we can say that `Cat` is a subtype of `Animal`. On the other hand, can we say that an animal is a cat? No, because there is an animal that is not a cat. For instance, a dog is an animal, but not a cat. Thus, `Animal` is not a subtype of `Cat`. We can do the same thing for record types. A record that has fields `a` and `b` is a record that has `a`. (For brevity, ignore the types of the fields here.) Therefore, $\{\mathtt{a} : \mathsf{num}, \mathtt{b} : \mathsf{num}\}$ is a subtype of $\{\mathtt{a} : \mathsf{num}\}$. On the other hand, a record that has `a` is not a record that has both `a` and `b` since it may lack `b`. As a consequence, $\{\mathtt{a} : \mathsf{num}\}$ is not a subtype of $\{\mathtt{a} : \mathsf{num}, \mathtt{b} : \mathsf{num}\}$.

Mathematically, subtyping is a relation over types and types.

$$<: \subseteq T \times T$$

We write $\tau_1 <: \tau_2$ to denote that τ_1 is a subtype of τ_2. For example, both $\mathtt{Cat} <: \mathtt{Animal}$ and $\{\mathtt{a} : \mathsf{num}, \mathtt{b} : \mathsf{num}\} <: \{\mathtt{a} : \mathsf{num}\}$ are true.

Now, let us see how subtyping induces polymorphsim. The key insight is that any expression of τ_1 can be treated as an expression of τ_2 without breaking type soundness when τ_1 is a subtype of τ_2. For example, suppose that there is an animal hospital that cures any animal. We can consider the hospital as a function that takes a value of `Animal`. A cat is an animal, so any cat can be cured in the hospital. Thus, if an expression evaluates to a value of `Cat`, it can be considered as an expression that evaluates to a value of `Animal` and safely given to the function representing the hospital. On the other hand, the inverse is false. If a hospital cures only cats and we know only that someone has an animal, then we cannot say to him or her to carry the animal to the hospital. There is no guarantee that the hospital will be able to cure the animal. Thus, the fact that τ_1 is a subtype of τ_2 does not imply that any expression of τ_2 can be treated as an expression of τ_1. In a similar fasion, any expression of $\{\mathtt{a} : \mathsf{num}, \mathtt{b} : \mathsf{num}\}$ can be treated as an expression of $\{\mathtt{a} : \mathsf{num}\}$, but the inverse is false. We can express this idea with the following typing rule:

Rule Typ-Sub
If the type of e is τ' under Γ and τ' is a subtype of τ,
then the type of e is τ under Γ.

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' <: \tau}{\Gamma \vdash e : \tau} \quad [\textsc{Typ-Sub}]$$

The rule states that if e is an expression of τ', then it is an expression of τ at the same time when τ' is a subtype of τ.

Subtyping has two important characteristics. It is reflexive and transitive. The reason is clear. Every set is a subset of itself. Thus, every type is a subtype of itself. In addition, if A is a subset of B and B is a subset of C, then A is a subset of C. Therefore, if τ_1 is a subtype of τ_2 and τ_2 is a subtype of τ_3, then τ_1 is a subtype of τ_3. We can formally state these properties with the following *subtyping rules*, which describe when two types are in the subtype relation.

Rule Sub-Refl
τ is a subtype of τ.

$$\tau <: \tau \quad \text{[Sub-Refl]}$$

Rule Sub-Trans
If τ_1 is a subtype of τ_2 and τ_2 is a subtype of τ_3,
then τ_1 is a subtype of τ_3.

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \quad \text{[Sub-Trans]}$$

If we have only Rule Sub-Refl and Rule Sub-Trans, we cannot prove any interesting subtype relationships, e.g. $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{a} : \mathsf{num}\}$, even though both of the rules can help us prove interesting subtype relationships by being used with other subtyping rules. Thus, we introduce subtyping rules for record types in the next section.

22.3 Subtyping of Record Types

Consider the previous example again. The type system should be able to prove $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{a} : \mathsf{num}\}$. To achieve the goal, we define the following subtyping rule:

Rule Sub-Width
$\{l_1 : \tau_1, \cdots, l_n : \tau_n, l : \tau\}$ is a subtype of $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$.

$$\{l_1 : \tau_1, \cdots, l_n : \tau_n, l : \tau\} <: \{l_1 : \tau_1, \cdots, l_n : \tau_n\} \quad \text{[Sub-Width]}$$

Any value of $\{l_1 : \tau_1, \cdots, l_n : \tau_n, l : \tau\}$ is a value of $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ because a record that has fields from l_1 to l_n and l additionally is a record that has fields from l_1 to l_n. Therefore, the rule is valid.

