Preface

This document contains the lecture notes for the NYU undergraduate course CSCI-UA.0480-003, “Principles of Programming Languages”, in fall 2015. The document will be extended throughout the semester. So please stay tuned!

Course Summary

Computing professionals have to learn new programming languages all the time. This course teaches the fundamental principles of programming languages that enable you to learn new languages quickly and help you decide which one is best suited for a given task.

We will explore new ways of viewing computation and programs, and new ways of approaching algorithmic problems, making you better programmers overall. The topics covered in this course include recursion and induction, algebraic data types and pattern matching, higher-order functions, continuations and tail recursion, programming language syntax and semantics, type systems, monads, and objects and classes. We will explore this material by building interpreters for programming languages of increasing complexity. The course will thus be accompanied by extensive programming assignments. We will use the programming language Scala for these assignments, which you will also learn in this course.
# Contents

1 Scala Basics
   1.1 Getting Started ............................................. 11
      1.1.1 Compiling and Running Scala Applications ............ 11
      1.1.2 The Scala REPL and Worksheets in the Scala IDE ... 12
   1.2 Scala Crash Course .......................................... 13
      1.2.1 Expressions, Values, and Types ............... 13
      1.2.2 Names ............................................. 14
      1.2.3 Functions .......................................... 15
      1.2.4 Scopes ............................................ 16
      1.2.5 Tuples ............................................ 17
   1.3 Recursion ..................................................... 18
      1.3.1 Evaluating Recursive Functions ............... 18
      1.3.2 Tail Recursion .................................. 20
   1.4 Classes and Objects ......................................... 22
      1.4.1 Classes, Fields, and Methods ............... 22
      1.4.2 Overriding Methods .................................. 24
      1.4.3 Singleton and Companion Objects ............ 25
   1.5 Algebraic Data Types ......................................... 26
      1.5.1 Case Classes ........................................ 26
      1.5.2 Pattern Matching .................................... 28
      1.5.3 Binding Names in Patterns ...................... 30
      1.5.4 Pattern Guards ..................................... 30
      1.5.5 Sealed Classes .................................... 31
      1.5.6 Option Types ...................................... 33

2 Foundations ..................................................... 35
   2.1 Structural Recursion and Induction .............. 35
      2.1.1 Structurally Recursive Definitions .......... 35
      2.1.2 Recursive Definitions of Functions .......... 37
      2.1.3 Structural Induction .................................. 38
      2.1.4 Well-founded Induction (optional) ............ 40
   2.2 Lists ......................................................... 41
      2.2.1 Defining Lists using Structural Recursion ... 41
      2.2.2 Functions on Lists .................................. 42
6 Types

6.1 Type Checking and Type Inference
   6.1.1 Type Checking .......................... 103
   6.1.2 A Simple Typed Language ................. 104
   6.1.3 Operational Semantics .................... 105
   6.1.4 Typing Relation ........................ 105
   6.1.5 Limitations ............................ 107

6.2 Soundness of Static Type Checking
   6.2.1 Soundness Proof (optional) .......... 109

6.3 Parametric Polymorphism (optional)
   6.3.1 Type Inference without Type Annotations ........................................... 113
   6.3.2 The Hindley-Milner Type System ....... 121

7 Imperative Programming

7.1 Variables and Assignments
   7.1.1 A Simple Language with Variables and Assignments ...................... 127
   7.1.2 Operational Semantics ............................................. 130
   7.1.3 Type Checking .................................................. 133

7.2 Parameter Passing Modes
   7.2.1 Parameter Passing Variants ..................... 134
   7.2.2 A Simple Language with Parameter Passing Modes .......................... 139
   7.2.3 Operational Semantics ........................................... 139
   7.2.4 Type Checking .................................................. 139
   7.2.5 Custom Control Constructs with Call by Name .................................. 141

7.3 Monads
   7.3.1 A Stateful JAKARTA SCRIPT Interpreter ......................... 144
   7.3.2 Monads and for Expressions in Scala .................................. 144
   7.3.3 The State Monad ................................................ 148

8 Object-Oriented Programming

8.1 Objects and Fields
   8.1.1 A Simple Untyped Language with Objects ......................................... 155
   8.1.2 Operational Semantics ............................................. 157

8.2 Basic Features of OO Languages
   8.2.1 Methods ................................................. 158
   8.2.2 Classes and Encapsulation ........................................ 159
   8.2.3 Calling Superclass Methods ...................................... 162
   8.2.4 Open Recursion .............................................. 162

8.3 Subtyping
   8.3.1 Typing Objects ........................................... 163
   8.3.2 Structural Subtyping ........................................... 164
   8.3.3 Subtyping and Assignments ...................................... 165
   8.3.4 Type Inference .............................................. 173
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Inference rules that define the big-step SOS of our expression language</td>
<td>70</td>
</tr>
<tr>
<td>4.2</td>
<td>Do rules for the small-step SOS of our expression language</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>Search rules for the small-step SOS of our expression language</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>Inference rules that define the big-step SOS of our language with functions. The modifications and additions to the rules from Figure 4.1 in Section 4.1.1 are highlighted.</td>
<td>85</td>
</tr>
<tr>
<td>5.2</td>
<td>Big-step operational semantics: dynamic type error rules</td>
<td>86</td>
</tr>
<tr>
<td>5.3</td>
<td>Inference rules that define the substitution-based big-step SOS for static binding semantics of our language with functions</td>
<td>90</td>
</tr>
<tr>
<td>5.4</td>
<td>Substitution-based big-step operational semantics: dynamic type error rules</td>
<td>91</td>
</tr>
<tr>
<td>6.1</td>
<td>Small-step operational semantics for typed JAKARTASCRIP T</td>
<td>106</td>
</tr>
<tr>
<td>6.2</td>
<td>Type inference rules</td>
<td>108</td>
</tr>
<tr>
<td>6.3</td>
<td>Typing constraint generation rules for the Hindley-Milner type system</td>
<td>122</td>
</tr>
<tr>
<td>6.4</td>
<td>A simple unification algorithm</td>
<td>123</td>
</tr>
<tr>
<td>7.1</td>
<td>Big-step operational semantics of imperative primitives</td>
<td>131</td>
</tr>
<tr>
<td>7.2</td>
<td>Big-step operational semantics of non-imperative primitives. The only changes compared to Figure 5.3 are the threading of the memory and the omission of implicit type conversions.</td>
<td>132</td>
</tr>
<tr>
<td>7.3</td>
<td>Type checking rules for non-imperative primitives (no changes compared to Figure 6.2)</td>
<td>134</td>
</tr>
<tr>
<td>7.4</td>
<td>Type checking rules for imperative primitives</td>
<td>135</td>
</tr>
<tr>
<td>7.5</td>
<td>New inference rules that define the big-step semantics of function call expressions for the different parameter passing modes</td>
<td>140</td>
</tr>
<tr>
<td>7.6</td>
<td>New type checking rules for function and call expressions</td>
<td>141</td>
</tr>
<tr>
<td>7.7</td>
<td>Representing in Scala the abstract syntax of JAKARTASCRIP T with variables and assignments</td>
<td>145</td>
</tr>
<tr>
<td>7.8</td>
<td>Partial implementation of the eval function of the stateful interpreter</td>
<td>146</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>7.9</td>
<td>Curried version of the interpreter shown in Figure 7.8</td>
<td>149</td>
</tr>
<tr>
<td>7.10</td>
<td>Implementation of the state monad</td>
<td>151</td>
</tr>
<tr>
<td>7.11</td>
<td>Monadic version of the interpreter shown in Figure 7.9</td>
<td>152</td>
</tr>
<tr>
<td>7.12</td>
<td>Variant of the interpreter shown in Figure 7.11 with for expressions</td>
<td>153</td>
</tr>
<tr>
<td>8.1</td>
<td>New evaluation rules for the language primitives related to objects</td>
<td>158</td>
</tr>
<tr>
<td>8.2</td>
<td>Typing rules for objects</td>
<td>164</td>
</tr>
<tr>
<td>8.3</td>
<td>The subtyping rule</td>
<td>165</td>
</tr>
<tr>
<td>8.4</td>
<td>The inference rules that define the subtyping relation</td>
<td>166</td>
</tr>
<tr>
<td>8.5</td>
<td>Algorithmic subtyping rules for JAKARTASCRIPT</td>
<td>175</td>
</tr>
<tr>
<td>8.6</td>
<td>Typing rules with subtyping</td>
<td>175</td>
</tr>
<tr>
<td>8.7</td>
<td>Rules for computing joins</td>
<td>180</td>
</tr>
<tr>
<td>8.8</td>
<td>Rules for computing meets</td>
<td>183</td>
</tr>
</tbody>
</table>
Chapter 1

Scala Basics

1.1 Getting Started

In the following, we assume that you have installed sbt and the ScalaIDE. If you have not yet done so, please do it now. You can find the installation instructions on the course web site.

1.1.1 Compiling and Running Scala Applications

Compiling and running Scala applications works similar to Java. For example, you can open a text editor and type in the following Scala code:

```scala
package greeter

object hello extends app {
  println("hello world!")
}
```

This code creates an object `hello` in the package `greeter`. The object `hello` extends `app`, which means that `hello` can serve as the entry point of a Scala application. When this application is started, the object `hello` prints the message "Hello_world!" on standard output.

The above code is roughly equivalent to the following Java code:

```java
package greeter;

class Hello {
  public static void main(String[] args) {
    System.out.println("Hello_World!");
  }
}
```

You can save the Scala code, say in a file called `Hello.scala`, and then compile it with the Scala compiler. To do this, open a command prompt, go to
the directory where you saved the file, and type `scalac Hello.scala`. This will create a file `Hello.class`, which contains the compiled byte code of the object `Hello`.

To execute the program, type `scala greeter.Hello` in your terminal. This will start the Scala runtime environment, which will execute the byte code in `Hello.class` using the Java virtual machine. You should see the message "Hello World!" printed in your terminal.

If you are using the Scala Eclipse IDE, you can create and run the Hello World application as follows:

1. Start the Scala IDE by executing `eclipse`.
2. If this is the first time you start the Scala IDE, Eclipse will ask you to create a workspace. You can create a new workspace that you can use for this course.
3. Choose File/New/Scala Project in the menu. Call the project `popl` and press Finish.
4. Right-click the project `popl` in the package explorer and choose New/Package. Give the package the name `greeter` and press Finish.
5. Right-click the package `greeter` in the package explorer and choose New/Scala application. Type `hello` under Object name.
6. Press Finish. This will create a file `hello.scala`, which will be opened in the editor.
7. In the open `hello.scala` file, add the `println` statement to the body of the object `Hello`.
8. Save the file. The file will be automatically compiled.
9. Right-click the file `hello.scala` in the package explorer and choose Run as/Scala application. You should see the greeting message "Hello World!" printed in the console tab at the bottom of the Eclipse window.

### 1.1.2 The Scala REPL and Worksheets in the Scala IDE

If you want to experiment with the Scala language, it is quite cumbersome to write an extra application for every small code snippet that you would like to be executed. To make life easier, Scala provides a useful tool called a read-eval-print loop, or short REPL. The Scala REPL is similar to a command line calculator. It allows you to type arbitrary Scala code in a terminal. The code is then evaluated and the result of the evaluation is printed in the terminal.

To start the Scala REPL, open a terminal and `scala`. This will start the REPL and a message similar to the following should appear:
Welcome to Scala version 2.11.7 (OpenJDK 64-Bit Server VM, Java 1.7.0_65).
Type in expressions to have them evaluated.
Type :help for more information.

scala>

Now, you can type a Scala expression. For example typing 3 + 4 yields

scala> 3 + 4
res0: Int = 7

You can exit the REPL by typing :quit or by pressing Ctrl-d.

If you only installed sbt and the Scala IDE, you can start the REPL by
typing `sbt console` in a terminal. The Scala Eclipse IDE provides a feature
similar to the REPL, called Worksheets. You can use this feature as follows:

1. Start the Scala Eclipse IDE. (In the following, I assume you have previ-
ously created the POPL project with the greeter package.)
2. Right-click the `greeter` package in the package explorer and choose New/S-
   cala Worksheet. Name the worksheet Worksheet and press Finish.
3. A Scala source file called `worksheet.sc` with a predefined object called
   Worksheet will open in the editor window.
4. You can now type Scala expressions in the body of the Worksheet object.
   Each time you save the file, the expressions are evaluated and the result
   of the evaluation appears as a comment behind each expression.

1.2 Scala Crash Course

In the following, we assume that you have started the Scala REPL. Though,
(almost) all of these steps can also be done in a Scala IDE worksheet.

1.2.1 Expressions, Values, and Types

After you type an expression in the REPL, such as 3 + 4, and hit enter:

scala> 3 + 4

The interpreter will print:

res0: Int = 7

This line includes:

- an automatically generated name `res0`, which refers to the value resulting
  from evaluating the expression,
- a colon `:`, followed by the type `Int` of the expression,
1 Scala Basics

- an equals sign =,
- the value 7 resulting from evaluating the expression.

The type `Int` names the class `Int` in the package `scala`. Packages in Scala partition the global name space and provide mechanisms for information hiding, similar to Java packages. Values of class `Int` correspond to values of Java’s primitive type `int` (Scala makes no difference between primitive and object types). More generally, all of Java’s primitive types have corresponding classes in the `scala` package.

We can reuse the automatically generated name `res0` to refer to the computed value in subsequent expressions (this only works in the REPL but not in a worksheet):

```scala
cscala> res0 * res0
res1: Int = 9
```

Java’s ternary conditional operator `?:` has an equivalent in Scala, which looks as follows:

```scala
cscala> if (res1 > 10) res0 - 5 else res0 + 5
res2: Int = -2
```

In addition to the `?:` operator, Java also has if-then-else statements. Scala, on the other hand, is a functional language and makes no difference between expressions and statements: every programming construct evaluates to some value. In particular, we can use if-then-else expressions where we would normally use if-then-else statements in Java.

```scala
cscala> if (res1 > 2) println("Large!")
else println("Not so large!"))
res3: Unit = ()
```

In this case, the if-then-else expression evaluates to the value `()`, which is of type `Unit`. This type indicates that the sole purpose of evaluating the expression is the side-effect of the evaluation (here, printing a message on standard output). In other words, in Scala, statements are expressions of type `Unit`. Thus, the type `Unit` is similar to the type `void` in Java (which however, has no values). The value `()` is the only value of type `Unit`.

### 1.2.2 Names

We can use the `val` keyword to give a user-defined name to a value, so that we can subsequently refer to it in other expressions:

```scala
cscala> val x = 3
x: Int = 3
cscala> x * x
res0: Int = 9
```
Note that Scala automatically infers that x has type Int. Sometimes, automated type inference fails, in which case you have to provide the type yourself. This can be done by annotating the declared name with its type:

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

A `val` is similar to a `final` variable in Java. That is, you cannot reassign it another value:

```scala
scala> x = 5
<console>:8: error: reassignment to val
 x = 5
```

Scala also has an equivalent to standard Java variables, which can be reassigned. These are declared with the `var` keyword:

```scala
scala> var y = 5
y: Int = 5
scala> y = 3
y: Int = 3
```

The type of a variable is the type inferred from its initialization expression. It is fixed throughout the lifetime of the variable. Attempting to reassign a value of incompatible type results in a type error:

```scala
scala> y = "Hello"
<console>:8: error: type mismatch;
  found   : String("Hello")
  required: Int
    y = "Hello"
```

However, for the time being we will pretend that variables do not exist. Repeat after me: `vals` are gooood! `vars` are baaaad!

### 1.2.3 Functions

Here is how you write functions in Scala:

```scala
scala> def max(x: Int, y: Int): Int = {
    |   if (x > y) x
    |   else y
  }
max: (x: Int, y: Int)Int
```

Function definitions start with `def`, followed by the function’s name, in this case `max`. After the name comes a comma separated list of parameters enclosed by parenthesis, here `x` and `y`. Note that the types of parameters must be provided
explicitly since the Scala compiler does not infer parameter types. The type
annotation after the parameter list gives the result type of the function. The
result type is followed by the equality symbol, indicating that the function
returns a value, and the body of the function which computes that value. The
expression in the body that defines the result value is enclosed in curly braces.

If the defined function is not recursive, as is the case for max, the result type
can be omitted because it is automatically inferred by the compiler. However, it
is often helpful to provide the result type anyway to document the signature of
the function. Moreover, if the function body only consists of a single expression
or statement, the curly braces can be omitted. Thus, we could alternatively
write the function max like this:

```scala
scala> def max2(x: Int, y: Int) = if (x > y) x else y
max2: (x: Int, y: Int)Int

Once you have defined a function, you can call it using its name:

scala> max(6, 3)
res3: Int = 3

Naturally, you can use values and functions that are defined outside of a
function's body in the function's body:

scala> val pi = 3.14159
pi: Double = 3.14159

scala> def circ(r: Double) = 2 * pi * r
circ: (x: Double)Double

You can also nest value and function definitions:

scala> def area(r: Double) = {
    |   val pi = 3.14159
    |   def square(x: Double) = x * x
    |   pi * square(r)
    |
area:(Double)Double

Recursive functions can be written as expected. For example, the following
function fac computes the factorial numbers:

scala> def fac(n: Int): Int = if (n <= 0) 1 else n*fac(n-1)
fac: (n: Int)Int

scala> fac(5)
res4: Int = 120

1.2.4 Scopes

Scala's scoping rules are almost identical to Java's:
val a = 5
// only a in scope
{
  val b = 4
  // b and a in scope

def f(x: Int) = {
  // f, x, b, and a in scope
  a * x + b
}
  // f, b, and a in scope
}
// only a in scope

There is one difference to Java, though. Scala allows you to redefine names in nested scopes, thereby shadowing definitions in outer scopes.

val a = 3
{
  val a = 4 // shadows outer definition of a
  a + a    // yields 8
}

However, as in Java, you cannot redefine a name in the same scope:

val a = 3
val a = 4 // does not compile

1.2.5 Tuples

Scala provides ways to create new compound data types without requiring you to define simplistic data-heavy classes. One of the most useful of these constructs are tuples. A tuple combines a fixed number of items together so that they can be passed around as a whole. The individual items can have different types. For example, here is a tuple holding an Int and a String:

scala> val p = (1, "banana")
p: (Int, String) = (1, "banana")

and here is a tuple holding three items: two Strings and the console:

scala> val q = ("apple", "pear", Console)
q: (String, String, Console.type) = (apple, pear, scala.Console$@47fbb6b9)

To access the items of a tuple, you can use method _1 to access the first item, method _2 to access the second, and so on:

scala> p._1
res5: Int = 1

scala> p._2
res6: String = banana
Additionally, you can assign each element of the tuple to its own `val`:

```scala
scala> val (fst, snd) = p
fst: Int = 1
snd: String = banana
```

Be aware that tuples are not automatically decomposed when you pass them to functions:

```scala
def f(x: Int, s: String) = x
f(p._1, p._2) // works
f(p) // does not compile
```

```scala
def g(p: (Int, String)) = p._1
```

```scala
g(p) // works
g((1, "banana")) // works
```

## 1.3 Recursion

Recursion will be our main devise for expressing unbounded computations. In the following, we study how recursive functions are evaluated. We will further see that there is a close connection between certain recursive functions and loops in imperative programs.

### 1.3.1 Evaluating Recursive Functions

Consider the following function which computes the sum of the integer values in the interval given by the parameters `a` and `b`.