Now, $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{a} : \mathsf{num}\}$ is true. Using this fact, the following proof tree proves that $\{\mathsf{a} : \mathsf{num}\}$ is a type of $\{\mathsf{a} = 1, \mathsf{b} = 2\}$:

$$\dfrac{\dfrac{\emptyset \vdash 1 : \mathsf{num} \qquad \emptyset \vdash 2 : \mathsf{num}}{\emptyset \vdash \{\mathsf{a} = 1, \mathsf{b} = 2\} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}} \qquad \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{a} : \mathsf{num}\}}{\emptyset \vdash \{\mathsf{a} = 1, \mathsf{b} = 2\} : \{\mathsf{a} : \mathsf{num}\}}$$

In addition, the following expression is now well-typed:

```
val x={a = 1, b = 2} in
val y=(λx:{a : num}.x.a) x in
(λx:{a : num, b : num}.x.a + x.b) x
```

$\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$ is a type of x, so the second function application is well-typed. At the same time, by Rule Typ-Sub, $\{\mathsf{a} : \mathsf{num}\}$ also is a type of x, so the first function application is well-typed as well.

If we use Rule Sub-Width and Rule Sub-Trans together, other interesting subtype relationships can be proven. For example, the following proof tree proves that $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}, \mathsf{c} : \mathsf{num}\}$ is a subtype of $\{\mathsf{a} : \mathsf{num}\}$.

$$\dfrac{\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}, \mathsf{c} : \mathsf{num}\} <: \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} \qquad \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{a} : \mathsf{num}\}}{\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}, \mathsf{c} : \mathsf{num}\} <: \{\mathsf{a} : \mathsf{num}\}}$$

By the same principle, $\{\}$, which is the empty record type, is a supertype of every record type. In other words, every record type is a subtype of $\{\}$.

Alas, the type system is still restrictive. The following expression is ill-typed but does not cause run-time errors:

```
val x={a = 1, b = 2} in
val y=(λx:{b : num, a : num}.x.a + x.b) x in
(λx:{a : num, b : num}.x.a + x.b) x
```

The type of x is $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$. Therefore, the second function application is well-typed, while the first function application is not. We need to make x be a value of $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$ and a value of $\{\mathsf{b} : \mathsf{num}, \mathsf{a} : \mathsf{num}\}$ at the same time. Like before, fixing Rule Typ-Record cannot be a proper solution. The correct solution is to add a new subtyping rule.

The key idea to define a new subtyping rule is that the order between fields does not matter at all. For example, a record that has fields a and b is a record that has fields b and a. Thus, it is safe to consider $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$ as a subtype of $\{\mathsf{b} : \mathsf{num}, \mathsf{a} : \mathsf{num}\}$. By generalizing this observation, we define the following subtyping rule:

Rule Sub-Perm
If $(l_1, \tau_1), \cdots, (l_n, \tau_n)$ is a permutation of $(l'_1, \tau'_1), \cdots, (l'_n, \tau'_n)$, then $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ is a subtype of $\{l'_1 : \tau'_1, \cdots, l'_n : \tau'_n\}$.

$$\dfrac{(l_1, \tau_1), \cdots, (l_n, \tau_n) \text{ is a permutation of } (l'_1, \tau'_1), \cdots, (l'_n, \tau'_n)}{\{l_1 : \tau_1, \cdots, l_n : \tau_n\} <: \{l'_1 : \tau'_1, \cdots, l'_n : \tau'_n\}} \quad \text{[Sub-Perm]}$$

The rule states that altering the order between the fields of a record type results in a subtype of the record type.

The following proof tree proves that $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$ is a subtype of $\{\mathsf{b} : \mathsf{num}, \mathsf{a} : \mathsf{num}\}$:

$$\frac{(\mathsf{a}, \mathsf{num}), (\mathsf{b}, \mathsf{num}) \text{ is a permutation of } (\mathsf{b}, \mathsf{num}), (\mathsf{a}, \mathsf{num})}{\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{b} : \mathsf{num}, \mathsf{a} : \mathsf{num}\}}$$

If we use multiple subtyping rules together, other interesting subtype relationships can be proven. For example, the following proof tree proves that $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$ is a subtype of $\{\mathsf{b} : \mathsf{num}\}$.

$$\frac{\dfrac{(\mathsf{a}, \mathsf{num}), (\mathsf{b}, \mathsf{num}) \text{ is a permutation of } (\mathsf{b}, \mathsf{num}), (\mathsf{a}, \mathsf{num})}{\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{b} : \mathsf{num}, \mathsf{a} : \mathsf{num}\}} \quad \{\mathsf{b} : \mathsf{num}, \mathsf{a} : \mathsf{num}\} <: \{\mathsf{b} : \mathsf{num}\}}{\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{b} : \mathsf{num}\}}$$

Even after the addition of Rule Sub-Width and Rule Sub-Perm, the type system still can be improved more. Consider the following expression:

```
val x={a = {a = 1, b = 2}} in
val y=(λx:{a : {a : num}}.x.a.a) x in
(λx:{a : {a : num, b : num}}.x.a.a + x.a.b) x
```

The above expression does not incur any run-time errors. However, the first function application is ill-typed, while the second function application is well-typed. We need to make $\{\mathsf{a} = \{\mathsf{a} = 1, \mathsf{b} = 2\}\}$ be a value of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}\}\}$ and a value of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}\}$ at the same time by adding a subtyping rule.

The current type system is too strict about the types of fields in records. For example, consider $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}\}$ and $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}\}\}$. A value of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}\}$ has at least one field, whose name is a. The value of the field is a value of $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$. We already know that any value of $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$ is a value of $\{\mathsf{a} : \mathsf{num}\}$. Therefore, we can say that a value of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}\}$ has at least one field, whose name is a and type is $\{\mathsf{a} : \mathsf{num}\}$. In fact, it is the characteristic of a value of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}\}\}$. As a result, any value of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}\}$ is a value of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}\}\}$ at the same time, so $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}\}$ must be a subtype of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}\}\}$. By generalizing this observation, we define the following subtyping rule:

Rule Sub-Depth
If τ_1 is a subtype of τ_1', $\cdots$, τ_n is a subtype of τ_n',
then $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ is a subtype of $\{l_1 : \tau_1', \cdots, l_n : \tau_n'\}$.

$$\frac{\tau_1 <: \tau_1' \qquad \cdots \qquad \tau_n <: \tau_n'}{\{l_1 : \tau_1, \cdots, l_n : \tau_n\} <: \{l_1 : \tau_1', \cdots, l_n : \tau_n'\}} \quad \text{[Sub-Depth]}$$

The rule states that strengthening[1] the type of each field in a record type results in a subtype of the record type.

By using Rule Sub-Width and Rule Sub-Depth together, we can prove that $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}\}$ is a subtype of $\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}\}\}$.

1: If we strengthen a type, a subtype is obtained in the sense that a subtype imposes a stronger condition on its value than the original type.

$$\frac{\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} <: \{\mathsf{a} : \mathsf{num}\}}{\{\mathsf{a} : \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}\} <: \{\mathsf{a} : \{\mathsf{a} : \mathsf{num}\}\}}$$

22.4 Subtyping of Function Types

It is time to consider a subtyping rule for function types. A function type consists of a parameter type and a return type. Let us discuss return types first.

Consider two function types: $\tau_1 \to \tau_2$ and $\tau_1 \to \tau_2'$. Assume that τ_2 is a subtype of τ_2'. A value of $\tau_1 \to \tau_2$ is a function that takes a value of τ_1 and returns a value of τ_2. Since τ_2 is a subtype of τ_2', any value of τ_2 is a value of τ_2'. Therefore, we can say that a function of $\tau_1 \to \tau_2$ returns a value of τ_2'. It implies that the function is a value of $\tau_1 \to \tau_2'$ at the same time. Thus, any value of $\tau_1 \to \tau_2$ is a value of $\tau_1 \to \tau_2'$, and $\tau_1 \to \tau_2$ is a subtype of $\tau_1 \to \tau_2'$. The following rule formalizes this fact:

Rule Sub-Ret
If τ_2 is a subtype of τ_2',
then $\tau_1 \to \tau_2$ is a subtype of $\tau_1 \to \tau_2'$.

$$\frac{\tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1 \to \tau_2'} \quad \text{[Sub-Ret]}$$

Function types preserve the subtype relationship between their return types. Suppose that there are two function types. If their parameter types are the same and the return type of the former is a subtype of the return type of the latter, then the former is a subtype of the latter. For example, $\mathsf{num} \to \{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\}$ is a subtype of $\mathsf{num} \to \{\mathsf{a} : \mathsf{num}\}$.

Now, let us discuss parameter types. Consider two function types: $\tau_1 \to \tau_2$ and $\tau_1' \to \tau_2$. Assume that τ_1' is a subtype of τ_1. A value of $\tau_1 \to \tau_2$ is a function that takes a value of τ_1 and returns a value of τ_2. Since τ_1' is a subtype of τ_1, any value of τ_1' is a value of τ_1. Therefore, a function of $\tau_1 \to \tau_2$ should work properly when a value of τ_1' is given. We can say that the function takes a value of τ_1' and returns a value of τ_2. It implies that the function is a value of $\tau_1' \to \tau_2$ at the same time. Thus, any value of $\tau_1 \to \tau_2$ is a value of $\tau_1' \to \tau_2$, and $\tau_1 \to \tau_2$ is a subtype of $\tau_1' \to \tau_2$. The following rule formalizes this fact:

Rule Sub-Param
If τ_1' is a subtype of τ_1,
then $\tau_1 \to \tau_2$ is a subtype of $\tau_1' \to \tau_2$.

$$\frac{\tau_1' <: \tau_1}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2} \quad \text{[Sub-Param]}$$

Function types reverse the subtype relationship between their parameter types. Suppose that there are two function types. If their return types are the same and the parameter type of the former is a supertype of the parameter type of the latter, then the former is a subtype of the latter. For

example, $\{\mathsf{a} : \mathsf{num}\} \to \mathsf{num}$ is a subtype of $\{\mathsf{a} : \mathsf{num}, \mathsf{b} : \mathsf{num}\} \to \mathsf{num}$.

We can combine Rule Sub-Ret and Rule Sub-Param to form a single rule.

Rule Sub-ArrowT
If τ_1' is a subtype of τ_1 and τ_2 is a subtype of τ_2',
then $\tau_1 \to \tau_2$ is a subtype of $\tau_1' \to \tau_2'$.

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'} \quad \text{[Sub-ArrowT]}$$

22.5 Top and Bottom Types

Now, we add two types to STFAE.

$$\tau ::= \cdots \mid \mathsf{top} \mid \mathsf{bottom}$$

top is the *top type*. The type denotes the set of every value. The set is a superset of any set of values. Thus, the top type is a supertype of every type. In other words, every type is a subtype of the top type. The following is the subtyping rule for the top type:

Rule Sub-TopT
τ is a subtype of top.

$$\tau <: \mathsf{top} \quad \text{[Sub-TopT]}$$

The top type can be used to give a single type to two or more completely irrelevant expressions. Suppose that the language has conditional expressions. Then, the type of the following expression is $\{\mathsf{a} : \mathsf{num}\}$:

$\mathsf{if0}\ 0\ \{\mathsf{a} = 1\}\ \{\mathsf{a} = 1, \mathsf{b} = 2\}$

However, the following expression is ill-formed in STFAE when the top type does not exist:

$\mathsf{if0}\ 0\ \{\mathsf{a} = 1\}\ 1$

By extending STFAE with the top type, the type of the above expression can be top.

bottom is the *bottom type*, which is the dual of the top type. The bottom type denotes the empty set. Since the empty set is a subset of any set, the bottom type is a subtype of every type, and every type is a supertype of the bottom type. The following is the subtyping rule for the bottom type:

Rule Sub-BottomT
bottom is a subtype of τ.

$$\mathsf{bottom} <: \tau \quad \text{[Sub-BottomT]}$$

Even though no value is a value of the the bottom type, the bottom type is useful. It can be the type of expressions that throw exceptions or call first-class continuations. Those expressions do not evaluate to any values. They just change control flows. Thus, it is quite natural to say that the type of such an expression is the bottom type.

22.6 Exercises

1. Write whether each expression is well-typed in STFAE without the top type, If so, draw the type derivation. Otherwise, explain why.
   a) if0 1 {} 2
   b) if0 1 {} {a = 2}
2. Consider TFAE with lists in Exercise 1 of Chapter 18.
   a) When can list τ_1 be a subtype of list τ_2? Write a new subtyping rule for list types.
   b) Suppose that we extend the language as follows:

      $e ::= \cdots \mid e[e] := e$
      $\tau ::= \cdots \mid \mathsf{top}$

      The typing rule for the new expression, which mutates an element of a list, is as follows:

      $$\frac{\Gamma \vdash e_1 : \mathsf{list}\ \tau \qquad \Gamma \vdash e_2 : \mathsf{num} \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1[e_2] := e_3 : \tau}$$

      When can list τ_1 be a subtype of list τ_2? Write a new subtyping rule for list types.
3. If we change Rule Sub-ArrowT like below, the language is not type sound.

   $$\frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

   Write an expression that is accepted by the new type system but causes a run-time error.
4. Consider TFAE with boxes in Exercise 2 of Chapter 18. When can box τ_1 be a subtype of box τ_2? Write a new subtyping rule for box types.
5. This exercise extends STFAE with first-class continuations.

   $e ::= \cdots \mid (\mathsf{vcc}\ x\ \mathsf{in}\ e){:}\tau$

   The type of $(\mathsf{vcc}\ x\ \mathsf{in}\ e){:}\tau$ is τ when it is well-typed.

   a) Write the typing rule of $(\mathsf{vcc}\ x\ \mathsf{in}\ e){:}\tau$ of the form $\boxed{\Gamma \vdash e : \tau}$.
   b) Draw the type derivation tree of (vcc x in (x 1) 42):num.
6. This exercse extends TVFAE to allow types with any number (including zero) of variants.

   $e ::= \cdots \mid \mathsf{type}\ t = x@\tau + \ldots + x@\tau;\ e \mid e\ \mathsf{match}\ x(x) \to e, \ldots, x(x) \to e$

   For example, you can write the following code in the extended language:

   ```
   type fruit = apple@num + banana@num + cherry@num;
   ...
   ```

Suppose that the operational semantics and the typing rules are the same as those of TVFAE except that some rules are revised to handle zero or more variants properly.
Some expressions are rejected by the type system even though they do not cause run-time errors. The following expression is such an example:

```
type abc = apple@num + banana@num + cherry@num;
val f = λx:abc.(
  x match
    apple(a) → a
    banana(b) → b
    cherry(c) → c
);
type ab = apple@num + banana@num;
f (apple 42)
```

We want to add subtyping to the language to allow more expressions including the above one. Add subtyping rule(s) of the form $\boxed{\Gamma \vdash \tau <: \tau}$ to the language. Assume that the following rules already exist:

$$\Gamma \vdash \tau <: \tau \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}$$

$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma \vdash \tau_2 <: \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'} \qquad \frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash e : \tau}$$

Appendix

Solutions to Selected Exercises | A

Chapter 3

1.
```
def names(l: List[Student]): List[String] = l match {
  case Nil => Nil
  case h :: t => h.name :: names(t)
}
```
2.
```
def tall(l: List[Student]): List[Student] = l match {
  case Nil => Nil
  case h :: t =>
    if (h.height > 170)
      h :: tall(t)
    else
      tall(t)
}
```
3.
```
def length(l: List[Int]): Int = l match {
  case Nil => 0
  case h :: t => 1 + length(t)
}
```
4.
```
def append(l: List[Int], n: Int): List[Int] = l match {
  case Nil => n :: Nil
  case h :: t => h :: append(t, n)
}
```
$O(n)$

Chapter 4

1. `def incBy(l: List[Int], n: Int): List[Int] = l.map(_ + n)`
2. `def gt(l: List[Int], n: Int): List[Int] = l.filter(_ > n)`
3.
```
def append(l: List[Int], n: Int): List[Int] =
  l.foldRight(n :: Nil)(_ :: _)
```
4.
```
def reverse(l: List[Int]): List[Int] =
  l.foldLeft(Nil: List[Int])((l, e) => e :: l)
```

Bibliography

Here are the references in citation order.

[OSV16] Martin Odersky, Lex Spoon, and Bill Venners. 'Programming in Scala: Updated for Scala 2.12'. In: *Artima Incorporation, USA,* (2016) (cited on pages 5, 20).

[CB14] Paul Chiusano and Rnar Bjarnason. *Functional programming in Scala*. Manning Publications Co., 2014 (cited on page 5).

[Rey09] John C Reynolds. *Theories of programming languages*. Cambridge University Press, 2009 (cited on page 133).

[Tai67] William W Tait. 'Intensional interpretations of functionals of finite type I'. In: *The journal of symbolic logic* 32.2 (1967), pp. 198–212 (cited on page 197).

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002 (cited on page 197).

[Gir72] Jean-Yves Girard. 'Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur'. PhD thesis. Éditeur inconnu, 1972 (cited on page 220).

[Rey74] John C Reynolds. 'Towards a theory of type structure'. In: *Programming Symposium*. Springer. 1974, pp. 408–425 (cited on page 220).

Special Terms

A
ADT algebraic data type. 43
AST abstract syntax tree. 69

B
BNF Backus-Naur form. 63

C
CBN call-by-name. 129
CBR call-by-reference. 126
CBV call-by-value. 126
CPS continuation-passing style. 142

J
JVM Java Virtual Machine. 7

R
REPL read-eval-print loop. 8

Alphabetical Index