```scala
def sum(a: Int, b: Int): Int = {
  if (a < b) a + sum(a + 1, b) else 0
}
```

How are calls to such functions evaluated? In general, we can think of the evaluation of a Scala expression as a process that rewrites expressions into simpler expressions. This rewriting process terminates when we obtain an expression that cannot be further simplified, e.g., an integer number. Expressions that cannot be simplified further are called *values*. Concretely, if we have a function call such as `sum(1 + 1, 0 + 2)`, we proceed as follows to compute a value using rewriting:

- First, we rewrite the call expression by rewriting the arguments of the call until they are reduced to values. In our example, this step yields the simplified call expression `sum(2, 2)`.  

Next, we replace the entire call expression by the body of the function. At the same time, we replace the formal parameters occurring in the function body by the actual arguments of the call. In our example, this step yields the expression

\[
\text{if } (2 < 2) 2 + \text{sum}(2 + 1, 2) \text{ else } 0
\]

Finally, we continue rewriting the function body recursively in the same manner until we obtain a value that cannot be simplified further. In our example, we finally obtain the value 0.

Here is how we compute the value of \(\text{sum}(1, 4)\) using rewriting:

\[
\begin{align*}
\text{sum}(1, 4) &\rightarrow \text{if } (1 < 4) 1 + \text{sum}(1 + 1, 4) \text{ else } 0 \\
&\rightarrow \text{if } (\text{true}) 1 + \text{sum}(1 + 1, 4) \text{ else } 0 \\
&\rightarrow 1 + \text{sum}(1 + 1, 4) \\
&\rightarrow 1 + \text{sum}(2, 4) \\
&\rightarrow 1 + (\text{if } (2 < 4) 2 + \text{sum}(2 + 1, 4) \text{ else } 0) \\
&\rightarrow 1 + (\text{if } (\text{true}) 2 + \text{sum}(2 + 1, 4) \text{ else } 0) \\
&\rightarrow 1 + (2 + \text{sum}(2 + 1, 4)) \\
&\rightarrow 1 + (2 + (\text{if } (3 < 4) 3 + \text{sum}(3 + 1, 4) \text{ else } 0)) \\
&\rightarrow 1 + (2 + (\text{if } (\text{true}) 3 + \text{sum}(3 + 1, 4) \text{ else } 0)) \\
&\rightarrow 1 + (2 + (3 + \text{sum}(3 + 1, 4))) \\
&\rightarrow 1 + (2 + (3 + (\text{if } (4 < 4) 4 + \text{sum}(4 + 1, 4) \text{ else } 0))) \\
&\rightarrow 1 + (2 + (3 + (\text{if } (\text{false}) 4 + \text{sum}(4 + 1, 4) \text{ else } 0))) \\
&\rightarrow 1 + (2 + (3 + 0)) \\
&\rightarrow 1 + (2 + 3) \\
&\rightarrow 1 + 5 \\
&\rightarrow 6
\end{align*}
\]

Termination. Does the rewriting process always terminate? Consider the following recursive function:

\[
def \text{loop}(x: \text{Int}): \text{Int} = \text{loop}(x)
\]

If we evaluate, e.g., the call \(\text{loop}(0)\), we obtain an infinite rewriting sequence:

\[
\text{loop}(0) \rightarrow \text{loop}(0) \rightarrow \text{loop}(0) \rightarrow \ldots
\]

In order to guarantee termination of a recursive function, we have to make sure that each recursive call makes progress according to some progress measure. For example, in the recursive call to the function \(\text{sum}\) in our example above, the difference \(b - a\) between the arguments decreases with every recursive call. This means that \(b - a\) will eventually reach 0 or become negative. At this point, we take the else branch in the body of \(\text{sum}\) and the evaluation terminates. For our non-terminating function \(\text{loop}\), it is impossible to find such a progress measure.
1.3.2 Tail Recursion

If we apply the function `sum` to larger intervals we observe the following:

```scala
scala> sum(1, 10000)
java.lang.StackOverflowError
...
```

The problem is that a call to a function requires the Scala runtime environment to allocate stack space that stores the arguments of the call and any intermediate results obtained during the evaluation of the function body in memory. For the function `sum`, the intermediate results of the evaluation must be kept on the stack until the final recursive call returns. We can see this nicely in the rewriting steps for the call `sum(1, 4)`. The length of the expression that we still need to simplify grows with each recursive call:

```
sum(1, 4)
  => ...
  => 1 + sum(2, 4)
  => ...
  => 1 + (2 + sum(2 + 1, 4))
  => ...
  => 1 + (2 + (3 + sum(2 + 1, 4)))
  => ...
  => 6
```

Only when the final call to `sum` has returned, can we simplify the entire expression to a value.

The stack space that is needed for evaluating a call `sum(a, b)` grows linearly with the recursion depth, which is given by the size of the interval `b - a`. Since the Scala runtime environment only reserves a relatively small amount of memory for the call stack, a call to `sum` for large interval sizes runs out of stack space. This is signaled by a `StackOverflowError` exception.

Can we implement the function `sum` so that it only requires constant space? To this end, consider the following imperative implementation of `sum`, which uses a `while` loop and mutable variables to perform the summation:

```scala
def sum(a: Int, b: Int): Int = {
  var acc = 0
  var i = a
  while (i < b) {
    acc = i + acc
    i = i + 1
  }
  acc
}
```

This implementation requires only constant space, since it involves only a single function call. Moreover, the execution of a single loop iteration for the summation does not allocate memory that persists across iterations. The intermediate
Recursion 21

results are stored in the variables \( i \) and \( \text{acc} \), which are reused in each iteration. Unfortunately, this implementation uses mutable variables, which makes it more difficult to reason about the correctness of the code. However, we can turn the imperative `while` loop into a recursive function by hoisting the loop counter \( i \) and accumulator \( \text{acc} \) to function parameters:

```python
def loop(acc: int, i: int, b: int): int = {
    if (i < b) loop(i + acc, i + 1, b) else acc
}

def sum(a: int, b: int): int = {
    loop(0, a, b)
}
```

Note how the function `loop` closely mimics the `while` loop in the imperative implementation without relying on mutable variables. We simply pass the new values that we obtain for the loop counter \( i \) and the accumulator \( \text{acc} \) to the recursive call.

The function `loop` has an important property: the recursive call to `loop` in the `then` branch of the conditional expression is the final computation that is performed before the function returns. That is, in the recursive case, the function directly returns the result of the recursive call. We refer to functions in which all recursive calls are of this form as tail-recursive functions. Contrast the new implementation of `sum` with our original implementation, which added \( a \) to the result of the recursive call and was therefore not tail-recursive. The tail recursive implementation has an interesting effect on our rewriting-based evaluation strategy:

\[
\text{sum}(1, 4) \rightarrow \text{loop}(0, 1, 4) \\
\rightarrow \text{if } (1 < 4) \text{ loop}(1 + 0, 1 + 1, 4) \text{ else } 0 \\
\rightarrow \text{if } (\text{true}) \text{ loop}(1 + 0, 1 + 1, 4) \text{ else } 0 \\
\rightarrow \text{loop}(1, 2, 4) \\
\rightarrow \text{if } (2 < 4) \text{ loop}(2 + 1, 2 + 1, 4) \text{ else } 1 \\
\rightarrow \text{if } (\text{true}) \text{ loop}(2 + 1, 2 + 1, 4) \text{ else } 1 \\
\rightarrow \text{loop}(3, 3, 4) \\
\rightarrow \text{if } (3 < 4) \text{ loop}(3 + 3, 3 + 1, 4) \text{ else } 3 \\
\rightarrow \text{if } (\text{true}) \text{ loop}(3 + 3, 3 + 1, 4) \text{ else } 3 \\
\rightarrow \text{loop}(6, 3, 4) \\
\rightarrow \text{if } (4 < 4) \text{ loop}(4 + 6, 4 + 1, 4) \text{ else } 6 \\
\rightarrow \text{if } (\text{false}) \text{ loop}(4 + 6, 4 + 1, 4) \text{ else } 6 \\
\rightarrow 6
\]

Observe that the size of the expressions that we obtain throughout the evaluation does not grow with the recursion depth. This is because the tail-recursive call to `loop` is the final computation that is performed in the body of `loop`, before the function returns.

For tail-recursive functions, the stack space that is allocated for the current call can be reused by the recursive call. In particular, the memory that is needed
to store the arguments of the current call can be reused to store the arguments of the recursive call. By reusing the current stack space, we effectively turn the recursive function back into an imperative loop. We refer to this optimization as tail call elimination. Most compilers for functional languages, including the Scala compiler, automatically eliminate tail calls. Thus, tail-recursive functions are guaranteed to execute in constant stack space. We can test this feature by rerunning the tail-recursive version of sum for large interval sizes:

```
scala> sumInts(1, 10000)
res0: Int = 49995000
```

This time the function terminates normally without throwing an exception.

With tail call elimination we get the best of both worlds: we obtain the efficiency of an imperative implementation and the simplicity of a functional implementation. When you want to write a tail-recursive function, it is often useful to first write the function using a while loop and then transform the loop into a tail-recursive function, as we have done above. Once you get more used to functional programming, you will find writing tail-recursive functions as natural as writing loops.

You may wonder whether non-tail-recursive functions should be avoided at all costs. This depends on the function. Often, tail-recursive functions are harder to understand. If you know that the recursion depth of your function will be small in practice, you may want to write the function without tail-recursion. If you are in doubt, you should value the clarity of your code higher than its efficiency. When you observe that your code is inefficient, you can still optimize it later.

1.4 Classes and Objects

In the previous sections, we have learned about the basic language features of Scala. In this section, we will learn how Scala programs are organized. Scala is an object-oriented language, so Scala programs are organized using classes and objects.

1.4.1 Classes, Fields, and Methods

Similar to Java, Scala allows you to define classes with fields and methods, which you can extend using inheritance, override, etc. Fortunately, Scala’s syntax for classes is much more lightweight than Java’s. For example, consider the following Java class which we can use to wrap pairs of integer values in a single object:
The class consists of:

- two fields called `first` and `second` of type `int` to store the two values;
- a constructor, which takes values to initialize the two fields;
- two getter methods to retrieve the two values (we follow good practice and declare all non-final fields as private so that their values cannot be modified without explicit method calls).

Here is how we can define the corresponding class in Scala (let us ignore for the moment that we can represent pairs much more easily using tuples):

```scala
class Pair(val first: Int, val second: Int)
```

There are some important differences between the Java and Scala version of the class `Pair`:

- In Scala, the class name is followed by a list of class parameters. These parameters serve two purposes:
  1. Parameters that are prefixed by a `val` or `var` keyword automatically create a field with the given name and type.
  2. The parameter list implicitly defines a constructor with a corresponding list of arguments. The values that are provided for arguments prefixed with `val` or `var` will be used to initialize the associated fields.
- The default visibility of classes, fields, and methods in Scala is public. Hence, we can access the values of the fields `first` and `second` directly and we do not need to define extra getter methods. Note that this makes sense because Scala discourages mutable state. In particular, we defined
the two fields as `vals`, so their values cannot be changed, once an instance of class `Pair` has been created. In Scala, you can leave out the braces around an empty class body, so `class C` is the same as `class C {}`.

We can create instances of class `Pair` and access their fields as usual:

```scala
scala> val p = new Pair(1,2)
p: Pair = Pair@1458e1cc
scala> p.first
res0: Int = 1
```

What if we do want to modify the values stored in a `Pair` object? In Java, we would do this by adding appropriate setter methods to the class:

```java
public class Pair {
    private int first;
    private int second;

    public void setFirst(int fst) {
        first = fst;
    }
    public void setSecond(int snd) {
        second = snd;
    }
}
```

In Scala, we could follow the same route: change all `vals` into `vars`, make them private, and add getter and setter methods. However, we want to avoid using `vars` as much as possible. The idiomatic solution in Scala is to make a copy of the entire object and change the appropriate value:

```scala
class Pair(val first: Int, val second: Int) {
    def setFirst(fst: Int): Pair = new Pair(fst, second)
    def setSecond(snd: Int): Pair = new Pair(first, snd)
}
```

### 1.4.2 Overriding Methods

Java allows us to override methods that are declared in super classes. Since method calls are dynamically dispatched at run-time, this feature allows us to modify the behavior of an object of the subclass when it is used in a context where an object of the super class is expected.

All Java classes extend the class `Object`. The class `Object` provides, among others, a method `toString`, which computes a textual representation of the object. In particular, the `toString` method can be used to pretty-print objects. By default, the textual representation of objects consists of the name of the
object’s class, followed by a unique object ID. We can modify the way objects of a specific class are printed, by overriding the toString method. In Java, this can be done as follows:

```java
public class Pair {
    private int first;
    private int second;

    ... public String toString() {
        return "Pair(" + first + "," + second + ")";
    }
}
```

In Scala, all classes extend the class scala.Any which also provides a method called toString. Scala’s class hierarchy is further subdivided into the classes scala.AnyVal and scala.AnyRef, which are directly derived from scala.Any. All instances of scala.AnyVal are immutable, whereas instances of scala.AnyRef may have mutable state. That is, scala.AnyRef corresponds to Java’s Object class.

If we want to override a method in a Scala class, we have to explicitly say so by using the override qualifier:

```scala
class Pair(val first: Int, val second: Int) {
    ... override def toString() = "Pair(" + first + "," + second + ")"
}
```

The pretty printer in the REPL will now use the new toString method to print Pair objects:

```scala
scala> val p = new Pair(1,2)
p: Pair = Pair(1, 2)
```

### 1.4.3 Singleton and Companion Objects

It is often useful to declare factory methods that simplify the construction of objects, which involve complex initialization code. In Java, we would declare such methods as static members of the corresponding class:

```java
public class Pair {
    ... public static Pair make(int fst, int snd) {
        return new Pair(fst, snd);
    }
}
```

We can now call Pair.make to create new Pair instances.
Scala does not support static methods as they violate the philosophy of object-oriented programming. Instead, it provides singleton objects. Singleton objects are declared just like classes, but using the keyword object instead of class. There exists exactly one instance of each object. Hence, object declarations cannot have parameter lists.

A singleton object whose name coincides with the name of another class c is called the companion object of c. Companion objects have access to all private members of instances of c. Consequently, a method or field that is defined in the companion object is equivalent to a static method/field of c in Java:

```scala
class Pair(val first: Int, val second: Int) {
  ...
}
object Pair {
  def make(fst: Int, snd: Int) = new Pair(fst, snd)
}
```

We can access members of companion objects just like static class members in Java:

```scala
scala> def p = Pair.make(3,4)
p: Pair = Pair(3, 4)
```

### 1.5 Algebraic Data Types

Algebraic data types and pattern matching are constructs that are commonly found in functional programming languages. They allow you to implement regular, non-encapsulated data structures (such as lists and trees) in a convenient fashion. We will make heavy use of this feature throughout this course.

#### 1.5.1 Case Classes

Suppose we want to implement a simple calculator program that takes arithmetic expressions such as

\[(3 + 6/2) * 5\]

as input and evaluates these expressions. This problem is quite similar to writing an interpreter for a programming language, except that the language that we are interpreting is much simpler.

One of the first question that we have to answer is: how do we represent expressions in our program? Our representation should allow us to easily implement common tasks such as pretty printing, evaluation, and simplification of expressions. In particular, the representation should make the precedence of operators in expressions explicit. E.g., consider the expression \(3 + 6/2\), then when we evaluate the expression, our representation should immediately tell us that we first have to divide 6 by 2 before we add 3. To achieve this, expressions
are represented as *abstract syntax trees*, or ASTs for short. For example, the abstract syntax tree of the expression \((3 + 6/2) \times 5\) can be visualized as follows:

```
      *
     /|
     / |
    3 / 6
   /   |
  5
```

Note that the AST tells us exactly how to evaluate the expression. We start at the root. At each node that we visit, we first recurse into the left subtree to evaluate the corresponding subtree. Then we do the same for the right subtree. Finally, we combine the results according to the operation labeling the current node. We will learn more about ASTs in Section 3.2.1. For now it suffices if you have an intuitive understanding what ASTs are.

Algebraic data types allow us to represent tree-like data structures such as ASTs. In Scala, algebraic data types are constructed using *case classes*. The following case classes define the ASTs of our arithmetic expressions:

```scala
abstract class Expr
/* Numbers such as 1, 2, etc. */
case class Num(num: Int) extends Expr
/* Expressions composed using binary operators */
case class BinOp(op: Op, left: Expr, right: Expr) extends Expr
/* Binary operators */
abstract class Op
case object Add extends Op /* + */
case object Sub extends Op /* - */
case object Mul extends Op /* * */
case object Div extends Op /* / */
```

The Scala compiler adds some convenient functionality to the case classes. First, it automatically generates companion objects with appropriate factory methods. These methods are particularly useful when you nest them to construct complex expressions:

```scala
scala> val e = BinOp(Add, Num(3), BinOp(Mul, Num(4), Num(5)))
e: BinOp = BinOp(Add, Num(3), BinOp(Mul, Num(4), Num(5)))
```

Second, the compiler adds natural implementations of the methods `toString`, `hashCode`, and `equals` to case classes. They will print, hash, and compare a whole tree consisting of the top-level case class instance and (recursively) all its arguments. In Scala, an expression of the form `x == y` always translates into a call of the form `x.equals(y)` (just like in Java). The overridden `equals` method therefore ensures that case class instances are always compared structurally. For example, we have:

```scala
scala> val e1 = BinOp(Add, Num(3), Num(4))
```
e1: BinOp = BinOp(Add, Num(3), Num(4))
scala> val e2 = BinOp(Add, Num(3), Num(4))
e2: BinOp = BinOp(Add, Num(3), Num(4))
scala> e1 == e2
res2: Boolean = true

Note that in the example above, e1 and e2 point to two different objects in memory. However, the two ASTs represented by these objects have exactly the same structure. Hence, e1 == e2 evaluates to true.

Next, all arguments in the parameter list of a case class implicitly get a val prefix, so they are maintained as fields:

scala> val n = e1.left
n: Num = Num(3)
scala> n.num
res3: Int = 3

Finally, the compiler adds a copy method to your case classes for making modified copies. This method is useful if you need to create a new case class object that is identical to another case class object except that some of its attributes are different:

scala> e1.copy(op = Sub)
res4: BinOp(Sub, Num(3), Num(4))

1.5.2 Pattern Matching

Suppose we want to implement an algorithm that simplifies expressions by recursively applying the following simplifications rules:

- $e + 0 \Rightarrow e$
- $e * 1 \Rightarrow e$
- $e * 0 \Rightarrow 0$

To identify whether a given expression matches one of the left-hand sides of the rules, we have to look at some of its subexpressions. E.g., to check whether an expression of the form $e_1 + e_2$ matches the left-hand side of the first rule, we have to look at the left subexpression $e_1$ to check whether $e_1 = 0$. Implementing this kind of pattern matching is quite tedious in many languages (including Java). Fortunately, the Scala language has inbuilt support for pattern matching that works hand-in-hand with case classes.

Let us first reformulate the three simplification rules in terms of our case class expressions:

- $\text{BinOp}(\text{Add}, e, \text{Num}(0)) \Rightarrow e$
- $\text{BinOp}(\text{Mul}, e, \text{Num}(1)) \Rightarrow e$
- $\text{BinOp}(\text{Mul}, e, \text{Num}(0)) \Rightarrow \text{Num}(0)$
Using pattern matching, these rules almost directly give us the implementation of the following function `simplifyTop`, which applies the rules at the top-level of the given expression `e`:

```python
def simplifyTop(e: Expr) = e match {
  case BinOp(Add, e1, Num(0)) => e1
  case BinOp(Mul, e1, Num(1)) => e1
  case BinOp(Mul, _, Num(0)) => Num(0)
  case _ => e
}
```

The body of `simplifyTop` is a *match expression*. A match expression consists of a selector, in this case `e`, followed by the keyword `match`, followed by a sequence of match alternatives enclosed in braces.

Each alternative starts with the keyword `case`, followed by a pattern, followed by an expression that is evaluated if the pattern matches the selector. The pattern and expression are separated by an arrow symbol `=>`.

A match expression is evaluated by checking whether the selector matches one of the patterns in the alternatives. The patterns are tried in the order in which they appear in the program. The first pattern that matches is selected and the part following the arrow is evaluated. The result of the entire match expression is the result of the expression in the selected alternative.

Here is an example of a recursive function that uses pattern matching to pretty print arithmetic expressions:

```python
def pretty(e: Expr): String = e match {
  case BinOp(op, e1, e2) =>
    val op_str = op match {
      case Plus => "+"
      case Minus => "-"
      case Mult => "*"
      case Div => "/"
    }
    "(" + pretty(e1) + op_str + pretty(e2) + ")"
  case Num(n) => n.toString()
}
```

```scala```
scala> val e = BinOp(Add, BinOp(Mul, Num(3), Num(4)), Num(1))
e: BinOp = BinOp(Add, BinOp(Mul, Num(3), Num(4)), Num(1))
scala> pretty(e)
res0: String = ((3 * 4) + 1)
```

There are different types of patterns. The most important types are:

- **Constant patterns**: A constant pattern such as `0` matches values that are equal to the constant (with respect to `=`).

- **Variable patterns**: A variable pattern such as `e1` matches every value. Here, `e1` is a variable that is bound in the pattern. The variable refers to the matched value in the right-hand side of the case clause.

- **Wildcard patterns**: A wildcard pattern `_` also matches every value, but it does not introduce a variable that refers to the matched value.

- **Constructor patterns**: A constructor pattern such as `BinOp(Add, e, Num(0))` matches all values of type `BinOp` whose first argument matches `Add`, whose second argument matches `e`, and whose second argument matches `Num(0)`. Note that the arguments to the constructor `BinOp` are themselves patterns. This allows you to write deep patterns that match complex case class values using a concise notation.

### 1.5.3 Binding Names in Patterns

Sometimes we want to match a subexpression against a specific pattern and also bind the matched expression to a name. This is useful when we want to reuse a matched subexpression in the right-hand side of the match alternative. For example, in the third simplification rule of `simplifyTop` we are returning `Num(0)`, which is also the second subexpression of the matched expression `e`. Instead of creating a new expression, `Num(0)` on the right-hand side of the rule, we can also directly return the second subexpression of `e`. We can do this by binding a name to that subexpression in the pattern using the operator `@` as follows:

```scala
def simplifyTop(e: Expr) = e match {
  case BinOp(Add, e1, Num(0)) => e1
  case BinOp(Mul, e1, Num(1)) => e1
  case BinOp(Mul, _, e2 @ Num(0)) => e2
  case _ => e
}
```

Note that the pattern in third match alternative now binds the name `e2` to the value matched by the pattern `Num(0)`. This value is then returned on the right-hand side of the rule.

### 1.5.4 Pattern Guards

Suppose we want to extend our expression simplifier so that it additionally implements the following simplification rule: \( e + e \Rightarrow 2 \times e \)

If we directly translate the rule to a corresponding match alternative, we obtain the following implementation of `simplifyTop`:

```scala
def simplifyTop(e: Expr) = e match {
  ... 
  case BinOp(Add, e1, e1) => BinOp(Mul, Num(2), e1)
  case _ => e
}
```

Unfortunately, the compiler will reject this function because we use the name `e1` twice within the same pattern. We can solve this problem by renaming the
second occurrence of the variable and enforcing their equality using a pattern guard:

```scala
def simplifyTop(e: Expr) = e match {
  ...
  case BinOp(Add, e1, e2) if e1 == e2 =>
    BinOp(Mul, Num(2), e2)
  case _ => e
}
```

In general, a pattern guard can be an arbitrary Boolean expression over the names that are in the scope of the match alternative. The pattern guard is appended to the pattern of a match alternative using the keyword `if`.

### 1.5.5 Sealed Classes

Consider the following function `simplifyAll` that applies our simplification rules recursively to the given expression:

```scala
def simplifyAll(e: Expr) = e match {
  case BinOp(Add, e1, Num(0)) => simplifyAll(e1)
  case BinOp(Mul, e1, Num(1)) => simplifyAll(e1)
  case BinOp(Mul, _, e2 @ Num(0)) => e2
  case BinOp(Add, e1, e2) if e1 == e2 =>
    BinOp(Mul, Num(2), simplifyAll(e2))
  case BinOp(op, e1, e2) =>
    BinOp(op, simplifyAll(e1), simplifyAll(e2))
  ...
}
```

Observe that in this function, the pattern alternatives are no longer exhaustive. That is, there exist values `e` that are not matched by any of the match alternatives, e.g., the value `Num(0)`. If `simplifyAll` is called with `Num(0)`, it will throw a runtime exception.

We can fix this code by adding an explicit match alternative for the `Num` constructor:

```scala
def simplifyAll(e: Expr) = e match {
  case BinOp(Add, e1, Num(0)) => simplifyAll(e1)
  case BinOp(Mul, e1, Num(1)) => simplifyAll(e1)
  case BinOp(Mul, _, e2 @ Num(0)) => e2
  case BinOp(Add, e1, e2) if e1 == e2 =>
    BinOp(Mul, Num(2), simplifyAll(e2))
  case BinOp(op, e1, e2) =>
    BinOp(op, simplifyAll(e1), simplifyAll(e2))
  case Num(_) => this
}
```

Is this code safe or are there still values for `e` that are not matched by any of the match alternatives? The answer is: it depends. Scala allows us to further
extend the abstract class Expr by new case classes. In particular, we could define a new case class

```scala
case class Var(name: String) extends Expr
```

in a different source file, but as part of the same package as the abstract class Expr. Now calling simplifyAll on, say, `Var("x")` would again yield a run-time exception.

We can prevent the extension of a class outside of the source file in which it is defined by declaring the class as `sealed`. For our expression type Expr this looks as follows:

```scala
sealed abstract class Expr
case class Num(num: Int) extendsExpr
case class BinOp(op: Op, left: Expr, right: Expr) extends Expr
```

```scala
sealed abstract class Op
case object Add extends Op
case object Mul extends Op
case object Div extends Op
case object Sub extends Op
```

Now, the code of simplifyAll guarantees that for every call `simplifyAll(e)`, one of the patterns in the match expression in `simplifyAll` will always match `e`. In fact, one of the nice features of sealed classes is that the compiler checks that pattern matching on sealed classes is exhaustive and warns us if we forgot to handle a particular match case.

Unfortunately, these exhaustiveness checks can sometimes produce spurious warnings. For example, suppose we have a function that is meant to pretty print number expressions, but not other expressions which have not yet been reduced:

```scala
def prettyNumber(e: Expr): String =
  e match {
    case Num(num) => num.toString()
  }
```

Further suppose that we know that our program ensures that `prettyNumber` is never called on a `BinOp` expression. Yet, the compiler still complains about the non-exhaustive pattern matching. We can suppress this warning by declaring `e` as `unchecked`:

```scala
def prettyNumber(e: Expr): String =
  (e: @unchecked) match {
    case Num(num) => num.toString()
  }
```

While `@unchecked` notations are sometimes necessary to suppress spurious warnings, you should be very careful about introducing them in your code. In most cases, the compiler generated warnings indicate actual problems in your code that need your attention.
1.5.6 Option Types

Suppose we want to write a function that evaluates arithmetic expressions to `Int` values. One question is: How should we deal with undefined operations such as division by zero:

\[ \text{BinOp(Div, e, Num(0)) => ?} \]

In Java, we would typically go for one of the following two solutions:

- throw an exception such as `ArithmeticException`;
- return `null` to indicate that the intended operation does not yield a valid result.

Both approaches have advantages and drawbacks.

Exceptions are a good solution if the undefined operation is indeed exceptional behavior that should, e.g., abort the program. In this case, we ensure that a computation that returns normally always yields a valid result. However, if the undefined operation commonly occurs in computations, we will have to catch the exception and handle it appropriately. This has two disadvantages. First, the exception mechanism is relatively expensive and should only be used in truly exceptional situations. Second, the exception handlers will clutter the code and the non-structured control-flow of thrown exceptions make it more difficult to understand what the program is doing.

If we return `null`, we avoid the two disadvantages of exceptions: the computation always returns normally, and there is no computational overhead such as recording the stack-trace to the point where the exception was thrown. However, `null` values introduce their own problems. Since `null` can have an arbitrary type, the type checker of the compiler will give us much weaker static correctness guarantees for our code. In particular, it will be unable to statically detect unintended accesses to the return value in cases where the return value is invalid (hello `NullPointerExceptions`).

In languages that support pattern matching, there is a common idiom that avoids the problem of introducing `null` values: option types.

The option type is an algebraic data type with two variants: `Some(v)` to indicate that a computation returned a proper result \( v \), and `None` to indicate that the intended operation was undefined and has no proper result.

In Scala, we can define an option type for `Int` values using `case` classes as follows:

```
sealed abstract class IntOption
case class Some(value: Int) extends IntOption
case object None extends IntOption
```

We can now use the option type similarly to `null` values in Java:

```
def div(x: Int, y: Int): IntOption =
  if (y == 0) None else Some(x / y)
```
Unlike in Java, where the static type checker is unable to distinguish a `null` value from a genuine result of a computation, the Scala type checker will force us to explicitly unwrap the `Int` value embedded in an `IntOption` before we can access it. Using pattern matching, we can do this conveniently. For example, suppose we want to convert the result of `div` to a double precision floating point number. By using pattern matching on the return value of `div`, we can recover from some of the cases in which

```scala
def divToDouble(x: Int, y: Int): Double =
  div(x, y) match {
    case Some(x) => x
    case None =>
      if (x < 0) Double.NegativeInfinity
      else if (x > 0) Double.PositiveInfinity
      else Double.NaN
  }
```

Since option types are so useful, Scala already provides a generic option type, called `Option`, in its standard library. Using the predefined type `Option` we can write the function `div` like this:

```scala
def div(x: Int, y: Int): Option[Int] =
  if (y == 0) None else Some(x / y)
```
Chapter 2

Foundations

Recursion and induction will be our main tools for formalizing programming languages. In this chapter, we study the mathematical foundations of these two closely related concepts. We will then apply these concepts to understand a ubiquitous recursive data structure: lists. We will then see that this formal approach allows us to prove mathematical properties of programs.

2.1 Structural Recursion and Induction

Recursion is a constructive technique for describing infinite sets (and thus infinite functions on these sets). Induction is a technique for proving properties about recursively defined sets. There exist different variants of recursion and induction. We are interested in the simplest form of these concepts which we refer to as structural recursion and structural induction, respectively.

2.1.1 Structurally Recursive Definitions

We explain all these concept using a very simple example. To this end, we define a set \( N \) that behaves just like (or mathematically speaking, is isomorphic to) the natural numbers \( \mathbb{N} \). We will represent the natural numbers as pure sets. Once we have defined \( N \), we will build structurally recursive functions on \( N \) that correspond to addition and multiplication.

We represent the natural numbers as follows using only sets: we start by 0, which we represent as the empty set \( \emptyset \). Each larger natural number \( n + 1 \) is constructed from its predecessor \( n \) by wrapping it in another set:

\[
\begin{align*}
0 & \quad \emptyset \\
1 & \quad \{\emptyset\} \\
2 & \quad \{\{\emptyset\}\} \\
\ldots
\end{align*}
\]
For $N$ we chose exactly those sets that represent natural numbers following the above convention.

We now show how we can describe the set $N$ using structural recursion, without referring to the natural numbers $\mathbb{N}$. We do this by providing recursive construction rules for the elements of $N$:

1. The empty set is an element of $N$.
2. If $x$ is an element of $N$, then the set $\{x\}$ is an element of $N$.
3. $N$ only contains elements that can be constructed using rules 1 and 2.

We can present these construction rules using the following inference rules:

\[
\begin{align*}
\emptyset &\in N \\
\text{Rule 1} \\
\{x\} &\in N, \quad x \in N \\
\text{Rule 2}
\end{align*}
\]

In general, inference rules take the form

\[
\frac{P_1 \ldots P_n}{C}
\]

Such a rule states that if the properties $P_1, \ldots, P_n$ hold, then also $C$ holds. The properties $P_i$ are called the premises of the rule, and the property $C$ the conclusion of the rule. If a rule has no premise, then the conclusion always holds. Such rules are also called axioms.

Using inference rule 1, we can construct the representation of the number 0. Using rule 2, we can construct the representation of the number $n+1$, given the representation of the number $n$. For the representation of the number 3 we need three construction steps:

1. $\emptyset$ with rule 1
2. $\{\emptyset\}$ with rule 2
3. $\{\{\emptyset\}\}$ with rule 2
4. $\{\{\{\emptyset\}\}\}$ with rule 2

The recursive definition of $N$ is to be understood such that $N$ contains exactly those objects that can be constructed by the inference rules in a finite number of steps.

Alternatively, we can describe the construction rules for the elements of $N$ using a recursive equation:

\[N = \{\emptyset\} \cup \{\{x\} \mid x \in N\}\]

That is, $N$ is the smallest set (with respect to subset inclusion) that satisfies the equation.

The recursive definition of $N$ has two important properties that make it a structurally recursive definition:
1. Every object in $N$ can be constructed only with exactly one rule. For example, $\emptyset$ can only be constructed with the first rule and $\{\emptyset\}$ only with the second rule.

2. The recursive rule constructs from an object $x \in N$ a larger object $\{x\} \in N$ that contains $x$ as a proper subobject.

Scala types that are defined using case class declarations can be viewed as structurally recursive definitions of sets. For example the following declarations correspond to our definition of $N$:

```scala
sealed abstract class n
case object emptyset extends n OO rule 1
case class setHelem: nI extends n OO rule R
```

Note that the `sealed` keyword encodes our third construction rule for the set $N$, which states that elements that cannot be constructed using the rules 1 and 2 do not belong to $N$.

### 2.1.2 Recursive Definitions of Functions

We now want to define a function $D : N \rightarrow N$ that maps the elements of $N$ to the natural numbers they represent. This is done as follows:

$$
D : N \rightarrow \mathbb{N} \\
D(\emptyset) = 0 \\
D(\{x\}) = 1 + D(x)
$$

This definition has two important properties that make it a structurally recursive definition:

1. For each defining rule of $N$, there is a corresponding defining rule for $D$. This implies that for every element of $N$ exactly one rule for $D$ applies.

2. Recursive applications of $D$ only apply to proper subobjects of its argument (in the second rule for $D$, the recursive application of $D$ for the argument $\{x\}$ is on the proper subobject $x$.)

Together, these properties guarantee that the rules for $D$ define a total function. We can also view the above definition as a blueprint for the definition of a function in Scala on our algebraic data type $N$:

```scala
def D(y: N): Int =
y match {
  case Emptyset => 0
  case Set(x) => 1 + D(x)
}
```
From the fact that the original definition is structurally recursive, it follows that \( D \) terminates normally for all input values \( y \).

Next, we use structural recursion to define a two-valued function \( + : N \times N \to N \) that corresponds to addition on natural numbers:

\[
+ : N \times N \to N \\
\emptyset + y = y \\
\{x\} + y = \{x + y\}
\]

Note that the second defining rule of \( D \) and \( + \) each determines how the value of the function for larger arguments is constructed from values of the function for smaller arguments. In the case of \( + \), the recursion goes over the first argument.

### 2.1.3 Structural Induction

In order to prove that all \( x \in N \) satisfy a given property \( A \), we can proceed as follows:

1. Prove that \( \emptyset \) satisfies \( A \).
2. Prove that for all \( x \in N \) that satisfy \( A \), \( \{x\} \) satisfies \( A \).

We call the proof rule that we just formulated the *induction rule* for \( N \). We can write this rule more compactly as an inference rule:

\[
\begin{array}{c}
A(\emptyset) \\
\forall x \in N : A(x) \Rightarrow A(\{x\})
\end{array} \quad \forall x \in N : A(x)
\]

Notice that the premises of this inference rule

1. \( A(\emptyset) \)
2. \( \forall x \in N : A(x) \Rightarrow A(\{x\}) \)

are derived directly from the defining rules of \( N \).

You should realize the correctness of the induction rule for \( N \). Perhaps the following explanation helps you if you have trouble understanding the rule. Let \( A \) be some property for which the premises (1) and (2) of the induction rule hold. We validate that from these premises follows the validity of

\[
A(\{\{\emptyset\}\})
\]

First, from premise (1) follows:

\[
A(\emptyset)
\]

From this fact and premise (2) we conclude:

\[
A(\{\emptyset\})
\]
Applying premise (2) twice more we obtain the property we wanted to show:

\[ A(\{\{\emptyset\}\}) \]

The trick lies in premise (2), which states that for any \( x \in N \), if \( A(x) \) holds, then also \( A(\{x\}) \) holds.

Next, we give a concrete example that shows how to use the induction rule. We want to prove that the following property holds:

\[ \forall x \in N : \forall y \in N : D(x + y) = D(x) + D(y) \]

To this end, we define the property

\[ A(x) \overset{\text{def}}{=} \forall y \in N : D(x + y) = D(x) + D(y) \]

and then prove that for all \( x \in N \), \( A(x) \) holds.

The induction rule tells us that it is sufficient to prove the following two properties (these are the premises of the induction rule instantiated with the concrete \( A \) that we defined above):

1. \( \forall y \in N : D(\emptyset + y) = D(\emptyset) + D(y) \)
2. \( \forall x \in N : (\forall y \in N : D(x + y) = D(x) + D(y)) \Rightarrow (\forall y \in N : D(\{x\} + y) = D(\{x\}) + D(y)) \)

It is not difficult to prove these properties.

To minimize the amount of writing we have to do and to improve clarity, we follow a specific pattern when we write induction proofs. We demonstrate this pattern in the proof below.

**Lemma 2.1.** \( \forall x \in N : \forall y \in N : D(x + y) = D(x) + D(y) \)

**Proof.** By structural induction over \( x \in N \):

Let \( x = \emptyset \) and \( y \in N \). Then

\[ D(x + y) = D(\emptyset + y) = D(y) \]

Definition of +

\[ = 0 + D(y) = D(\emptyset) + D(y) \]

Definition of \( D \)

\[ = D(x) + D(y) \]


Let \( x = \{x'\} \) and \( y \in N \). Then

\[ D(x + y) = D(\{x'\} + y) = D(\{x' + y\}) \]

Definition of +

\[ = 1 + D(x' + y) = 1 + D(x') + D(y) \]

Definition of \( D \)

\[ = D(\{x'\}) + D(y) \]

Induction hypothesis

\[ = D(x) + D(y) \]

Definition of \( D \)
For every set that is defined by structural recursion there is an induction rule that is derived from the definition of the set.

2.1.4 Well-founded Induction (optional)

Recursion and induction appear in many variants. The common essence of all these variants can be formulated using the notion of well-founded relations.

Let $X$ be a set. A binary relation $\succ \subseteq X \times X$ is called well-founded, if there exists no infinite sequence $x_1, x_2, x_3, \ldots$ of elements in $X$ such that $x_1 \succ x_2 \succ x_3 \succ \cdots$. We call a pair $(X, \succ)$ consisting of a set $X$ and a well-founded relation $\succ$ on $X$ a well-founded set.

Let $(X, \succ)$ be a well-founded set. With the notation $x \succ y$ we intuitively mean that $x$ is in some sense larger than $y$. Well-foundedness then means that there are no infinite descending chains $x_1 \succ x_2 \succ x_3 \succ \cdots$ of smaller and smaller elements.

As an example, let us reconsider the set $N$. Obviously the relation $x \succ y$ defined as $\iff x = \{y\} \land x \in N$ is a well-founded relation on $N$. The reflexive and transitive closure of $\succ$ corresponds to the canonical ordering on natural numbers $\geq \subseteq N \times N$.

Let $(X, \succ)$ be a well-founded set and $M \subseteq X$. An element $x \in M$ is called minimal element of $M$ if there exists no $y \in M$ such that $x \succ y$.

**Lemma 2.2.** Let $(X, \succ)$ be a well-founded set. Then every nonempty subset of $X$ has at least one minimal element.

**Proof.** By contradiction. Let $M$ be a nonempty subset of $X$ that has no minimal element. Then there exists for every $x \in M$ some $y \in M$ such that $x \succ y$. Since $M$ contains at least one element, we can construct an infinite descending chain of elements in $M$, and thus in $X$. It follows that $\succ$ is not well-founded. Contradiction. \[\square\]

Let $(X, \succ)$ be a well-founded set and $A(x)$ a property for $x \in X$. We call the following inference rule well-founded induction\[\square\] for $X$, $\succ$, and $A$:

$$\forall x \in X : (\forall y \in X : x \succ y \Rightarrow A(y)) \Rightarrow A(x)$$

$$\forall x \in X : A(x)$$

The following theorem states the correctness of this rule.

**Theorem 2.3.** Let $(X, \succ)$ be a well-founded set and $A(x)$ a property of $x \in X$. If

$$\forall x \in X : (\forall y \in X : x \succ y \Rightarrow A(y)) \Rightarrow A(x)$$

then for all $x \in X$, $A(x)$ holds.

\[\text{Sometimes, well-founded induction is also called Noetherian induction after the mathematician Emmy Noether.}\]
Proof. By contradiction. Let $M$ be the subset of $X$ that contains all elements for which $A$ does not hold. We assume that $M$ is nonempty. Then we can choose a minimal element $x_0$ in $M$ according to Lemma 2.2. Since $M$ contains all elements of $X$ for which $A$ does not hold, it follows that

$$\forall y \in X : x_0 \succ y \Rightarrow A(y)$$

Then $A(x_0)$ follows from the premise of the induction rule. Contradiction. \qed

2.2 Lists

Lists are one of the most important data structures in functional programming languages. A list is a sequence of data values of some common element type, e.g., a sequence of integer numbers 3, 6, 1, 2. Unlike linked lists, which you have studied in your Data Structures course, lists in functional programming languages are immutable. As with other immutable data structures, immutable lists have the advantage that their representation in memory can be shared across different list instances. For example, the two lists 1, 4, 3 and 5, 2, 4, 3 can share their common sublist 4, 3. This feature enables immutable lists to be used for space-efficient, high-level implementations of algorithms if the data structure is used correctly. In this section, we will define immutable lists using structural recursion and see that this definition corresponds to Scala’s list data type.

2.2.1 Defining Lists using Structural Recursion

Mathematically, we can represent lists of integer numbers as nested tuples. For example, the empty list is represented by the empty tuple $\langle \rangle$, and the list containing the sequence of numbers 5, 2, and 3 is represented by the tuple $\langle 5, \langle 2, \langle 3, \langle \rangle \rangle \rangle \rangle$. The following structurally recursive definition formalizes this idea:

\[
\langle \rangle \in \text{List} \quad \text{hd} \in \mathbb{Z} \quad \text{tl} \in \text{List} \\
\langle \text{hd}, \text{tl} \rangle \in \text{List}
\]

For a non-empty list $\ell$ of the form $\langle \text{hd}, \text{tl} \rangle$, we refer to the integer number $\text{hd}$ as the head of $\ell$, and we call the remaining list $\text{tl}$ the tail of $\ell$. For example, the head of the list $\langle 4, \langle 2, \langle \rangle \rangle \rangle$ is 4 and its tail is $\langle 2, \langle \rangle \rangle$. We also refer to a non-empty list as a cons cell. To improve readability, we denote the empty list $\langle \rangle$ by $\text{nil}$.

In Scala, we can define lists of integers using an algebraic data type:

```scala
sealed abstract class List
case object Nil extends List
case class Cons(hd: Int, tl: List) extends List
```

The Scala standard library actually provides a generic list type that is parametric in its element type. The definition of this type is similar to the one that we give here. We will study Scala’s list type more closely in Section 5.5.3.
Here Nil represents the empty list and a cons cell \(<\text{hd}, \text{tl}\)> is represented by \text{Cons}(\text{hd}, \text{tl}). Note the close resemblance between the mathematical definition and the Scala definition of lists.

Here is how we construct a Scala list containing the values 1, 4, 2:

```scala
scala> val l = Cons(1, Cons(4, Cons(2, Nil)))
l: List = Cons(1, Cons(4, Cons(2, Nil)))
```

We can also use pattern matching to deconstruct lists into their components:

```scala
scala> val Cons(h, t) = l
h: Int = 1
t: List = Cons(4, Cons(2, Nil))
```

```scala
scala> l match {
  case Nil => println("l is empty")
  case Cons(h, t) => println(s"l's head is \$h.")
}
l's head is 1.
```

### 2.2.2 Functions on Lists

Using structural recursion we can now define simple functions on lists. For example, the following function computes the length of a given list:

\[
\text{length} : \text{List} \to \mathbb{N} \\
\text{length}(\text{nil}) = 0 \\
\text{length}(\langle \text{hd}, \text{tl} \rangle) = 1 + \text{length}(\text{tl})
\]

The next function is more interesting, it takes two lists \(\ell_1\) and \(\ell_2\) and creates a new list by concatenating \(\ell_1\) and \(\ell_2\).

\[
\text{append} : \text{List} \times \text{List} \to \text{List} \\
\text{append}(\text{nil}, \ell_2) = \ell_2 \\
\text{append}(\langle \text{hd}, \ell_1 \rangle, \ell_2) = \langle \text{hd}, \text{append}(\ell_1, \ell_2) \rangle
\]

For example, for \(\ell_1 = \langle 4, \langle 6, \langle 1, \text{nil} \rangle \rangle \rangle\) and \(\ell_2 = \langle 5, \langle 1, \text{nil} \rangle \rangle\) we get

\[
\text{append}(\ell_1, \ell_2) = \langle 4, \langle 6, \langle 1, \langle 5, \langle 1, \text{nil} \rangle \rangle \rangle \rangle \rangle.
\]

Finally, using \text{append} we can define a function \text{reverse} that takes a list \(\ell\) and creates a new list that contains the elements of \(\ell\) in reverse order:

\[
\text{reverse} : \text{List} \to \text{List} \\
\text{reverse}(\text{nil}) = \text{nil} \\
\text{reverse}(\langle \text{hd}, \ell_1 \rangle) = \text{append}(\text{reverse}(\ell_1), \langle \text{hd}, \text{nil} \rangle)
\]
For example, we have $\text{reverse}([4, (2, \text{nil})]) = (2, (4, \text{nil}))$.

Note that the definition of $\text{reverse}$ is still structurally recursive since in the recursive case $\text{reverse}$ is only applied to the tail $tl$ of the input list.

The mathematical definitions of $\text{length}$, $\text{append}$, and $\text{reverse}$ directly translate to corresponding Scala functions:

```scala
def length(l: List): Int = l match {
  case Nil => 0
  case Cons(hd, tl) => 1 + length(tl)
}

def append(l1: List, l2: List): List = l1 match {
  case Nil => l2
  case Cons(hd, tl) => Cons(hd, append(tl, l2))
}

def reverse(l: List): List = l match {
  case Nil => Nil
  case Cons(hd, tl) => append(reverse(tl), Cons(hd, Nil))
}
```

Unfortunately, these Scala functions are not very efficient. For example, the running time of $\text{reverse}$ is quadratic in the length of the list $l$. Moreover, this function is not tail-recursive and hence requires linear space in the length of $l$. The implementations of $\text{length}$ and $\text{append}$ are also not tail-recursive. While we typically do not care about computational efficiency when we define mathematical functions, we do care about it when we write programs. To obtain efficient implementations we would rather implement these functions tail-recursively. For example, we can rewrite $\text{reverse}$ so that it runs in linear time and constant space:

```scala
def reverse2(l: List): List = {
  def rev(l: List, acc: List): List = l match {
    case Nil => acc
    case Cons(h, t) => rev(t, Cons(h, acc))
  }
  rev(l, Nil)
}
```

**Exercise 2.1.** Give the mathematical definition of the tail-recursive $\text{reverse2}$ function. Call this function $\text{reverse2}$. Hint: to define $\text{reverse2}$, first define an auxiliary function $\text{rev} : \text{List} \times \text{List} \to \text{List}$.

**Exercise 2.2.** Define a tail-recursive Scala versions of the functions $\text{length}$ and $\text{append}$. Hint: use $\text{reverse2}$ in the definition of $\text{append}$.
2.2.3 Proving Properties of Functions on Lists

Lists are defined by structural recursion. Hence, we can use structural induction to prove properties about functions (and programs) that operate on lists. As an example, the following proposition states that the length of a list obtained by appending two lists \( \ell_1 \) and \( \ell_2 \) is equal to the sum of the lengths of \( \ell_1 \) and \( \ell_2 \).

**Proposition 2.4.** For all \( \ell_1, \ell_2 \in \text{List} \) the following property holds

\[
\text{length}(\text{append}(\ell_1, \ell_2)) = \text{length}(\ell_1) + \text{length}(\ell_2).
\]

**Proof.** Since \( \text{append} \) is defined by structural recursion on its first argument, the proof proceeds by structural induction on \( \ell_1 \): Let \( \ell_1, \ell_2 \in \text{List} \) such that \( \ell_1 = \text{nil} \). Then

\[
\text{length}(\text{append}(\ell_1, \ell_2)) = \text{length}(\text{append}(\text{nil}, \ell_2))
\]

\[
= \text{length}(\ell_2) \quad \text{Def. of append}
\]

\[
= 0 + \text{length}(\ell_2) \quad 0 \text{ is neutral element}
\]

\[
= \text{length}(\text{nil}) + \text{length}(\ell_2) \quad \text{Def. of length}
\]

\[
= \text{length}(\ell_1) + \text{length}(\ell_2)
\]

Let \( \ell_1, \ell_2 \in \text{List} \) such that \( \ell_1 = (\text{hd}, \text{tl}) \). Then

\[
\text{length}(\text{append}(\ell_1, \ell_2)) = \text{length}(\text{append}((\text{hd}, \text{tl}), \ell_2))
\]

\[
= \text{length}(\langle \text{hd}, \text{append}(\text{tl}, \ell_2) \rangle) \quad \text{Def. of append}
\]

\[
= 1 + \text{length}(\text{append}(\text{tl}, \ell_2)) \quad \text{Def. of length}
\]

\[
= 1 + (\text{length}(\text{tl}) + \text{length}(\ell_2)) \quad \text{Induction hypothesis}
\]

\[
= (1 + \text{length}(\text{tl})) + \text{length}(\ell_2) \quad \text{Associativity of +}
\]

\[
= \text{length}((\text{hd}, \text{tl})) + \text{length}(\ell_2) \quad \text{Def. of length}
\]

\[
= \text{length}(\ell_1) + \text{length}(\ell_2)
\]

Exercise 2.3. For the function \( \text{reverse}_2 \) that you defined in Exercise 2.1, use structural induction to prove that it computes the same function as \( \text{reverse} \). That is, for all \( \ell \in \text{List} \), \( \text{reverse}(\ell) = \text{reverse}_2(\ell) \).
Chapter 3

Syntax

When we describe programming languages formally, we distinguish between the syntax and the semantics of a language. The syntax describes the structure of a program, whereas the semantics describes its meaning (i.e., what the program computes). In order to understand programming languages, it is important to keep these two concepts separated. In particular, we use different mathematical objects to represent the syntax and semantics of programs.

When we talk about the syntax of a programming language, we further distinguish between its concrete syntax and its abstract syntax. The concrete syntax defines which sequences of characters represent programs. The abstract syntax describes the structure of a program as an abstract syntax tree. Such a tree abstracts from some of the specifics of the concrete syntax, such as parenthesis in expressions, semicolons after statements, etc. The abstract syntax also makes the precedence and associativity of operators explicit.

3.1 Concrete Syntax (optional)

The parsing problem which is to convert sequences of characters (i.e., concrete syntax) into abstract syntax trees is a well-understood problem. You can learn more about this topic in a compiler construction course. In this course, we will therefore work directly with abstract syntax. Nevertheless, it is useful to have a basic understanding of how the concrete syntax of a programming language can be formalized and what the typical problems are when writing parsers for programming languages.

3.1.1 Formal Languages

In computer science, we formally describe languages as sets of words. Each word is a finite sequence of symbols drawn from a set that we call the alphabet of the language. For example, we can describe an arithmetic expression “3 + 5 * 8” as a word that is given by the sequence of symbols ‘3’, ‘+’, ‘5’, ‘*’, and ‘8’.
To describe languages in a compact form, we use grammars. A grammar is given by a set of rules, called productions. Productions tell us how the words of the language can be constructed from the symbols in the alphabet. For example, the following grammar describes the language of all arithmetic expressions:

\[
\begin{align*}
E & \rightarrow EOE \\
E & \rightarrow (E) \\
E & \rightarrow x \quad \text{where } x \in \mathbb{Z} \\
O & \rightarrow + \\
O & \rightarrow - \\
O & \rightarrow * \\
O & \rightarrow /
\end{align*}
\]

Note that the third rule actually stands for an infinite set of productions (one for each \( x \in \mathbb{Z} \)):

\[
\begin{align*}
\ldots\\
E & \rightarrow -2 \\
E & \rightarrow -1 \\
E & \rightarrow 0 \\
E & \rightarrow 1 \\
E & \rightarrow 2 \\
\ldots
\end{align*}
\]

We call the uppercase symbols that occur on the left-hand sides of productions, such as \( E \) and \( O \), nonterminal symbols. The remaining symbols that are drawn from the alphabet of the language such as + and 3 are called terminal symbols.

Each grammar has an associated (nonterminal) starting symbol. In our example grammar, this is the symbol \( E \). Starting from this symbol we apply the productions one by one until we obtain a word that consists only of terminal symbols. In each step, we pick one nonterminal in the current working word, choose a production in which this nonterminal occurs on the left-hand side, and replace the chosen nonterminal in the working word by the right-hand side of the chosen production.

Using the above productions, we can derive words such as

\[
\begin{align*}
1 \\
3 + 5 * 8 \\
-14/(42 + (0 - 1))
\end{align*}
\]
Here is a derivation of the word 3 + 5 \cdot 8:

\[
E \Rightarrow EOE \\
\Rightarrow 3OE \\
\Rightarrow 3+E \\
\Rightarrow 3+EOE \\
\Rightarrow 3+5OE \\
\Rightarrow 3+5*E \\
\Rightarrow 3+5*8
\]

Alternatively, we can represent this derivation by its parse tree:

The problem of constructing a parse tree for a given word in a language is the parsing problem. This problem can be solved automatically for the important class of context-free languages in which programming languages are typically expressed. In a context-free language, each derivation step rewrites a single nonterminal symbol in the working word irregardless of the context in which this nonterminal occurs. So called parser generators can automatically construct parsers for context-free languages from a description of their grammar. We next define this class of languages and their associated grammars formally.

### 3.1.2 Context-Free Languages and Grammars

A context-free grammar is a tuple \( G = (\Sigma, N, P, S) \) where

- \( \Sigma \) is a finite set of terminal symbols,
- \( N \) is a finite set of nonterminal symbols disjoint from \( \Sigma \),
- \( P \subseteq (N, (\Sigma \cup N)^*) \) is a finite set of productions, and
- \( S \in N \) is the starting symbol.

We denote a production \((X, w) \in P\) by \( X \rightarrow w \).

Let \( G = (\Sigma, N, P, S) \) be a context-free grammar. For any two words, \( u, v \in (\Sigma \cup N)^* \), we say \( v \) directly derives from \( u \), written \( u \Rightarrow v \), if there exists a production \( X \rightarrow w \) in \( P \) and \( u_1, u_2 \in (\Sigma \cup N)^* \) such that \( u = u_1Xu_2 \) and \( v = u_1wru_2 \). That is, \( v \) is the result of applying \( X \rightarrow w \) to \( u \). We denote by \( \Rightarrow^* \)
the reflexive and transitive closure of the relation ⇒ and we say that v derives
from u if $u \Rightarrow^* v$.

The language of G, denoted $\mathcal{L}(G)$, is the set of all terminal words that can
be derived from S:

$$\mathcal{L}(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$$

A language $\mathcal{L} \subseteq \Sigma^*$ is called context-free if it is the language of some context-free
grammar $G$.

Note that the grammar for arithmetic expressions that we gave above is
technically not a context-free grammar because the set of productions (as well
as the set of terminal symbols) is infinite. For now, we will skim over this
technicality. We will see later how we obtain a proper context-free grammar for
arithmetic expressions.

### 3.1.3 Backus-Naur-Form

Often, context-free grammars are given in so-called Backus-Naur-Form (BNF).
In this form, we use the symbol ::= instead of → to separate the two sides of
a production. Moreover, in a BNF, productions $X \rightarrow w_1, \ldots, X \rightarrow w_n$ for the
same nonterminal symbol $X$ can be summarized by a single rule $X ::= w_1 \mid \cdots \mid w_n$. For example, here is our grammar of arithmetic expressions in BNF:

$$x \in \mathbb{Z}$$

$$E ::= E O E \mid (E) \mid x$$

$$O ::= + \mid - \mid \ast \mid /$$

### 3.1.4 Eliminating Ambiguity

Let us reconsider our grammar for arithmetic expressions and the derivation of
the expression “$3 + 5 \ast 8$” given by the following parse tree:

This is not the only possible derivation of “$3 + 5 \ast 8$”. Another one is given by
the following parse tree:
Concrete Syntax (optional)

We call a grammar in which a word has more than one derivation ambiguous. Ambiguity is a problem because the semantics of programs is given in terms of their abstract syntax trees, which are derived from parse trees. Typically, the semantics of a program depends on the structure of its parse tree. For example, with the canonical semantics of arithmetic expressions, the first parse tree would evaluate to 43 whereas the second parse tree would evaluate to 64. Ideally, we would like to change our grammar so that the second parse tree no longer represents a valid derivation. This can be done by augmenting the grammar with additional disambiguation rules.

The problem of detecting whether a given context-free grammar is ambiguous is undecidable. Consequently, there does not exist a general algorithm that turns an ambiguous grammar into an unambiguous one. We therefore have to make do with ad hoc techniques for resolving ambiguities. Fortunately, for grammars that describe programming languages, there exist some general recipes that work well in practice.

The ambiguity in our arithmetic expression grammar that we have observed for the expression “3 + 5 * 8” stems from the fact that the grammar does not distinguish between the additive operators, ‘+’ and ‘-’, and the multiplicative operators, ‘*’ and ‘/’. We would like the multiplicative operators to bind stronger than the additive operators. We also say that the multiplicative operators have higher precedence. We can encode operator precedence by grouping expressions based on the types of operators and changing the productions so that expressions are expanded in the right order:

\[
\begin{align*}
E &::= E \ A \ E \mid T \\
A &::= + \mid - \\
T &::= T \ M \ T \mid F \\
M &::= * \mid / \\
F &::= x \mid (E)
\end{align*}
\]

Now the only valid parse tree for the expression “3 + 5 * 8” is the tree:

```
     E
   /   |
  /     O
 /       E
3       5
  |
+     |
  *     S
   |
    *  
   8
```
Our grammar is still ambiguous, though. For example, consider the expression “3 + 5 + 8”. Here are two possible parse trees for this expression:

It seems that this ambiguity does not matter from a semantic point of view because addition on the integers is associative. That is, both trees would evaluate to 16. However, in computer programs we are normally working with bounded representations of integers. In this case, the arithmetic operations are often not associative due to potential overflow of these bounds. We would therefore like the operations to be parsed in a specific order. For example, arithmetic operators are usually defined as left-associative rather than right-associative, which means that the expression “3 + 5 + 8” should be parsed similar to “(3 + 5) + 8” rather than “3 + (5 + 8)”.

To encode left-associativity of operators in our grammar, we can replace the right side of each binary expression by the base case of that expression type. This will force the repetitive matches of subexpressions onto the left side:

\[
E ::= E A T | T \\
A ::= + | - \\
T ::= T M F | F \\
M ::= * | / \\
F ::= x | (E)
\]
3.1.5 Regular Languages

Finally, let us modify our grammar for arithmetic expressions so that it is actually context-free, i.e., the terminal symbols and productions are finite sets. We do this in two steps. First, we define a context-free grammar that describes integer numbers in decimal representation:

\[
\begin{align*}
Z &::= -H \mid H \\
H &::= 0 \mid 1T \mid \cdots \mid 9T \\
T &::= \epsilon \mid 0T \mid \cdots \mid 9T
\end{align*}
\]

The starting symbol of this grammar is \( Z \) and the terminal symbols are \( \Sigma = \{-, 0, \ldots, 9\} \).

Next, we combine this grammar with the grammar:

\[
\begin{align*}
E &::= EAT \mid T \\
A &::= + \mid - \\
T &::= TMF \mid F \\
M &::= \star \mid / \\
F &::= Z \mid (E)
\end{align*}
\]

to obtain a context-free grammar for arithmetic expressions.

The productions of our grammar for integer numbers have a special form. They all match one of the following shapes:

\[
\begin{align*}
X &::= \epsilon \\
X &::= a \\
X &::= Y \\
X &::= aY
\end{align*}
\]

where \( X, Y \) are nonterminals and \( a \) is a terminal symbol. Grammars in which all productions are of these shapes form a special subclass of context-free grammars, called regular grammars. The languages of these grammar are correspondingly called regular languages.

Regular languages can be parsed more efficiently than general context-free languages. Compilers therefore split the parsing of the input program into two phases: a so-called lexing phase in which the character sequence representing the input program is converted into a token sequence, and the actual parsing phase in which the token sequence is converted into a parse tree. The tokens in the token sequence are subsequences of characters in the input program that have been grouped together, e.g., to form keywords of the language or numbers (as in the example of our arithmetic expression language). The program that takes care of the lexing phase is called lexer or tokenizer. The lexer is typically auto-generated from a regular grammar, whereas the actual parser is auto-generated from a general context-free grammar.
3.2 Abstract Syntax

We have learned that grammars define formal languages, which are sets of sequences of characters over some alphabet. We refer to this interpretation of a grammar as the concrete syntax of a language. Alternatively, we can also interpret grammars as structural recursive definitions of certain sets of tuples (representing trees). In this view, we speak of the abstract syntax of a language. The abstract syntax abstracts from aspects of the concrete syntax that are only relevant for parsing. This includes, e.g., disambiguation rules for operator precedence and associativity, parenthesis, language keywords, etc.

3.2.1 Abstract Syntax Trees

We use Backus-Naur-Form (BNF) notation to describe the abstract syntax of a language. In order to make it easier to detect whether the grammar defines the concrete or abstract syntax of a language, we write the productions of grammars for the abstract syntax as definitions of sets. As an example, consider the following grammar that defines the abstract syntax of an arithmetic expression language:

\begin{align*}
    n & \in \text{Num} & \text{numbers} \\
    x & \in \text{Var} & \text{variables} \\
    e & \in \text{Expr} :: = n \mid x \mid e_1 \text{bop} e_2 & \text{expressions} \\
    \text{bop} & \in \text{Bop} :: = + \mid * & \text{binary operators}
\end{align*}

Note that in the definition of \text{Expr}, the (meta) variables \(e_1\) and \(e_2\) also range over expressions \(Expr\). In general, we will follow the convention that in recursive grammar definitions we only declare a generic variable for each set that we define, in this case the variable \(e\) for the set \(Expr\). We then assume that all variables that have the same name but possibly different indices, here the variables \(e_1\) and \(e_2\), also range over the same set.

The expression language includes variables \(x \in \text{Var}\). The sets \text{Num} and \text{Var} are parameters of the grammar definition. In the following, you may assume that these two sets are just referring to integer numbers \(\text{Num} = \text{Var} = \mathbb{Z}\). Later in our implementation of arithmetic expressions, which will be part of our interpreter, we will identify \text{Num} with the type of double-precision floating point numbers, and \text{Var} with the type of strings.

Our grammar borrows the notation of the concrete syntax to represent the elements of the sets \text{Expr} and \text{Bop}. However, we just use the concrete syntax to sugar coat the actual mathematical representation of abstract syntax trees. We now describe this representation formally.

As a first step, we tag the productions for each set in the grammar with a unique number. We refer to these tags as \textit{variant numbers}. In our running example, we obtain the following tagged grammar of arithmetic expressions (for
Abstract Syntax

clarity, the variant numbers are underlined): 

\[ e \in \text{Expr} ::= 1 : n \mid 2 : x \mid 3 : e_1 \text{bop} e_2 \] 

expressions 

\[ \text{bop} \in \text{Bop} ::= 1 : + \mid 2 : * \] 

binary operators 

An abstract syntax tree is similar to a parse tree, except that the nodes of the tree are labeled by the variant numbers of the productions that were used in the derivation of the expression. For example, the concrete syntactic expression \( 6 \times (5 + x) \) is notational sugar for the following abstract syntax tree:

```
3
/ \ \\
1 2
6
```

We can formally represent such trees by nested tuples. Each node of the tree is represented by a tuple consisting of the variant number that labels that node, followed by the sequence of tuples representing all the subtrees rooted in the children of that node. For example, the abstract syntax tree above stands for the following tuple

\( \langle 3, (1, 6), (2, 3, (1, 5), (1, (2, x))) \rangle \).

Consequently, the grammar rules for the sets \( \text{Expr} \) and \( \text{Bop} \) really stand for inference rules that define the sets \( \text{Expr} \) and \( \text{Bop} \) using structural recursion on tuples:

\[ n \in \text{Num} \]

\( (1, n) \in \text{Expr} \)

\[ x \in \text{Var} \]

\( (2, x) \in \text{Expr} \)

\[ e_1, e_2 \in \text{Expr} \]

\( \text{bop} \in \text{Bop} \)

\[ (3, e_1, \text{bop}, e_2) \in \text{Expr} \]

\[ (1) \in \text{Bop} \]

\[ (2) \in \text{Bop} \]

By using tuples and inference rules, we obtain a mathematical precise definition of the abstract syntax of a language. However, the tuple representation is notationally heavy and cumbersome to work with. We will therefore continue to represent abstract syntax trees using concrete syntax. Often we will even omit the definition of the exact concrete syntax, relying instead on our intuition to map concrete to abstract syntax and vice versa using common conventions for operator precedence, etc. This will sometimes lead to ambiguities. For example, in the case of our arithmetic expression language, the object 4 may now stand for the number 4, the concrete expression consisting of the terminal symbol 4, and the abstract syntax tree \( (1, 4) \) of that expression. This ambiguity might be confusing in the beginning. However, it will always be clear from the context which of these mathematical objects we are referring to. As you get more familiar with these concepts, you will be able to easily distinguish between them.
Since expressions are defined using structural recursion, we can also use structural recursion to define functions on expressions. For example, we can define a function $ov$ that takes an expression $e$ and computes the set of all variables occurring in $e$:

$$ov : \text{Expr} \rightarrow 2^{\text{Var}}$$

$$ov(n) = \emptyset$$

$$ov(x) = \{x\}$$

$$ov(e_1 \text{bop}_2 e_2) = ov(e_1) \cup ov(e_2)$$

Note that by $2^{\text{Var}}$ we denote the powerset of the set $\text{Var}$.

It is instructive to compare our definition of the abstract syntax of arithmetic expressions with a corresponding definition given in terms of algebraic data types in Scala. This can be done as follows:

```scala
sealed abstract class expr
case class numHn(n: Double) extends expr
case class varHx(x: String) extends expr
case class binopHbop(bop: bopL e1: exprL eR: expr) extends expr
sealed abstract class bop
case object plus extends bop
case object times extends bop
```

We can then define the Scala version of the function $ov$:

```scala
def ov(e: Expr): Set[String] =
  e match {
    case Num(n) => Set()
    case Var(x) => Set(x)
    case BinOp(_, e1, e2) => ov(e1) ++ ov(e2)
  }
```

Note that in the function $ov$ we are using the type `Set` from the Scala standard library, which can represent finite sets of objects.

### 3.2.2 Environments and Expression Evaluation

Consider the expression $e = x \ast (y + 6)$. If we provide values for the variables $x$ and $y$, we can compute the value of the entire expression. For example, the expression $e$ evaluates to 12, if we assign $x = 1$ and $y = 2$.

An environment is a partial function $env : \text{Var} \rightarrow \text{Num}$, that assigns variables to values. We represent partial functions as sets of pairs. For example, the environment $env$ that assigns $x$ to 1 and $y$ to 2 is denoted by

$$env = \{(x, 1), (y, 2)\}$$
We typically use the following arrow notation for the individual variable assignments in an environment:

$$env = \{ x \mapsto 1, y \mapsto 2 \}$$

Moreover, for an environment $env$, we write $env[x \mapsto v]$ for the environment that is like $env$ but maps $x$ to the value $v$:

$$env[x \mapsto v](y) = \begin{cases} env(y) & \text{if } y \neq x \\ v & \text{otherwise} \end{cases}$$

We denote the set of all environment by $Env$.

Given an environment that assigns values to all the variables in an expression, we can evaluate that expression. We can formalize this idea by defining a function $eval$ using structural recursion:

$$eval : Env \times Expr \rightarrow Num$$

$$eval(env, n) = n$$
$$eval(env, x) = env(x)$$
$$eval(env, e_1 + e_2) = eval(env, e_1) + eval(env, e_2)$$
$$eval(env, e_1 \times e_2) = eval(env, e_1) \times eval(env, e_2)$$

Note that in the third case the symbol + occurs on both sides of the equation. On the left side, it stands for the abstract syntax of the addition operator, i.e., the object $\langle 1 \rangle \in Bop$. On the right side it stands for the mathematical addition operation on integers.

For a given environment $env$ and expression $e$, $eval(env, e)$ is only well-defined if $env$ is defined on all variables occurring in $e$. Mathematically, we can express this condition by $ov(e) \subseteq dom(env)$. Here, dom is the function that maps a function $f$ to its domain, i.e., the set of values on which $f$ is defined.

Again, we can directly translate the definition of our function $eval$ to a corresponding Scala function. We represent environments in Scala using the type $env$. We define this type in terms of the $Map$ type in the Scala standard library, which we can use to represent finite partial functions:

```scala
type env = Map[String, Double]
def dom(env: Env): Set[String] = env.keySet
```

The Scala version of the function $eval$ is then defined as follows:

```scala
def eval(env: Env, e: Expr): Double =
e match {
  case Num(n) => n
  case Var(x) => env(x)
  case BinOp(Plus, e1, e2) =>
    eval(env, e1) + eval(env, e2)
```
In order to encode the required precondition of $eval$ efficiently, we can rewrite this function using a nested helper function as follows:

```scala
def eval(env: Env, e: Expr): Double = {
  require (ov(e) subsetOf dom(env))
  def eval(e: Expr): Double =
    e match {
      case Num(n) => n
      case Var(x) => env(x)
      case BinOp(Plus, e1, e2) => eval(e1) + eval(e2)
      case BinOp(Times, e1, e2) => eval(e2) * eval(e2)
    }
}
```

### 3.2.3 Substitutions

When we do calculations with expressions, we often have to replace subexpressions by other expressions. We call such replacements substitutions. Particularly important are substitution operations that replace variables by expressions.

Consider the following expression:

$$3 \times x + x$$

This expression contains two occurrences of the variable $x$. If we replace both of these occurrences by the expression $x + 4$, we obtain the expression:

$$3 \times (x + 4) + (x + 4)$$

Let $e_1$ and $e_2$ be expressions and $x \in \text{Var}$ a variable, then we denote by

$$e_1[e_2/x]$$

the expression that we obtain by replacing all occurrences of $x$ in $e_1$ by $e_2$. In particular, we have:

$$(3 \times x + x)[(x + 4)/x] = 3 \times (x + 4) + (x + 4)$$

We can define the substitution function formally using structural recursion:

$$[\cdot / \cdot] : \text{Expr} \times \text{Var} \times \text{Expr} \to \text{Expr}$$

$$(n[e/x] = n)$$

$$(x[e/y] = \text{if } x = y \text{ then } e \text{ else } x)$$

$$(e_1 \text{ bop } e_2)[e/x] = e_1[e/x] \text{ bop } e_2[e/x]$$

The Scala version of the substitution function looks as follows:
def subst(e1: Expr, x: String, e2: Expr): Expr =
   e1 match {
      case Num(_) => e1
      case Var(y) => if (x == y) e2 else e1
      case BinOp(bop, e11, e12) =>
         BinOp(bop, subst(e11, x, e2), subst(e12, x, e2))
   }

The following Lemma captures an important property that relates substitution and expression evaluation:

**Lemma 3.1 (Substitution Lemma).** Let $e_1, e_2 \in \mathit{Expr}$, $x \in \mathit{Var}$, and $\mathit{env} \in \mathit{Env}$ such that $\mathit{ov}(e_1) \cup \mathit{ov}(e_2) \subseteq \mathit{dom}(\mathit{env})$. Then

$$\mathit{eval}(\mathit{env}, e_1[e_2/x]) = \mathit{eval}(\mathit{env}[x \mapsto \mathit{eval}(\mathit{env}, e_2)], e_1)$$

**Proof.** By structural induction on $e_1$ (exercise).

### 3.3 Binding and Scoping

In programming languages, identifiers are used as names for values. They are introduced through variable and constant declarations, and procedural abstractions. During evaluation, identifiers are bound to values. The lifetime of such a binding is determined by the *scope* of the binding. We will study basic binding and scoping mechanisms using a simple language of arithmetic expressions with constant declarations.

#### 3.3.1 Expressions with Constant Declarations

We extend our grammar of arithmetic expressions from Section 3.2 with constant declarations of the form:

```scala
const x = e_d; e_b
```

During evaluation of the constant declaration, the variable $x$ is bound to the value of the expression $e_d$ so that occurrences of $x$ in $e_b$ refer to that value, much like a val declaration in Scala. We call $e_d$ the *defining expression* of the declaration and $e_b$ the *body* of the declaration. We also refer to $e_b$ as the *scope* of the binding of $x$. The scope of a binding determines the lifetime of the binding during evaluation of an expression. If the scope of a binding is completely determined by the syntactic structure of the expression, as in the case of our constant declarations, we speak of *static binding* and otherwise we speak of *dynamic binding*. We will see examples of dynamic binding later when we introduce procedural abstractions. However, most modern programming languages use only static binding.
The complete abstract syntax of our extended language is now as follows:

\[
\begin{align*}
  n & \in \text{Num} \quad \text{numbers} \\
  x & \in \text{Var} \quad \text{variables} \\
  e & \in \text{Expr} ::= n \mid x \mid e_1 \text{bop} e_2 \mid \text{const } x = e_d; e_b \quad \text{expressions} \\
  \text{bop} & \in \text{Bop ::= + } \mid * \quad \text{binary operators}
\end{align*}
\]

Note that in our concrete syntax, constant declarations are right associative. Thus, the expression

\[
\text{const } x = 5; \text{const } y = x + 2; \text{const } z = y \times 5; z + z
\]

is implicitly parenthesized as follows:

\[
\text{const } x = 5; (\text{const } y = x + 2; (\text{const } z = y \times 5; z + z))
\]

Further note that constant declarations may also occur within defining expressions of other constant declarations:

\[
\text{const } y = (\text{const } x = 5; x + 2); \text{const } z = y \times 5; z + z
\]

We distinguish between defining occurrences and using occurrences of variables. A defining occurrence introduces a new binding during evaluation of an expression, whereas a using occurrence yields a value according to the current binding of the variable. Consider the following expression:

\[
\text{const } x = 2 \times 3; \text{const } y = x + 5; x \times (z + y)
\]

If we mark the defining occurrences by over-lining them, we get the following:

\[
\text{const } x = 2 \times 3; \text{const } y = x + 5; x \times (z + y)
\]

In addition to defining and using occurrences, we also distinguish between bound and free occurrences. Defining occurrences are always bound occurrences, whereas using occurrences can be either bound or free. A using occurrence of a variable is bound if it occurs inside the scope of a defining occurrence of the same variable. A using occurrence of a variable that that has no associated defining occurrence is free.

We can make the binding structure in an expression explicit using arrows:

\[
\text{const } x = 2 \times 3; \text{const } y = x + 5; x \times (z + y)
\]

Every arrow points from a bound using occurrence of a variable to the associated defining occurrence. The using occurrence of \(z\) is the only free occurrence.

With the different notions of variable occurrences in place, we can define functions \(ov\), \(fv\), and \(bv\) that, given an expression \(e\), compute the set of all
variables occurring in \( e \), the set of free variables occurring in \( e \), respectively, the set of bound variables occurring in \( e \).

\[
\begin{align*}
\text{ov} : & \text{Expr} \rightarrow 2^{\text{Var}} \\
\text{ov}(n) &= \emptyset \\
\text{ov}(x) &= \{x\} \\
\text{ov}(e_1 \text{ bop } e_2) &= \text{ov}(e_1) \cup \text{ov}(e_2) \\
\text{ov}(\text{const } x = e_d; e_b) &= \{x\} \cup \text{ov}(e_d) \cup \text{ov}(e_b)
\end{align*}
\]

\[
\begin{align*}
\text{fv} : & \text{Expr} \rightarrow 2^{\text{Var}} \\
\text{fv}(n) &= \emptyset \\
\text{fv}(x) &= \{x\} \\
\text{fv}(e_1 \text{ bop } e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2) \\
\text{fv}(\text{const } x = e_d; e_b) &= (\text{fv}(e_b) \setminus \{x\}) \cup \text{fv}(e_d)
\end{align*}
\]

\[
\begin{align*}
\text{bv} : & \text{Expr} \rightarrow 2^{\text{Var}} \\
\text{bv}(n) &= \emptyset \\
\text{bv}(x) &= \emptyset \\
\text{bv}(e_1 \text{ bop } e_2) &= \text{bv}(e_1) \cup \text{bv}(e_2) \\
\text{bv}(\text{const } x = e_d; e_b) &= \{x\} \cup \text{ov}(e_d) \cup \text{ov}(e_b)
\end{align*}
\]

### 3.3.2 Evaluation with Bindings

The notion of variable binding is strongly related to evaluation as it is during evaluation when a defining occurrence of a variable is bound to a value. We can make this formally precise by extending the \textit{eval} function from Section 3.2.2 to our expression language with constant declarations.

As before, the evaluation function will be defined with respect to a value environment \( \text{env} : \text{Var} \rightarrow \text{Num} \) and we denote by \( \text{Env} \) the set of all such environments.

Before we give the definition of the \textit{eval} function, let us go through an example. Consider the expression

\[
\text{const } x = x + 2; x \ast y
\]

and the environment

\[
\text{env} = \{x \mapsto 1, y \mapsto 2\}
\]

Evaluation of a constant declaration proceeds as follows. First, we evaluate the defining expression \( x + 2 \) in the current environment \( \text{env} \). In particular, the free occurrence of \( x \) in this expression is evaluated to \( \text{env}(x) = 1 \). The result of
evaluating the expression \( x + 1 \) is thus 3. Next, we bind the result value to the declared variable \( x \), obtaining a new environment \( env' \):

\[
env' = env[x \mapsto 3] = \{ x \mapsto 3, y \mapsto 2 \}
\]

Then, we evaluate the body \( x \ast y \) of the constant declaration using the updated environment \( env' \), yielding 6 as the value of the entire expression. Note that if the constant declaration is nested within a larger expression, then the updated environment \( env' \) is discarded once evaluation of the body is completed (the body of the declaration is the scope of the binding). We then proceed with the original environment \( env \) to evaluate some other part of the larger expression.

We can formalize the new evaluation function \( eval \) using structural recursion as follows:

\[
eval : Env \times Expr \rightarrow \mathbb{Num}
\]

\[
eval(env, n) = n
\]

\[
eval(env, x) = env(x)
\]

\[
eval(env, e_1 + e_2) = eval(env, e_1) + eval(env, e_2)
\]

\[
eval(env, e_1 \ast e_2) = eval(env, e_1) \cdot eval(env, e_2)
\]

\[
eval(env, \text{const } x = e_d; e_b) = \text{let } env' = env[x \mapsto eval(env, e_d)] \text{ in } eval(env', e_b)
\]

To ensure that \( env \) is well-defined we must require that \( env \) is defined on all free variables of the evaluated expression \( e \), i.e., \( \text{fv}(e) \subseteq \text{dom}(env) \).

### 3.3.3 Substitutions and Bindings

In the previous class, we have seen that substitution is a simple affair for languages that do not have any binding constructs. For languages with binding constructs the situation is more complicated because we may run into the problem of variable capturing. To understand this problem, consider the expression

\[
e = \text{const } y = 2 \ast 3; y + x
\]

and suppose we want to substitute the free occurrence of \( x \) in \( e \) by the expression \( y \), i.e., compute \( e[y/x] \). If we compute the substitution naively, we obtain the expression

\[
e' = \text{const } y = 2 \ast 3; y + y
\]

However, this substitution is not correct because in \( e' \) the originally free occurrence of \( y \) in the expression is captured by the constant declaration that binds \( y \). We want to define the substitution function in such a way that variable capturing is avoided. The reason for this is that we always want the substitution property to hold. For our new language, the substitution property can be formulated as follows: for all \( env \in Env, x \in \text{Var}, e, e_s \in Expr \), if \( \text{fv}(e) \cup \text{fv}(e_s) \subseteq \text{dom}(env) \), then

\[
eval(env, e[e_s/x]) = eval(env[x \mapsto eval(env, e_s)], e)
\]
For the example above, the substitution property is violated because for the environment
\[ env = \{ x \mapsto 1, y \mapsto 2 \} \]
we have
\[ eval(env, e') = 12 \]
whereas
\[
\begin{align*}
\text{eval}(\text{env}[x \mapsto \text{eval}(\text{env}, y)], e) &= \text{eval}(\text{env}[x \mapsto 2], e) \\
&= \text{eval}((x \mapsto 2, y \mapsto 2), e) \\
&= 8
\end{align*}
\]

Before we can define a capture-avoiding substitution function, let us first define a preliminary substitution function \( subst \) that satisfies the substitution property under certain conditions:

\[
\begin{align*}
subst : \text{Expr} \times \text{Var} \times \text{Expr} &\rightarrow \text{Expr} \\
subst(n, x, e_s) &= n \\
subst(y, x, e_s) &= \text{if } x = y \text{ then } e_s \text{ else } y \\
subst(e_1 \text{ bop } e_2, x, e_s) &= subst(e_1, x, e_s) \text{ bop } subst(e_2, x, e_s) \\
subst(\text{(const y}\text{=}e_d;e_b), x, e_s) &= \text{let } e'_b = \text{if } x = y \text{ then } e_b \text{ else } subst(e_b, x, e_s) \text{ in } \text{const y}\text{=}subst(e_d, x, e_s);e'_b
\end{align*}
\]

A substitution \( subst(e, x, e_s) \) that uses the preliminary substitution function satisfies the substitution property provided that none of the free variables of the substituent expression \( e_s \) are bound anywhere in \( e \), formally \( \text{fv}(e_s) \cap \text{bv}(e) = \emptyset \). This additional side-condition ensures that in \( subst(e, x, e_s) \), none of the free variables of \( e_s \) will be captured. In particular, this condition is always satisfied if \( e_s \) is closed, i.e., \( \text{fv}(e_s) = \emptyset \).

For example, consider the expression
\[ e_r = \text{const } z = 2 \ast 3; z + x \]
then we have
\[ subst(e_r, x, y) = \text{const } z = 2 \ast 3; z + y \]
which is a valid substitution.

We now have a substitution function that yields valid substitutions under certain conditions. Still, it is useful to have a substitution function that always satisfies the substitution property, so that we don’t have to worry about clumsy side conditions.

The key observation to obtain a general substitution function is that for the evaluation of an expression the actual names of bound variables don’t matter. Given an expression, we are free to rename bound variables, as long as we rename them consistently and without variable capturing. Consistent renaming of bound variables does not affect the result of evaluation. For example, the expression \( e_r \) defined above is a consistent renaming of our earlier expression
\[ e = \text{const } y = 2 \ast 3; y + x \]
In particular, we have for all environments $env$ with $x \in \text{dom}(env)$:

$$\text{eval}(env, e) = \text{eval}(env, e_r)$$

Thus, in terms of evaluation, the two expressions $e$ and $e_r$ are equivalent. This gives us the following recipe for a general substitution function: given $e$, $x$, and $e_s$, compute $e[e_s/x]$ as follows:

- first, consistently rename $e$ to obtain $e_r$ such that $\text{bv}(e_r) \cap \text{fv}(e_s) = \emptyset$
- and then compute $\text{subst}(e_r, x, e_s)$.

In programming language terminology we often speak of $\alpha$-renaming instead of consistent renaming and we call two terms $e$ and $e_r$ $\alpha$-equivalent if $e$ can be obtained from $e_r$ by consistent renaming of bound variables. We can quite easily formalize the intuitive idea of $\alpha$-equivalence by defining an appropriate equivalence relation $\sim$ on expressions:

$$\begin{align*}
  &\frac{e_b' = \text{subst}(e_b, x, y)}{\text{const } x = e_d; e_b \sim e_d'}
  \quad \frac{y \notin \text{ov}(e_b)}{\text{const } y = e_d'; e_b'} \\
  &\frac{e_1 \sim e_1' \quad e_2 \sim e_2'}{e_1 \text{ bop } e_2 \sim e_1' \text{ bop } e_2'}
  \quad \frac{e_d \sim e_d' \quad e_b \sim e_b'}{\text{const } x = e_d; e_b \sim e_d'; e_b'}
\end{align*}$$

With the definition of $\alpha$-equivalence in place, we can now formally define the condition that a general substitution function must satisfy so that we always ensure the substitution property.

**Definition 3.2.** A substitution function for $Expr$ is a function

$$s : Expr \times Var \times Expr \rightarrow Expr$$

such that for all $e, e_s, e_r \in Expr$ and $x \in Var$, if $e \sim e_r$ and $\text{bv}(e_r) \cap \text{fv}(e_s) = \emptyset$ then

$$s(e, x, e_s) \sim \text{subst}(e_r, x, e_s)$$

One can show that a general substitution function always exists, provided that $Var$ is infinite (i.e., we never run out of variables for renaming). However, it is somewhat cumbersome to formally define a specific substitution function that satisfies the above definition. The problem is that a concrete substitution function requires us to make a deterministic choice how we pick fresh variables for $\alpha$-renaming. When one implements substitution functions in an interpreter or compiler, the easiest way to solve this problem is to maintain a global counter that can be used to generate fresh variable names when they are needed.
Whenever we will use substitutions $e_r[e_s/x]$ throughout the remainder of the course, we will always satisfy the condition that the expression $e_s$ is closed. Hence, we do not have to worry about variable capturing and we can just use the function $subst$ to compute the substitution. Nevertheless, it is important to understand the concepts of variable capturing and consistent renaming.
Chapter 4

Semantics

The semantics of a programming language assigns meaning to programs. Typically, one distinguishes three types of semantics: denotational semantics, axiomatic semantics, and operational semantics. Denotational semantics assigns to each program expression the mathematical object that it computes. We have already seen an example of denotational semantics, namely the \texttt{eval} function that maps an arithmetic expression to the mathematical number that the expression evaluates to. Axiomatic semantics, on the other hand, describes the meaning of programs in terms of its effect on assertions about the program’s states. It makes statements of the form: if $P$ holds in a program state, then $Q$ holds in the new state after a command $c$ has been executed. The main use of axiomatic semantics is to formally prove that a program satisfies its specification. Finally, operational semantics focuses on the computational aspects of a programming language. It is the type of semantics that is best suited for studying programming language interpreters. We will therefore focus our attention on operational semantics.

The operational semantics of a language describes how a program is evaluated, one step at a time. Typically, the operational semantics of a language is defined using structural recursion over the syntax of the language. One therefore also speaks of structural operational semantics, or short SOS. We distinguish between big-step and small-step SOS. We will discuss both of these variants, starting with big-step SOS.

4.1 Big-Step Structural Operational Semantics

The big-step SOS of a language defines what values a given expression in a language may evaluate to. It is perhaps the most natural type of semantics for studying the operational aspects of a language because it provides a blueprint for an interpreter of the language that almost immediately translates into an actual implementation. For this reason, some people also refer to the big-step SOS as the natural semantics.
4.1.1 Defining the Big-Step SOS

We first extend our language from Sec. 3.3 with new programming constructs to make it feel more like a realistic subset of JavaScript. Specifically, we introduce:

- a new type of values, Booleans, along with Boolean expressions for conjunctions $e_1 \land e_2$ and disjunctions $e_1 \lor e_2$;
- comparison operators for equality $e_1 == e_2$ and disequality $e_1 != e_2$; and
- conditional expressions $e_1 ? e_2 : e_3$, which evaluate to $e_2$ if $e_1$ is true and $e_3$, otherwise.

The abstract syntax of our new language is as follows:

$$
x \in \text{Var} \quad \text{variables}
n \in \text{Num} \quad \text{numbers}
b \in \text{Bool} ::= \text{true} | \text{false} \quad \text{Booleans}
v \in \text{Val} ::= n | b \quad \text{values}
e \in \text{Expr} ::= v | x | e_1 \text{ bop } e_2 | \text{const } x = e_d; e_b | e_1 ? e_2 : e_3 \quad \text{expressions}
bop \in \text{Bop} ::= + | * | \&\& | || | == | != \quad \text{binary operators}
$$

Note that we introduce a new syntactic class \text{Val} that consists of numbers and Booleans. We call the elements of \text{Val} \textit{values}. A value is an expression that does not require further evaluation.

We now define the big-step structural operational semantics of our language. As for the \text{eval} function from the previous chapter, the big-step semantics will be defined with respect to a value environment $env$ that assigns values to the free variables of the expression under evaluation. We modify the type of environments to be partial mappings of variables to values:

$$
env \in \text{Env} = \text{Var} \rightarrow \text{Val}
$$

This is a subtle but important change: from now on, all computations in our language will produce values, which are themselves expressions of our language. That is, we view computation as rewriting of syntactic expressions.

The definition of the big-step SOS will look similar to the \text{eval} function that we have seen in Chapter 3. In particular, it will still give meaning to expressions by mapping them to values and it will still be defined using structural recursion on the abstract syntax of expressions. One important difference to the \text{eval} function is that the big-step SOS is a relation rather than a function. This means that the semantics may allow for non-determinism, i.e., one expression $e$ may evaluate to more than one value. Although, we will not make use of this feature, a non-deterministic semantics is often useful to give more freedom to a language implementer. For example, the semantics of many programming languages does not determine the order in which expressions are evaluated. This
Big-Step Structural Operational Semantics gives a compiler or interpreter for the language more room for applying program optimizations. If expression can have side-effects (such as the post-increment operator $x++$ that is supported by many languages), then changing the order in which an expression is evaluated may change the result of the evaluation.

We formalize the big-step SOS as a relation

$$
\text{env} \vdash e \Downarrow v \subseteq \text{Env} \times \text{Expr} \times \text{Val}
$$

A judgment form $\text{env} \vdash e \Downarrow v$ states that under value environment $\text{env}$, the expression $e$ evaluates to value $v$. We define this evaluation relation by providing one or more inference rules for each syntactic construct in our language. The complete set of rules is given in Figure 4.1. We discuss them one at a time.

We start with values $v$, which by their very nature do not need to be further evaluated. This is captured by the following simple inference rule:

$$
\text{env} \vdash v \Downarrow v
$$

We give each inference rule a name, which we put next to the rule. Next are variables, which are assigned to values by the environment, yielding the following inference rule:

$$
\frac{x \in \text{dom}(\text{env})}{\text{env} \vdash x \Downarrow \text{env}(x)} \text{EvalVar}
$$

We move on to the arithmetic operators $+$ and $*$ for which the semantics is getting more interesting. Since our language has two types of values, numbers and Booleans, we can write expressions such as $1 + \text{false}$. The question is: how should we evaluate such expressions? There is no clear answer to this question and it often depends on choices made in the overall language design. Most languages associate types to expressions that govern how such mixed expressions should be treated. Some languages such as Scala provide typing rules that strictly forbid expressions such as $1 + \text{false}$ to be even considered for evaluation. These typing rules can be enforced statically at compile time or dynamically at run time. Other languages such as JavaScript have no strict typing rules. Instead, they define type coercions that determine how values of one type are converted to values of another type. We will study typed languages later. For now, we follow JavaScript’s treatment of expressions with mixed types. To do so, we introduce two coercion functions that convert values of type $\text{Bool}$ to values of type $\text{Num}$ and vice versa:

$$
toNum : \text{Val} \rightarrow \text{Num}
$$

$$
toNum(n) = n
$$

$$
toNum(\text{true}) = 1
$$

$$
toNum(\text{false}) = 0
$$

$$
toBool : \text{Val} \rightarrow \text{Bool}
$$

$$
toBool(b) = b
$$

$$
toBool(n) = \text{if } n = 0 \text{ then false else true}
$$
We can now define the semantics of the operator $\ast$ by the following rule:

$$
\frac{env \vdash e_1 \downarrow v_1 \quad env \vdash e_2 \downarrow v_2 \quad v = toNum(v_1) + toNum(v_2)}{env \vdash e_1 \ast e_2 \downarrow v} \quad \text{EvalPlus}
$$

The semantics of the operator $\ast$ is defined by an according rule called EvalTimes.

One important detail that you need to understand about the rule EvalPlus is that this rule does not specify the order in which the two subexpressions $e_1$ and $e_2$ are evaluated. Although the premises $env \vdash e_1 \downarrow v_1$ comes before the premise $env \vdash e_2 \downarrow v_2$ in the rule, the order in which these two premises are checked is unspecified. The behavior of the rule is therefore non-deterministic and it is up to the implementer of the rule to decide which order should be used. Note that evaluation order is often important in practice. For example, if arithmetic operations are subject to overflow, then the result of an expression can depend on the order of evaluation. Also, if $e_1$ and $e_2$ have side-effects, then the evaluation order matters. If the operational semantics should define a specific evaluation order, e.g. left-to-right, then one needs to switch to a small-step SOS.

Next, we define the semantics of constant declarations, which closely follows the semantics that we used in the eval function:

$$
\frac{env \vdash e_d \downarrow v_d \quad env' = env[x \mapsto v_d] \quad env' \vdash e_b \downarrow v_b}{env \vdash const x = e_d; e_b \downarrow v_b} \quad \text{EvalConstDecl}
$$

We next move on to the comparison operators, which define in terms of actual equality of values:

$$
\frac{env \vdash e_1 \downarrow v_1 \quad env \vdash e_2 \downarrow v_2 \quad b = (v_1 = v_2)}{env \vdash e_1 == e_2 \downarrow b} \quad \text{EvalEqual}
$$

Again, the order in which the subexpressions are evaluated is not specified by the rule. The semantics of $!==$ is given accordingly by a rule EvalDisequal.

The semantics of conditional expressions $e_1 \_ e_2 : e_3$ is now straightforward. To simplify the presentation, we introduce two rules, one for the case where $e_1$ evaluates to true and one for the case where it evaluates to false:

$$
\frac{env \vdash e_1 \downarrow v_1 \quad toBool(v_1) = true \quad env \vdash e_2 \downarrow v_2}{env \vdash e_1 \_ e_2 : e_3 \downarrow v_2} \quad \text{EvalIfThen}
$$

$$
\frac{env \vdash e_1 \downarrow v_1 \quad toBool(v_1) = false \quad env \vdash e_3 \downarrow v_3}{env \vdash e_1 \_ e_2 : e_3 \downarrow v_3} \quad \text{EvalIfElse}
$$

Finally, we define the semantics of the Boolean operators $\&\&$ and $||$. There are some subtleties in how Boolean operators are treated in programming languages. We start simple and give a preliminary semantics in which we treat the Boolean operators as if they were logical operators. For this purpose, we define
the two logical operators for conjunction, $\land$, and disjunction, $\lor$, on Booleans as follows:

\[
\land : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}
\]

\[
\text{true} \land b = b
\]

\[
\text{false} \land b = \text{false}
\]

\[
\lor : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}
\]

\[
\text{true} \lor b = \text{true}
\]

\[
\text{false} \lor b = b
\]

The semantics of the operator `&&` is then given by the following inference rule:

\[
\frac{env \vdash e_1 \Downarrow v_1 \quad b_1 = \text{toBool}(v_1) \quad env \vdash e_2 \Downarrow v_2 \quad b_2 = \text{toBool}(v_2)}{env \vdash e_1 \land \land e_2 \Downarrow b_1 \land b_2} \text{EvalAnd}
\]

The semantics of `||` is given in terms of $\lor$ by a corresponding rule `EvalOr`.

### 4.1.2 Short-Circuit Evaluation

Our preliminary semantics of the Boolean operators that we formalized in the rules `EvalAnd` and `EvalOr` does not actually correspond to the semantics that is implemented in most programming languages. Observe that the operator $\land$ has an interesting property. The term $\text{false} \land b_2$ is $\text{false}$ no matter whether $b_2$ is $\text{true}$ or $\text{false}$. This allows us to optimize the evaluation of $e_1 \land \land e_2$ as follows. If $e_1$ evaluates to $\text{false}$, then there is no point in continuing evaluation of $e_2$. We can immediately return $\text{false}$. Only if $e_1$ evaluates to $\text{true}$, do we continue evaluation of $e_2$. We refer to this optimization as *short-circuit* evaluation. Short-circuit evaluation is a useful programming feature. Not only does it safe computation time, it can also help us write more compact code. For example, if $e_2$ is an expression that can only be safely evaluated if some condition $e_1$ is satisfied, then we can simply write this as $e_1 \land \land e_2$. The short-circuit semantics ensures that $e_2$ will never be evaluated, unless $e_1$ is $\text{true}$.

To formalize the short-circuit semantics of `&&`, we introduce two rules: one for the case where $e_1$ evaluates to $\text{true}$ and one were it evaluates to $\text{false}$. We start with the second case:

\[
\frac{env \vdash e_1 \Downarrow v_1 \quad \text{false} = \text{toBool}(v_1)}{env \vdash e_1 \land \land e_2 \Downarrow v_1} \text{EvalAndFalse}
\]

One interesting detail of this rule is that the result of the evaluation is not $\text{false}$ but $v_1$, which gives $\text{false}$ when it is coerced to the value type `Bool`. This means that the `&&` operator can actually yield values that are not Booleans. For example, the following expression now evaluates to $\text{true}$:

\[
0 == (0 \&\& \text{true})
\]

This might be counter-intuitive but the `EvalAndFalse` rule faithfully reflects the actual semantics of the `&&` operator in JavaScript.
\begin{align*}
\text{env} \vdash v \downarrow v & \quad \text{EVALVAL} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALPLUS} \quad x \in \text{dom}(\text{env}) \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALVAR} \\
\text{env} \vdash e_1 \downarrow e_2 \downarrow v & \quad v = \text{toNum}(v_1) + \text{toNum}(v_2) \\
\text{env} \vdash e_1 \downarrow e_2 \downarrow v & \quad \text{EVALTIMES} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALCONSTDECL} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALEQUAL} \\
\text{env} \vdash e_1 \downarrow e_2 \downarrow v & \quad \text{EVALDISEQUAL} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALIFTHEN} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALIFELSE} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALANDFALSE} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALANDTRUE} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALORTURE} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALORFALSE} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_1 \downarrow v_1 & \quad \text{EVALBOOL} \\
\text{env} \vdash e_2 \downarrow v_2 & \quad \text{EVALBOOL}
\end{align*}

Figure 4.1: Inference rules that define the big-step SOS of our expression language
The case where \( e_1 \) evaluates to \( \text{true} \) is covered by the following rule:

\[
\frac{env \vdash e_1 \downarrow v_1 \quad \text{true} = \text{toBool}(v_1) \quad env \vdash e_2 \downarrow v_2}{env \vdash e_1 \& e_2 \downarrow v_2} \quad \text{EVALAndTrue}
\]

The short-circuit semantics of the operator \( |\| \) can be defined correspondingly. The complete set of rules that defines the big-step semantics of our language is given in Figure 4.1

### 4.1.3 Interpreting Big-Step Derivations

We can use the inference rules of the big-step semantics to build derivation trees for the judgment forms \( env \vdash e \downarrow v \) top to bottom, i.e., starting with the base cases given by the rules EVALVal and EVALVar. The derivations constructed this way represent proofs of valid evaluations. For example, the following derivation proves that \( \{x \mapsto 2\} \vdash x + 3 \downarrow 5 \) is a valid evaluation:

\[
\begin{align*}
& x \in \text{dom}(\{x \mapsto 2\}) \\
& \{x \mapsto 2\} \vdash x \downarrow 2 \quad \text{EVALVar} \\
& \{x \mapsto 2\} \vdash 3 \downarrow 3 \quad \text{EVALVar} \\
& 5 = 3 + 2 \quad \text{EVALPlus} \\
& \{x \mapsto 2\} \vdash x + 3 \downarrow 5 \quad \text{EvalPlus}
\end{align*}
\]

We can also build the derivations bottom-up, starting with some concrete judgment form \( env \vdash e \downarrow v \). This is possible because the inference rules are syntax directed. By looking at the syntactic structure of the expression \( e \) we always know which rule to apply next. In other words, by reading the inference rules bottom-up, we obtain the blueprint for the implementation of a language interpreter. We explain this through an example.

Suppose we are given the environment

\[
env = \{x \mapsto 2\}
\]

and we want to evaluate the expression

\[
e = x ? 5 : \text{false}
\]

Then we can compute the result of the evaluation by filling in the red question marks in the following partial derivation:

\[
\frac{?}{\{x \mapsto 2\} \vdash x ? 5 : \text{false} \downarrow ?} \quad \text{Eval?}
\]

First, we need to find out which rule we could have possibly applied in the final step of the derivation. Looking at the rules in Figure 4.1 and the shape of expression \( e \), we observe that the only candidates that we have to consider are the rules EVALIfThen and EVALIfElse. We don’t know yet which of the rules it is. However, the two rules have a common premise that continues evaluation with the subexpression \( e_1 \). The expression \( e_1 \) in the rule matches
the subexpression $x$ in $e$. Thus, we can fill in some of the missing parts of the derivation as follows:

$$
\begin{array}{l}
\{x \mapsto 2\} \vdash x \downarrow \text{Eval} \downarrow ? = \text{toBool}(?) \quad \{x \mapsto 2\} \vdash ? \downarrow ? \text{Eval} \downarrow ? \text{EvalIf}?
\end{array}
$$

Next, we focus our attention on the evaluation of $x$. We see that the only rule that can complete this subderivation is the rule EvalVar. This allows us to fill in more of the missing parts:

$$
\begin{array}{l}
x \in \text{dom}(\{x \mapsto 2\}) \quad \text{EvalVar} \quad ? = \text{toBool}(?) \quad \{x \mapsto 2\} \vdash ? \downarrow ? \text{Eval} \downarrow ? \text{EvalIf}?
\end{array}
$$

Now, we have obtained 2 as the result of evaluating $x$ in $env$. Applying the function $\text{toBool}$ to 2 yields true. Thus we must have used the rule EvalIfThen, which allows us to fill more of the remaining holes in the derivation:

$$
\begin{array}{l}
x \in \text{dom}(\{x \mapsto 2\}) \quad \text{EvalVar} \quad \text{true} = \text{toBool}(2) \quad \{x \mapsto 2\} \vdash 5 \downarrow 5 \text{Eval} \downarrow ? \text{EvalIfThen}
\end{array}
$$

To complete the derivation, we observe that the only matching rule for the missing rule application is EvalVal. This allows us to fill in the remaining missing parts:

$$
\begin{array}{l}
x \in \text{dom}(\{x \mapsto 2\}) \quad \text{EvalVar} \quad \text{true} = \text{toBool}(2) \quad \{x \mapsto 2\} \vdash 5 \downarrow 5 \text{EvalVal} \text{EvalIfThen}
\end{array}
$$

Thus, the result of evaluating $e$ in environment $env$ is 5.

### 4.2 Small-Step Operational Semantics

One drawback of the big-step structural operational semantics is that it cannot faithfully model advanced language features such as concurrency. Also, subtle details of a language implementation, e.g., the order in which expressions are evaluated, are not captured by a big-step SOS. Sometimes these details are important. For example, the values that a program computes may depend on the chosen execution order. If a language fixes a specific execution order, then this should be reflected in the operational semantics. For example, in JavaScript all binary operators (except for the short-circuited ones) are evaluated left to right. Another restriction of big-step semantics is that it cannot describe non-terminating computations. To capture such details, we have to switch to a small-step SOS. This type of SOS is more fine-grained and can model execution order and non-termination. However, it is perhaps less natural than big-step SOS as it usually yields more complex inference rules.
4.2.1 Step-Wise Expression Evaluation

We will be using the same language as for the big-step SOS in the previous section. We will formalize the small-step SOS by a step relation that iteratively simplifies a given expression until a value has been obtained. For example, the expression

\[(2 + 8) \times 3\]

will be simplified as follows:

\[(2 + 8) \times 3 \rightarrow 10 \times 3 \rightarrow 30\]

In each step, a subexpression that can be simplified in a single step is chosen in the current expression \(e\). Then that subexpression is simplified and the result substituted back into the complete expression \(e\). In the above example, the subexpressions that are chosen for simplification in the next step are underlined.

We will again define the semantics using syntax-directed inference rules. These rules can be categorized into two types of rules:

- **“Do” rules**: these rules perform an actual simplification step. The second step in our example corresponds to the application of a do rule: we simplified the expression \(10 \times 3\) by multiplying the two values, yielding the value 30.

- **“Search” rules**: these rules select the next subexpressions that should be simplified and then apply the corresponding do rule. The first step in our example corresponds to the application of a search rule: we selected the subexpression \(2 + 8\) in the expression \((2 + 8) \times 3\) and performed the do rule for that subexpression, yielding \(10 \times 3\).

The step relation will still be depending on a value environment, as we still have to be able to assign values to free variables occurring in an expression. That is, formally the small-step SOS is a relation

\[\vdash \_ \rightarrow \_ \subseteq Env \times Expr \times Expr\]

Observe the difference to the big-step SOS, which related expressions and the final values of the computation. Instead, the small-step SOS relates expressions with expressions. A judgment form \(env \vdash e \rightarrow e'\) states that under value environment \(env\), the expression \(e'\) is obtained from \(e\) by a single simplification step. The do rules are summarized in Figure 4.2 and the search rules in Figure 4.3. We discuss the two sets of rules in tandem.

First, note that there is no rule for values. By definition a value cannot be simplified any further. Once simplification of an expression yields a value, the evaluation of that expression terminates. In fact, we will define the small-step SOS in such a way that if for some environment \(env\) and expression \(e\), there exists no expression \(e'\) such that \(env \vdash e \rightarrow e'\), then \(e\) must be a value. That is, whenever the small-step evaluation terminates, we obtain a value.
\[ x \in \text{dom}(env) \]
\[ \frac{\text{env} \vdash x \rightarrow \text{env}(x)}{\text{DoVAR}} \]
\[ v = \text{toNum}(v_1) + \text{toNum}(v_2) \]
\[ \frac{\text{env} \vdash v_1 + v_2 \rightarrow v}{\text{DoPLUS}} \]
\[ b = (v_1 = v_2) \]
\[ \frac{\text{env} \vdash v_1 \implies v_2 \rightarrow b}{\text{DoEQUAL}} \]
\[ to\text{Bool}(v_1) = \text{true} \]
\[ \frac{\text{env} \vdash v_1 ? e_2 : e_3 \rightarrow e_2}{\text{DoIFTHEN}} \]
\[ to\text{Bool}(v_1) = \text{false} \]
\[ \frac{\text{env} \vdash v_1 ? e_2 : e_3 \rightarrow e_3}{\text{DoIFELSE}} \]
\[ \text{false} = to\text{Bool}(v_1) \]
\[ \frac{\text{env} \vdash v_1 \land e_2 \rightarrow v_1}{\text{DoANDFALSE}} \]
\[ \text{true} = to\text{Bool}(v_1) \]
\[ \frac{\text{env} \vdash v_1 \mid e_2 \rightarrow v_1}{\text{DoORTURE}} \]
\[ \text{true} = to\text{Bool}(v_1) \]
\[ \frac{\text{env} \vdash v_1 \mid e_2 \rightarrow e_2}{\text{DoORFALSE}} \]

Figure 4.2: Do rules for the small-step SOS of our expression language

\[ \frac{\text{env} \vdash e_1 \rightarrow e_1'}{\text{SEARCHBOP1}} \]
\[ \text{bop} \notin \{\land, \mid\} \]
\[ \frac{\text{env} \vdash e_2 \rightarrow e_2'}{\text{SEARCHBOP2}} \]
\[ \frac{\text{env} \vdash e_d \rightarrow e_d'}{\text{SEARCHCONSTDECL1}} \]
\[ \frac{\text{const} x = e_d; e_b \rightarrow \text{const} x = e_d'; e_b}{\text{SEARCHCONSTDECL2}} \]
\[ \frac{\text{env} = \text{env}[x \mapsto v_d]}{\text{SEARCHIF}} \]
\[ \frac{\text{env} \vdash e_1 \rightarrow e_1'}{\text{SEARCHIF}} \]

Figure 4.3: Search rules for the small-step SOS of our expression language
Our first do rule concerns variables. The rule is almost identical to the EvalVar rule:

\[
\begin{align*}
  x \in \text{dom}(\text{env}) \\
  \text{env} \vdash x \rightarrow \text{env}(x)
\end{align*}
\]

DoVar

The arithmetic expressions \(e_1 + e_2\) and \(e_1 * e_2\) are handled by a combination of search and do rules. The search rules enforce that first \(e_1\) and then \(e_2\) are completely evaluated to values. Then a do rule performs the actual computation. We first give the do rule for \(*\):

\[
\begin{align*}
  v = \text{toNum}(v_1) + \text{toNum}(v_2) \\
  \text{env} \vdash v_1 + v_2 \rightarrow v
\end{align*}
\]

DoPlus

The important point is that the rule only applies to addition of values \(v_1 + v_2\) but not expressions \(e_1 + e_2\) where \(e_1\) and \(e_2\) still need to be evaluated, i.e., where either \(e_1 \notin \text{Val}\) or \(e_2 \notin \text{Val}\) holds. The premise that \(v_1\) and \(v_2\) are values is stated implicitly through the choice of the names of these meta variables. We could make this premise explicit by writing the DoPlus rule like this:

\[
\begin{align*}
  e_1, e_2 \in \text{Val} \\
  v = \text{toNum}(e_1) + \text{toNum}(e_2) \\
  \text{env} \vdash e_1 * e_2 \rightarrow v
\end{align*}
\]

DoPlus

The two DoPlus rules are equivalent. For notational convenience, we will continue to state the domains over which meta variables in rules range by choosing appropriate names for these meta variables.

The cases where one of the subexpressions of \(e_1 + e_2\) has not yet been reduced to a value are handled by appropriate search rules that we discuss next. We want to define the search rules in such a way that they enforce a left-to-right evaluation order for an expression \(e_1 + e_2\). We do this by defining two separate rules. The first rule is

\[
\begin{align*}
  \text{env} \vdash e_1 \rightarrow e'_1 \\
  \text{env} \vdash e_1 + e_2 \rightarrow e'_1 + e_2
\end{align*}
\]

SearchPlus1

This rule takes an expression \(e_1 + e_2\), recursively applies a single simplification step in the subexpression \(e_1\), which yields a simpler subexpression \(e'_1\), and then returns the overall simpler expression \(e'_1 + e_2\). Note that the premise of the rule, \(\text{env} \vdash e_1 \rightarrow e'_1\), implies that \(e_1\) is not a value (because values cannot be further simplified).

Once \(e_1\) has been fully evaluated, a second search rule takes over to evaluate \(e_2\). This second rule is as follows:

\[
\begin{align*}
  \text{env} \vdash e_2 \rightarrow e'_2 \\
  \text{env} \vdash v_1 + e_2 \rightarrow v_1 + e'_2
\end{align*}
\]

SearchPlus2

Note that the rule only applies if the first subexpression is already a value. The premise of the rule again implies that \(e_2\) is not yet a value.
The complete evaluation of an expression \( e_1 + e_2 \) then works as follows. First, the rule \( \text{SearchPlus1} \) iteratively reduces \( e_1 \) to a value \( v_1 \), obtaining the expression \( v_1 + e_2 \). Then the rule \( \text{SearchPlus2} \) iteratively reduces \( e_2 \) to a value \( v_2 \), obtaining the expression \( v_1 + v_2 \). Finally, the rule \( \text{DoPlus} \) simplifies \( v_1 + v_2 \) to the sum of \( v_1 \) and \( v_2 \). Note that for any expression \( e_1 + e_2 \) only one of the three rules applies, i.e., the evaluation order is completely deterministic. This is in contrast to the big-step SOS, where the evaluation order was not fixed.

The semantics of the operator \( \times \) is captured by an according do rule called \( \text{DoTimes} \) and search rules that are almost identical to the rules \( \text{SearchPlus1} \) and \( \text{SearchPlus2} \). In fact, all binary operators except the short-circuiting ones are evaluated left-to-right. Hence, we can generalize the search rules \( \text{SearchPlus1} \) and \( \text{SearchPlus2} \) so that they also handle all these other operators:

\[
\frac{env \vdash e_1 \rightarrow e'_1}{env \vdash \text{bop} e_1 e_2 \rightarrow e'_1 e_2} \quad \text{SearchBop1}
\]

\[
\frac{bop \notin \{\&\&, ||\}}{env \vdash v_1 \text{bop} e_2 \rightarrow v_1 e'_2} \quad \text{SearchBop2}
\]

Note that the rule \( \text{SearchBop2} \) is restricted to non-short-circuiting operators. For the operators \( \&\& \) and \( || \) we have two additional do rules that reduce the expression to the second operand in the non-short-circuiting case:

\[
\frac{\text{true} = \text{toList}(v_1)}{env \vdash v_1 \&\& e_2 \rightarrow e_2} \quad \text{DoAndTrue}
\]

\[
\frac{\text{false} = \text{toList}(v_1)}{env \vdash v_1 || e_2 \rightarrow e_2} \quad \text{DoOrFalse}
\]

Due to these rules, we do not need search rules that reduce the second operand for \( \&\& \) and \( || \).

We do not discuss the do rules of the remaining binary operators as they are straightforward adaptations of the corresponding rules from the big-step SOS.

Next, we discuss the rules for constant declarations \( \text{const} x = e_d; e_b \). Again, we will need to split the evaluation into three rules, one do rule and two search rules. We start with the search rules. The first search rule handles the evaluation of the defining expression \( e_d \):

\[
\frac{env \vdash e_d \rightarrow e'_d}{env \vdash \text{const} x = e_d; e_b \rightarrow \text{const} x = e'_d; e_b} \quad \text{SearchConstDecl1}
\]

The second rule evaluates the body expression \( e_b \), but it only applies once evaluation of \( e_d \) has been completed:

\[
\frac{env' = env[x \mapsto v_d]}{env \vdash \text{const} x = e_d; e_b \rightarrow \text{const} x = v_d; e'_b} \quad \text{SearchConstDecl2}
\]

Note that this rule does not yet eliminate the actual declaration of \( x \), even though its defining expression has already been reduced to a value. We need to keep the declaration alive while \( e_b \) is still being reduced in order to be able to
compute the updated environment $env'$. The final elimination of the constant declaration is done by the following do rule:

\[
\begin{array}{c}
env \vdash \textbf{const } x = v_d; v_b \rightarrow v_b \\
\hline
\text{DoConstDecl}
\end{array}
\]

The rules for conditional expressions $e_1 ? e_2 : e_3$ are again straightforward.
Procedural Abstraction

Procedures are a central feature of almost all programming languages. A procedure allows us to abstract from the specifics of a particular computation by parameterizing over the specific values used in the evaluation of an expression. This way, we can describe similar computations without having to write the code for the common steps of these computations repeatedly in the program. Moreover, by allowing procedures to call themselves, we obtain recursion. As we have seen, recursion is the key to describing computations where the number of computation steps cannot be bounded a priori.

You will often here the terms *procedure* and *function* being used synonymously in programming language jargon. Some people consider *functions* to be the more general concept and think of a procedure as a special case of a function that has no return value. In the setting of functional programming languages, where every procedure has a return value, this distinction does not make sense. Here, some people use the term *procedure* for the general concept and call a procedure *function* only if it has no side effects. To avoid any confusions with our terminology, we will exclusively use the term *function*.

We will study functions in their most general form. Namely, we consider functions to be just another type of expressions in our language. That is, a function may occur anywhere in our program where an expression may occur. Specifically, we allow functions to occur both as arguments and return values of other functions. We will see that this generality enables some powerful programming techniques.

5.1 Functions and Function Calls

We extend our language with function expressions of the form:

\[
\text{function}(x) \ e
\]

We refer to such expressions as *anonymous functions* because the function expression itself is not bound to a name. We call \(x\) the *(formal) parameter* of
the function expression and \( e \) the body of the function expression. In order to be able to use functions, we additionally introduce function calls, \( e_1(e_2) \), where \( e_1 \) must evaluate to a function expression. We will use the term function application synonymously to function call.

Intuitively, a call expression \((\text{function}(x) e_1)(e_2)\) evaluates to expression \( e_1 \) with all free occurrences of \( x \) replaced by \( e_2 \). For example, consider the following (mathematical) function \textit{plusTwo}:

\[
\text{plusTwo} : \text{Num} \rightarrow \text{Num} \\
\text{plusTwo}(x) = x + 2
\]

suppose we want to express this function in our new language in order to compute the value \( \text{plusTwo}(3) = 5 \). We can do this in our extended language as follows:

```javascript
const plusTwo = function (x) { return x + 2; }
plusTwo(3)
```

Note that our concrete syntax for anonymous functions is slightly more compact than JavaScript’s syntax. In our more compact syntax, we take the value that results from the evaluation of the function body as the return value of the function, similar to Scala. In JavaScript, the return value of a function has to be made explicit using a \texttt{return} statement. Here is how the above program would look like in proper JavaScript:

```javascript
const plusTwo = function (x) {
    return x + 2;
};
plusTwo(3)
```

We will sometimes use the more verbose JavaScript syntax so that you can easily play with the code examples using a JavaScript interpreter.

\textbf{Remark.} With anonymous functions and function calls alone, we cannot express recursive functions directly. Surprisingly, this does not impose any restrictions on the expressivity of our language. That is, even without explicit recursive functions, we can still encode recursion indirectly. In fact, one of the fundamental results of computability theory states that any mathematical function that is computable (by a Turing machine) can be expressed in the subset of our new language that consists only of variables, function expressions, and function calls. This restricted subset of our language is referred to as the \textit{lambda calculus}. The lambda calculus was first introduced in 1936 by the logician Alonzo Church to prove that there exists no general algorithm for determining the truth value of a mathematical statement in arithmetic over natural numbers. He and Alan Turing later proved that Turing machines and the lambda calculus have the same computational expressivity. Due to the connection to the lambda calculus, function expressions are also often referred to as \textit{lambda abstractions}. 
For the convenience of writing recursive functions succinctly, we will also add named function expressions to our language. These take the form

\[ \text{function } x_1(x_2) e \]

where \( x_1 \) refers to the name of the function. For simplicity, the scope of \( x_1 \) is restricted to \( e \).

As an example of a use of recursive functions, consider the following JavaScript program, which defines a recursive function for computing the factorial numbers, and then computes the factorial of 5.

```javascript
function factorial(n) {
    return n > 1 ? n * factorial(n - 1) : 1
}
factorial(5)
```

Here is how we write this program in our language:

```javascript
const factorial =
    function factorial(n)
        (n > 1 ? n * factorial(n - 1) : 1);
factorial(5)
```

We assume that the additional operators > and - are interpreted as expected. Note that the name `factorial` is bound twice: once in the function body by the recursive function expression to obtain the recursive definition of the factorial function, and once by the `const` declaration to make it available for subsequent calls in the program (i.e., the body of the `const` declaration).

The abstract syntax of our new language is given by the following grammar. The new constructs are highlighted.

\[
x \in \text{Var} \quad \text{variables}
\]

\[
n \in \text{Num} \quad \text{numbers}
\]

\[
b \in \text{Bool} ::= \text{true} \mid \text{false} \quad \text{Booleans}
\]

\[
v \in \text{Val} ::= n \mid b \mid \text{function } p(x) e \mid \text{typeerror} \quad \text{values}
\]

\[
e \in \text{Expr} ::= v \mid x \mid e_1 \text{bop} e_2 \mid \text{const } x = e_d; e_b \quad \text{expressions}
\]

\[
\mid e_1 \text{ ? } e_2 : e_3 \mid e_1(e_2)
\]

\[
bop \in \text{Bop} ::= + \mid * \mid \&\& \mid || \mid === \mid !== \quad \text{binary operators}
\]

\[
p ::= x \mid \epsilon \quad \text{function names}
\]

Note that function expressions are values just like numbers and Booleans. We will see that this has some surprising consequences. Also, in our concrete syntax, function application is left-associative, i.e., the expression \( f(x)(y) \) is equivalent to \( (f(x))(y) \).

In addition to function expressions and function calls, we also introduce the value `typeerror`. This value is not part of the concrete syntax of our language. We will use this value to indicate a failure of evaluating certain expressions when we define the operational semantics of our new language constructs.
5.2 Currying and Partial Function Application

In our new language, a function expression \( \text{function}(x) e \) only allows a single parameter \( x \). At first, this seems too restrictive because we cannot write functions that take more than one parameter. For example, consider the following binary function \( \text{plus} \) that takes two numbers and adds them:

\[
\text{plus} : \text{Num} \times \text{Num} \to \text{Num} \\
\text{plus}(x, y) = x + y
\]

Due to the restriction on the number of parameters of function expressions, we cannot express the function \( \text{plus} \) directly in our language. The trick to overcome this problem is that we redefine \( \text{plus} \) as a cascaded function that first takes \( x \) and then takes \( y \), instead of taking both values simultaneously:

\[
\text{plus} : \text{Num} \to (\text{Num} \to \text{Num}) \\
\text{plus}(x)(y) = x + y
\]

Observe that \( \text{plus} \) is now a function that takes a number \( x \) and returns another function. This returned function takes a number \( y \) and returns a number, namely, \( x + y \). The transformation of functions that take more than one parameter into cascaded functions that each take one parameter is referred to as \textit{currying}.\footnote{The term \textit{currying} is derived from the name of the logician Haskell Curry, who popularized this technique.} We also call the second version of \( \text{plus} \) a \textit{curried function}. The grammar of our language allows the body \( e \) of a function expression \( \text{function}(x) e \) to be itself a function expression. Thus, we can express the curried version of \( \text{plus} \) in our language as follows:

\[
\text{const plus = function } (x) \text{ (function } (y) (x + y)); \\
\text{plus}(2)(3)
\]

This program will evaluate to 5 as expected.

One useful feature of curried functions is that we can apply them partially. For example, we can express the function \( \text{plusTwo} \) using the function \( \text{plus} \) as follows:

\[
\text{const plus = function } (x) \text{ (function } (y) (x + y)); \\
\text{const plusTwo = plus}(2); \\
\text{plusTwo}(3) + \text{plusTwo}(2)
\]

This program will evaluate to 9.

5.3 Operational Semantics of Function Calls

We now formalize the operational semantics of our extended language. We will focus on the big-step SOS. Once the rules for the big-step semantics are
understood, it is straightforward to adapt and extend the inference rules for the small-step semantics.

We discuss two different semantics of function calls: one with dynamic binding and one with static binding. We start with the former as it is slightly simpler to implement.

5.3.1 Environment-based Semantics with Dynamic Binding

First, note that we defined function expressions as values of our language. That is, by our definition of values, a function expression \( \text{function } p(x) \ e \) should not be further evaluated. In fact, from our old rule EvalVal for evaluating values, we immediately obtain

\[
\text{env} \vdash \text{function } p(x) \ e \downarrow \text{function } p(x) \ e
\]

For a function to have any computational effect, we must call it. Thus, we have to provide the actual inference rules that provide the semantics for function calls. We first handle the case of a call expression \( e_1(e_2) \) where \( e_1 \) evaluates to an anonymous function:

\[
\begin{align*}
\text{env} & \vdash e_1 \downarrow v_1 \\
v_1 &= \text{function}(x) \ e \\
\text{env} & \vdash e_2 \downarrow v_2 \\
\text{env}' &= \text{env}[x \mapsto v_2] \\
\text{env}' &\vdash e \downarrow v \\
\text{env} &\vdash e_1(e_2) \downarrow v
\end{align*}
\]

The rule follows our intuition of how a function call \( e_1(e_2) \) should work. We first evaluate the callee \( e_1 \) and only if the resulting value \( v_1 \) is a function expression \( \text{function}(x) \ e \) can we continue. Then we evaluate the argument expression \( e_2 \) of the call, yielding value \( v_2 \). Next, we bind the parameter \( x \) to the argument \( v_2 \) to obtain a new environment \( \text{env}' \). Finally, we evaluate the body, \( e \), of the function under the new environment to obtain the final result value \( v \) of the call.

Note that the rule does not actually specify the order in which the callee \( e_1 \) and its argument \( e_2 \) are evaluated. The actual evaluation order of call expressions in JavaScript is left-to-right. However, we need to switch to a small-step SOS to be able to capture this detail.

The rule EvalCall captures one detail in the execution order of call expressions: the argument \( e_2 \) is evaluated before we evaluate the body expression \( e \). This semantics of function calls is referred to as call-by-value. This is opposed to a semantics where \( e_2 \) is evaluated each time \( x \) is used inside of \( e \). We refer to the latter form as call-by-name semantics. In Chapter 7, we will discuss the different variants of function call semantics in more detail.

The generalization of the rule EvalCall to calls that involve recursive functions is now almost obvious:

\[
\begin{align*}
\text{env} &\vdash e_1 \downarrow v_1 \\
v_1 &= \text{function } x_1(x_2) \ e \\
\text{env} &\vdash e_2 \downarrow v_2 \\
\text{env}' &= \text{env}[x_1 \mapsto v_1][x_2 \mapsto v_2] \\
\text{env}' &\vdash e \downarrow v \\
\text{env} &\vdash e_1(e_2) \downarrow v
\end{align*}
\]

EvalCallRec
In addition to binding the parameter \( x_1 \) to the argument value \( v_2 \) of the call, we also bind the name of the function \( x_1 \) to the function expression \( v_1 \) itself. Any recursive calls to \( x_1 \) inside of \( e \) will then be properly resolved during evaluation.

### 5.3.2 Dynamic Type Errors

Before we added functions and function calls to our language, all expressions could be evaluated to some value (because of automatic type conversion). With functions, we encounter one of the very few run-time errors in JavaScript: trying to call something that is not a function. In JavaScript, calling a non-function raises a run-time error. In the formal semantics, we model this with evaluating to the “marker” \( \text{typeerror} \):

\[
\begin{align*}
env \vdash e_1 \downarrow v_1 & \quad v_1 \neq \text{function } p(x) \ e \\
env \vdash e_1(e_2) \downarrow \text{typeerror}
\end{align*}
\]

Such a run-time error is known as dynamic type error. Languages are called dynamically typed when they allow all syntactically valid programs to run and only check for type errors during program execution.

Once we have encountered a dynamic type error somewhere during evaluation, there is no way to recover from it. We model this by adding propagation rules that propagate type errors encountered in subexpressions to the top-level. For example, the following rule propagates a type error while evaluating the defining expression in a constant declaration:

\[
\begin{align*}
env \vdash e_d \downarrow \text{typeerror} \\
env \vdash \text{const } x = e_d; e_b \downarrow \text{typeerror}
\end{align*}
\]

The complete set of type error (propagation) rules is summarized in Figure 5.2.

The other rules of the big-step semantics are given in Figure 5.1. Note that in the rules \( \text{EvalEqual} \) and \( \text{EvalDisequal} \), we disallow equality and disequality checks (i.e., \( == \) and \( !== \)) on function expressions. If either argument to these two operators is a function expression, then we consider this a dynamic type error. This design decision is reflected by the type error rules \( \text{EvalTypeErrorEqual}_1 \) and \( \text{EvalTypeErrorEqual}_2 \) of Figure 5.2. The choice of disallowing equality between functions is a slight departure from the semantics of JavaScript. In JavaScript, equality between functions is reduced to equality of pointers to memory locations where the implementations of the involved functions are stored. We cannot model this notion of equality faithfully in our current semantics because we don’t have an explicit memory (yet).

### 5.3.3 Dynamic vs. Static Binding

Recall that when we defined the binding structure and scoping rules of \texttt{const} declarations \( \text{const } x = e_d; e_b \), we required that every free using occurrence of \( x \) in \( e_b \) should be bound to the value of \( e_d \). We referred to this type of binding as static binding. That is, by looking at the syntactic structure of a program we can
Figure 5.1: Inference rules that define the big-step SOS of our language with functions. The modifications and additions to the rules from Figure 4.1 in Section 4.1.1 are highlighted.
Figure 5.2: Big-step operational semantics: dynamic type error rules
determine which value each variable refers to. Unfortunately, with the big-step semantics provided above, we no longer correctly implement static binding.

In order to understand this issue, consider the following program:

```plaintext
1  const x = 2;
2  const plustwo = function (y) (x + y);
3  const f = function (x) (plustwo(x));
4  f(3)
```

With static binding, the using occurrence of `x` on line 2 should always refer to the defining occurrence of `x` on line 1. That is, this occurrence of `x` should always evaluate to 2. We would therefore expect that the program above evaluates to 5. However, if we evaluate the program according to the rules of our big-step semantics, we obtain the value 6.

The problem is the rule `EvalCall`. When we evaluate the call `f(3)` then the parameter `x` of `f` is bound to the value 3, overwriting the current binding of `x` to 2. Thus, when we subsequently look-up the value of `x` in the updated environment during the call to `plustwo` inside of `f`, we retrieve 3 instead of 2.

We refer to this semantics of variable bindings as *dynamic binding*. The problem with dynamic binding is that the actual binding of variables to values depends on how a program is evaluated. It cannot be derived statically from the syntactic structure of the program. Consequently, programs that are evaluated under dynamic binding semantics often have unintuitive behavior that is difficult to debug. All modern programming languages therefore use static binding instead of dynamic binding by default. The use of dynamic binding in some early programming languages is now considered a historical mistake.

The problem with dynamic binding is that before the call to `plustwo` on line 3 is evaluated, the variable `x` is rebound to the value 3 that is passed into `f` on line 4. Thus, we evaluate the body of `plustwo` in the environment `[x ↦ 3, y ↦ 3]`, yielding 6 as the overall result of the evaluation.

We now modify our operational semantics so that we obtain the desired static binding semantics.

### 5.3.4 Substitution-based Semantics with Static Binding

We consider a simple substitution-based semantics with static binding. This semantics is not used in practice, because it is computationally inefficient. However, it is easy to understand and implement. In the following, we first use a small-step semantics to introduce the intuition behind substitution-based semantics but only provide the inference rules for a big-step semantics. However, everything we discuss equally works for a small-step and a big-step semantics.

For our new semantics, we assume that we start our evaluation with an expression `e` that is closed. That is, `e` does not contain any free variables: `fv(e) = ∅`. The semantics then maintains this property throughout the entire evaluation of `e`, i.e., during evaluation we will only have to deal with closed expressions. The new semantics will therefore no longer need an environment
to store the values bound to the free variables that occur in the evaluated expression.

In order to avoid the introduction of free variables during the evaluation of subexpressions, we have to eliminate bound variables as soon as evaluation passes through a variable binding construct, such as `const` declarations or function expressions. These are the points during evaluation where substitutions come into play. For example, when we now evaluate a `const` declaration `const x = e_d; e_b`, we first evaluate the defining expression `e_d` of `x` as before to obtain a result value `v_d`. Though, we no longer store the binding of `v_d` to `x` in an environment for later retrieval during the evaluation of the body expression `e_b`. Instead, we use a substitution `e_b[v_d/x]` to replace all free occurrences of `x` in `e_b` in a single step. Then we continue evaluation with the expression resulting from the substitution. This expression is again closed. Moreover, the substitution realizes the intended static binding semantics of variables.

We demonstrate this using the example program from the previous section:

```plaintext
const x = 2;
const plusTwo = function (y) (x + y);
const f = function (x) (plusTwo(x));
f(3)
```

The evaluation of this program with the substitution-based small-step semantics now looks as follows:

```plaintext
const x = 2;
const plusTwo = function (y) (x + y);
const f = function (x) (plusTwo(x));
f(3)
```

```plaintext
-> const plusTwo = function (y) (2 + y);
const f = function (x) (plusTwo(x));
f(3)
```

```plaintext
-> const f = function (x) (((function (y) (2 + y))(x))(3)
```

```plaintext
-> (function (x) (((function (y) (2 + y))(x))(3)
```

```plaintext
-> 2 + 3
```

```plaintext
-> 5
```

Note that the final value that we compute is 5 instead of 6, which is the value that we obtained with the dynamic-binding semantics.

Let us now formalize the substitution-based semantics in terms of inference rules. We do this using a big-step semantics. The most drastic change compared to the dynamic binding semantics that we discussed in Section 5.3.1 is that we eliminate value environments from the evaluation relation. That is, we define a new relation `e ↓ v` that relates expressions and values directly. For this to work,
we must require that the expressions $e$ that we consider for evaluation are all closed, i.e., they contain no free variables $fv(e) = \emptyset$.

The rules that define the new big-step semantics are summarized in Figures 5.3 and 5.4. The crucial changes are in the rules EvalConstDecl, EvalCall, and EvalCallRec, i.e., the rules that previously modified the environment by introducing new variable bindings. Moreover, the rule EvalVar for evaluating free variables is no longer needed, since we do not have free variables in the evaluated expressions by assumption. The remaining rules, which used to simply pass the environment along for evaluating subexpressions are almost unchanged. We only removed the environments. We discuss the rules EvalConstDecl, EvalCall, and EvalCallRec in detail, starting with the rule EvalConstDecl:

$$
\frac{e_d \Downarrow v_d \quad e'_b = e_b[v_d/x] \quad e'_b \Downarrow v}{\text{EvalConstDecl}}
$$

As before, the rule evaluates the defining expression $e_d$, obtaining some value $v_d$. Now, recall that the old rule bound $x$ to $v_d$ in an environment to evaluate the body $e_b$ of the const declaration. Instead, the new rule simply calculates a new expression $e'_b$ that is obtained from $e_b$ by substituting all free occurrences of $x$ in $e_b$ by the value $v_d$. We can think of this substitution as performing, in a single step, all recursive applications of the rule EvalVar when evaluating $e_b$ in the old environment-based semantics. Recall from Section 3.3.3 that we defined the substitution $e_b[v_d/x]$ in such a way that it respects the static binding structure of the expression. That is, only the free occurrences of $x$ in $e_b$ will be substituted. This way, we obtain the desired static binding semantics.

The rules EvalCall and EvalCallRec for evaluating function calls are adapted in a similar fashion:

$$
\frac{e_1 \Downarrow v_1 \quad v_1 = \text{function}(x) e \quad e_2 \Downarrow v_2 \quad e' = e[v_2/x] \quad e' \Downarrow v}{\text{EvalCall}}
$$

$$
\frac{e_1 \Downarrow v_1 \quad v_1 = \text{function}(x_1(x_2)) e \quad e_2 \Downarrow v_2 \quad e' = e[v_2/x_2][v_1/x_1] \quad e' \Downarrow v}{\text{EvalCallRec}}
$$

Exercise 5.1. Reconsider the problematic example program:

```plaintext
const x = 2;
const plusTwo = function(y) (x + y);
const f = function(x) (plusTwo(x));
f(3)
```

Verify that this program indeed evaluates to 5 under the new substitution-based big-step semantics.
Figure 5.3: Inference rules that define the substitution-based big-step SOS for static binding semantics of our language with functions.
\[ e_1 \downarrow \text{function } p(x) e \quad \text{bop} \in \{==, \neq\} \quad \text{EvalTypeErrorMessage}_{\text{EQUAL}_1} \]

\[ e_2 \downarrow \text{function } p(x) e \quad \text{bop} \in \{==, \neq\} \quad \text{EvalTypeErrorMessage}_{\text{EQUAL}_2} \]

\[ e_1 \downarrow v_1 \quad v_1 \neq \text{function } p(x) e \quad \text{EvalTypeErrorMessage}_{\text{CALL}} \]

\[ e_1 \downarrow \text{typeerror} \quad \text{EvalPropIf} \quad e_1 \downarrow \text{typeerror} \quad \text{EvalPropBop}_1 \]

\[ e_2 \downarrow \text{typeerror} \quad \text{EvalPropBop}_2 \quad e_1 \downarrow \text{typeerror} \quad \text{EvalPropBop}_2 \]

\[ e_1 \downarrow v_1 \quad v_1 \neq \text{typeerror} \quad \text{true} = \text{toBool}(v_1) \quad e_2 \downarrow \text{typeerror} \quad \text{EvalPropAnd} \]

\[ e_1 \downarrow v_1 \quad v_1 \neq \text{typeerror} \quad \text{false} = \text{toBool}(v_1) \quad e_2 \downarrow \text{typeerror} \quad \text{EvalPropOr} \]

\[ e_1 \downarrow \text{typeerror} \quad \text{EvalPropCall}_1 \quad e_2 \downarrow \text{typeerror} \quad \text{EvalPropCall}_2 \]

\[ e_1 \downarrow \text{typeerror} \quad \text{EvalPropCall}_2 \quad e_2 \downarrow \text{typeerror} \quad \text{EvalPropCall}_2 \]

\[ e_1 \downarrow \text{typeerror} \quad \text{EvalPropConstDecl} \]

Figure 5.4: Substitution-based big-step operational semantics: dynamic type error rules
5.4 Higher-Order Functions

In Section 5.2 we have seen that the body of a function expression can itself be a function expression. This observation led us to the notion of curried functions. What about call expressions $e_1(e_2)$? We require that $e_1$ must be a function expression for a call expression to be meaningful. However, we impose no syntactic constraints on the argument expression $e_2$ of the call. In particular, we allow $e_2$ to be again a function expression. We refer to functions that can be applied to other functions as *higher-order functions*. If you want to become a better programmer, learning how to use higher-order functions effectively is an important step.

5.4.1 Abstracting from Computations

We start with a simple example to motivate the usefulness of higher-order functions. Suppose that we want to write a function `sumInts` that takes the bounds $a$ and $b$ of a (half-open) interval of integer numbers $[a, b]$ and computes the sum of the values in that interval. For example, `sumInts(1)(4)` should yield 6. The following recursive implementation does what we want:

```plaintext
function sumInts(a) {
    return function(b) {
        return a < b ? a + sumInts(a+1)(b) else 0;
    }
}
```

Now, consider the following function `sumSqr` that computes the sum of the squares of the numbers in an interval $[a, b]$:

```plaintext
function sumSqr(a) {
    return function(b) {
        return a < b ? a * a + sumSqr(a+1)(b) else 0;
    }
}
```

The functions `sumInts` and `sumSqr` are almost identical. They only differ in the summand that is added in each recursive call. In the case of `sumInts` it is $a$, and in the case of `sumSqr`, it is $a \cdot a$. We can write a higher-order function `sum` that abstracts from these differences. The function `sum` takes another function `f` as additional parameter. The function `f` captures the computation that is performed in the summand:

```plaintext
function sum(f) {
    return function(a) {
        return function(b) {
            a < b ? f(a) + sum(f)(a + 1)(b) else 0
        }
    }
}
```
Now we can define `sumInts` and `sumSqr` more succinctly in terms of `sum` as follows:

```javascript
const sumInts = sum(function(a) { return a; });
const sumSqr = sum(function(a) { return a * a; });
```

### 5.4.2 Realizing for Loops with Higher-Order Functions

The combination of curried functions, partial application, and higher-order functions yields an extremely powerful programming technique that often allows us to express programs much more succinctly than in traditional imperative programming languages such as Java. In the following, we discuss an example that uses higher-order functions and currying to extend our language with a loop construct.

Consider the following JavaScript program, which uses a `for` loop over an accumulator variable `acc` to compute the factorial of 5:

```javascript
function factorial(n) {
  var acc = 1;
  for (i = 1; i <= n; i++) {
    acc = acc * i;
  }
  return acc;
}
factorial(5)
```

While our language provides neither `for` loops nor mutable `var` variables, we can faithfully encode the above implementation of factorial using a curried tail-recursive function:

```javascript
const factorial = function(n) {
  function loop(i) {
    function (acc) {
      i <= n ? loop(i + 1)(acc * i) : acc
    }
  }(1)(1);
}
factorial(5)
```

Since `for` loops are such a useful programming construct, we might want to make them available more generally in our language. We can do this by taking the function `loop` out of the body of `factorial` and abstracting over the actual computation performed by that function:

```javascript
const for =
  function (n) {
    function (body) {
      function (init) {
        function loop (i) {
          ```
The function \texttt{for} is a curried higher-order function. The expression

\[
\texttt{for}(n)(\texttt{body})(\texttt{init})
\]

computes the value that is stored in the variable \texttt{acc} when the following JavaScript program is executed:

\[
\texttt{var acc = init; for (i = 1; i <= n; i++) { acc = body(i)(acc); }}
\]

5.5 Higher-Order Functions and Collections in Scala

Higher-order functions provide a powerful mechanism for abstracting over common computation patterns in programs. This mechanism is particularly useful for designing libraries with rich interfaces that support callbacks to client code. We will study these mechanisms using the example of Scala’s collection libraries.

5.5.1 Higher-Order Functions in Scala

Before we dive into the intricacies of Scala collections, let us first see how higher-order functions are defined in Scala. To this end, we revisit the \texttt{sum} function from Section 5.4.1 Here is how we can define this function in Scala:

\[
\texttt{def sum(f: Int} 
\texttt{ => Int, a: Int, b: Int) = { if (i < b) f(a) + sum(f, a + 1, b) else 0 }}
\]

The \texttt{function type Int} \texttt{=} \texttt{Int} of the parameter \texttt{f} indicates that \texttt{f} is a function that takes a value of type \texttt{Int} and maps it again to an \texttt{Int}.

We can now define the function \texttt{sumSqs} by first defining a function \texttt{square} that squares an integer number, and then applying \texttt{sum} to \texttt{square}:
def square(i: Int) = i * i
def sumSqrs(a: Int, b: Int) = sum(square, a, b)

In order to make the use of higher-order functions more convenient, Scala supports writing anonymous functions (aka function literals), similar to JavaScript. In Scala, anonymous functions take the general form:

(x1: T1, ..., xn: Tn) => body

where the xi are the parameters of the function, the Ti are the associated types, and body is the body of the function. We can thus define the functions sumInts and sumSqrs using anonymous functions as follows:

def sumInts(a: Int, b: Int) = sum((i: Int) => i, a, b)
def sumSqrs(a: Int, b: Int) = sum((i: Int) => i * i, a, b)

### 5.5.2 Curried Functions in Scala

Reconsider our definition of sumInts and sumSqrs in terms of sum:

```scala
def sumInts(a: Int, b: Int) = sum((i: Int) => i, a, b)
def sumSqrs(a: Int, b: Int) = sum((i: Int) => i * i, a, b)
```

One annoyance with these definitions is that we have to redeclare the parameters a and b which are simply passed to sum. We can avoid this by redefining sum as a curried function that first takes the function f applied to the values in the interval and then returns a function that takes the bounds of the interval a and b.

There are various ways to define curried functions in Scala. One way is to define the nested function explicitly by name using a nested `def` declaration and then returning that function:

```scala
def sum(f: Int => Int): (Int, Int) => Int = {
  def sumHelp(a: Int, b: Int): Int = {
    if (a < b) f(i) + sum(f)(a+1, b) else 0
  }
  sumHelp
}
```

Using the curried version of sum the definition of sumInts and sumSqrs can be simplified like this:

```scala
def sumInts = sum((i: Int) => i)
def sumSqrs = sum((i: Int) => i * i)
```

Note that when we apply curried higher-order functions to anonymous functions, then the compiler can often infer the parameter types. This simplifies the definitions even further:

```scala
def sumInts = sum(i => i)
def sumSqrs = sum(i => i * i)
```
In our curried version of \texttt{sum}, the function \texttt{sumHelp} is not recursive and is directly returned after being declared. We can thus simplify the definition of \texttt{sum} further by turning \texttt{sumHelp} into an anonymous function:

\begin{verbatim}
def sum(f: Int => Int)(a: Int, b: Int): Int = 
  if (a < b) f(a) + sum(f)(a+1, b) else 0
\end{verbatim}

Since curried functions are so common in functional programs, the Scala language provides a special syntax for them. Instead of nesting the function declarations, we can write a curried function by providing the parameters of the nested function in a separate parameter list:

\begin{verbatim}
def sum(f: Int => Int)(a: Int, b: Int): Int = 
  if (a < b) f(a) + sum(f)(a, b) else 0
\end{verbatim}

If we partially apply a curried function written in this form, we have to make this explicit by appending the underscore symbol \_ to the partial application. The definitions of \texttt{sumInts} and \texttt{sumSqrS} thus look as follows in this case:

\begin{verbatim}
def sumInts = sum(i => i)
def sumSqrS = sum(i => i * i)
\end{verbatim}

### 5.5.3 Higher-Order Functions on Lists

An important use case of higher-order functions is to realize callbacks to client code from within library functions. We discuss this scenario using the specific example of the class \texttt{List} in the Scala standard library.

Recall the list data structure that we studied in Section 2.2. We can define lists of integers recursively using an algebraic data type as follows:

\begin{verbatim}
sealed abstract class List
case object Nil extends List
  case class Cons(hd: Int, tl: List) extends List
\end{verbatim}

That is, a list is either empty, denoted by \texttt{Nil}, or a cons cell consisting of an integer \texttt{hd} and the tail of the list \texttt{tl}.

The generic class \texttt{List[A]} in the Scala standard library generalizes this data structure to lists over an arbitrary element type \texttt{A}. The empty list is also denoted by \texttt{Nil} and a cons cell is denoted \texttt{hd :: tl}. We can thus construct lists as follows:

\begin{verbatim}
scala> val l1 = 1 :: (4 :: (2 :: Nil))
l1: List[Int] = List(1, 4, 2)

scala> val l2 = "apple" :: ("banana" :: Nil)
l2: List[String] = List(apple, banana)
\end{verbatim}

Note that the infix cons operator :: is right-associative, so the parenthesis in the above example can be omitted:
As expected, we can use pattern matching to deconstruct lists into their components:

```scala
scala> val h :: t = l1
h: Int = 1
t: List[Int] = List(4, 2)

scala> l match {
  case Nil => println("l is empty")
  case hd :: tl => println(s"l's head is $hd")
}
l's head is 1.
```

Here is how we can define the `concat` function from Section 2.2 using Scala’s `List` class:

```scala
def concat[A](l1: List[A], l2: List[A]): List[A] = 
  l1 match {
    case Nil => l2
    case hd :: tl => hd :: concat(tl, l2)
  }
```

Note that `concat` is a `generic` function that is parameterized by the type `A` of the elements stored in the lists. Here is an example of how to use `concat`:

```scala
scala> concat(List(3, 4, 1), List(2, 6))
res0: List[Int] = List(3, 4, 1, 2, 6)
```

Here is how we can define the tail-recursive version of the `reverse` function:

```scala
def reverse[A](l: List[A]): List[A] = {
  def rev(l: List[A], acc: List[A]): List[A] = 
    l match {
      case Nil => acc
      case hd :: tl => rev(tl, hd :: acc)
    }
    rev(l, Nil)
}
```

```scala
scala> reverse(List(3, 4, 1, 2))
res0: List[Int] = List(2, 1, 4, 3)
```

From the above examples we can see that functions operating on lists follow a common pattern: they traverse the list, decomposing it into its elements, and then apply some operation to each of the elements. We can extract this common pattern and implement them in more general higher-order functions that abstract from the specific operations being performed.
A particularly common operation on lists is to traverse a list and applying some function to each element, obtaining a new list. For example, suppose we have a list of `Double` values which we want to scale by a given factor to obtain a list of scaled values. The following function implements this operation:

```scala
def scale(factor: Double, l: List[Double]): List[Double] = 
  l match {
    case Nil => Nil
    case hd :: tl => factor * hd :: scale(factor, tl)
  }
```

A similar operation is implemented by the following function, which takes a list of integers and increments each element to obtain a new list:

```scala
def incr(l: List[Int]): List[Int] = 
  l match {
    case Nil => Nil
    case hd :: tl => hd + 1 :: incr(tl)
  }
```

The type of operation that is performed by `scale` and `incr` is called a `map`. We can implement the map operation as a higher-order function that abstracts from the concrete operation that is applied to each element in the list:

```scala
def map[A, B](l: List[A])(op: A => B): List[B] = 
  l match {
    case Nil => Nil
    case hd :: tl => op(hd) :: map(tl)(op)
  }
```

The function `map` is parametric in both the element type `A` of the input list, as well as the element type `B` of the output list. That is, a map operation transforms a `List[A]` into a `List[B]` by applying an operation `op: A => B` to each element in the input list. Note that the order of the elements in the input list is preserved.

We can now redefine `scale` and `incr` as instances of `map`:

```scala
def scale(factor: Double, l: List[Double]) = 
  map(l)(x => factor * x)
def incr(l: List[Int]) = map(l)(x => x + 1)
```

Note that Scala provides a syntactic short form for defining anonymous functions by replacing variables in expressions with underscores. This notation is often useful to obtain succinct code when using higher-order functions. For example, the Scala compiler will automatically expand the following code to the above definitions of `scale` and `incr`:

```scala
def scale(factor: Double, l: List[Double]) = 
  map(l)(factor * _)
def incr(l: List[Int]) = map(l)(_ + 1)
```
5.5.4 Folding Lists

We have seen that we can often identify common patterns in functions on data structures and implement them in generic higher-order functions. We can then conveniently reuse these generic functions, reducing the amount of code we have to write. In this section, we will look at the most general patterns for performing operations on collections, namely fold operations.

As a motivating example, consider the following function, which computes the sum of the values stored in a list of integers:

```scala
def sumHl: list[Int] => Int = {
  case Nil => 0
  case hd :: tl => hd + sumHtlI
}
```

Consider a list `l` of `n` integer values:

```
d_1 :: d_2 :: ... :: d_n :: Nil
```

Then unrolling the recursion of `sum` on `l` yields the following computation:

```
d_1 + (d_2 + ... (d_n + 0)...)
```

That is, in the `i`th recursive call, we add the current head `d_i` to the sum of the values in the current tail, where we consider the sum of an empty list `Nil` to be 0. If we represent this computation as a tree, this tree looks as follows:

```
                   +
                  /   
                 +   
                /   
               +   
              /   
             +   z
```

We can now generalize from the specific computation performed by the represented expression. That is, in the general case we have a list of values of type `A` instead of `Int`. Then, instead of the specific initial value `0` for the empty list, we are given an initial value `z` of some type `B`. Finally, instead of adding the current head to the sum of the current tail of the list, we apply a generic operation `op` in each step. The operation `op` takes the current value `d_i`, which is of type `A`, and the result of the computation on the tail, which is of type `B`, and returns again a value of type `B`. The resulting expanded recursive computation is then represented by the following tree:

```
                   op
                  /   
                 op   
                /   
               op   
              /   
             op   z
```

The operation `op` takes the current value `d_i`, which is of type `A`, and the result of the computation on the tail, which is of type `B`, and returns again a value of type `B`. The resulting expanded recursive computation is then represented by the following tree:
We refer to this type of computation as a *fold* of the list because the list is traversed and recursively folded into a single value. Note that the tree is leaning towards the right. We therefore refer to this type of fold operation as a *fold-right*. That is, the recursive computation is performed in right-to-left order of the values stored in the list.

The following higher-order function implements the fold-right operation:

```scala
def foldRight[A, B](l: List[A])(z: B)(op: (A, B) => B): B = l match {
  case Nil => z
  case hd :: tl => op(hd, foldRight(tl)(z)(op))
}
```

We can now redefine `sum` in terms of `foldRight`:

```scala
def sum(l: List[Int]): Int = foldRight(l)(0)(_ + _)
```

Many of the other functions that we have seen before perform fold-right operations on lists. In particular, we can define `concat` using `foldRight` as follows:

```scala
def concat[A](l1: List[A], l2: List[A]): List[A] = foldRight(l1)(l2)(_ :: _)
```

Even the higher-order function `map` is just a special case of a fold-right:

```scala
def map[A, B](l: List[A])(op: A => B): List[B] = foldRight(l)((Nil: List[B]))((h, l) => op(h) :: l)
```

Note that due to limitations of Scala’s type inference algorithm, we have to manually annotate the type `List[B]` of the empty list constructor `Nil` that we use to build the resulting list of the map operation.

All the above operations on lists have in common that they combine the elements in the input list and the result of the recursive computation in right-to-left order. We can also consider fold operations that perform the computation in reverse order:

```scala
op(op(...(op(op(z, d1), d2), ...), d_{n-1}), d_n)
```

The corresponding computation tree then looks as follows:

```
  op
 /   \
op   d_n
 |
 op   d_{n-1}
 |         
 op   d_2
 |      
 op   d_1
 |    
 z
```

Note that the tree is now leaning towards the left and the elements are combined in left-to-right order. We therefore refer to this type of computation as a *fold-left*. The following function implements the generic fold-left operation on lists:
def foldLeft[A,B](l: List[A])(z: B)(op: (B, A) => B): B = 
  l match { 
    case Nil => z 
    case hd :: tl => foldLeft(tl)(op(z, hd))(op) 
  }

Since addition is associative and commutative, we can alternatively define `sum` using `foldLeft` instead of `foldRight`:

```scala
def sum(l: List[Int]): Int = foldLeft(l)(0)(_ + _)
```

In fact, this definition of `sum` is more efficient than our previous implementations because `foldLeft` is tail-recursive, whereas our implementation of `foldRight` is not. In general, only one of the two types of fold operations can be used to implement a specific computation on lists. For example, we can express `reverse` in terms of a fold-left as follows:

```scala
def reverse[A](l: List[A]): List[A] = 
  foldLeft(l)((Nil: List[A]))((l1, x) => x :: l1)
```

If we replaced `foldLeft` by `foldRight` in this definition, we would not obtain the correct result. The computed output list would be structurally identical to the input list.

### 5.5.5 Scala’s Collection Classes

Since higher-order functions on collections are so incredibly useful for writing concise code, the data structures in the Scala standard API already implement a large number of these functions. The functions are realized as methods of the corresponding collection classes. For example Scala’s `List` class already provides the methods `foldLeft`, `foldRight`, `map`, etc.

As with any programming language, you should study the Scala standard API carefully so that you get an overview of the provided functionality and so that you do not “reinvent the wheel” when you write your own programs.

To get a glimpse of the expressive power of the functions implemented in the collection classes, consider the following code snippet. The code defines a list of integers and a list of strings and then folds the two lists into a single string. This string consists of a comma separated sequence of strings, where each string is a pair of elements from the two lists concatenated together using the colon symbol:

```scala
scala> val l1 = List(1, 2, 3)
l1: List[Int] = List(1, 2, 3)

scala> val l2 = List("a", "b","c")
l2: List[String] = List(a, b, c)

scala> (l1,l2) zipped map (_ + ":" + _) reduce
```
It is instructive to re-implement this code snippet in a language such as Java to appreciate how much more concise and comprehensive the functional implementation is.
Chapter 6

Types

Types play an important role in many programming languages. In particular, types are used to detect common programming errors statically at compile-time, i.e., before the program is even executed. In this chapter, we will study a variant of the JavaScript fragment from Chapter 5 and extend it with a simple type system. We will develop a type inference algorithm that allows us to enforce statically that the evaluation of a given program will not produce a dynamic type error.

6.1 Type Checking and Type Inference

As we have seen in the prior chapters, dealing with implicit type conversion and checking for dynamic type errors complicates the operational semantics of a language and, in turn, the interpreter implementation. Some languages restrict the possible programs that can be executed to those that are guaranteed not to result in a dynamic type error. This restriction of programs is enforced with an analysis phase after parsing known as type checking. If a language guarantees that any program that passes the type checker will not produce a dynamic type error, we call this language type safe. In this section, we will study a type safe variant of JakartaScript.

6.1.1 Type Checking

The following expressions in our language from Chapter 5 will result in a dynamic type error during evaluation:

\[
\begin{align*}
(3 + 4)(0) \\
1 \times 3 & \text{ == function}(x) 0
\end{align*}
\]

The first expression will fail when we try to evaluate the call because the callee expression \((3 + 4)\) does not evaluate to a function. The second expression fails because our operational semantics disallows comparisons that involve function
values. The goal of static type checking is to identify such expressions before we actually evaluate them. This way we can statically detect certain programming errors and provide a correctness guarantee for the evaluation, namely, that the evaluation will never “get stuck” and produce a dynamic type error. We refer to this feature of a programming language as a static type system.

Unfortunately, the problem of checking whether the evaluation of a given expression in our language will result in a dynamic type error is undecidable. In fact, this is true for any Turing-complete programming language. There are two ways to deal with this dilemma: (1) we can require that the programmer provides some help to the static type checking algorithm by annotating the program with type information and (2) we can allow the type checker to reject certain expressions as unsafe even though the expression could be safely evaluated. The type systems of most statically typed programming languages use a combination of (1) and (2). The crux in designing a static type system is to find a good balance between the annotation burden for the programmer, the restrictions imposed on what programs are considered safe, and the computational complexity of the actual type checking algorithm. We will explore some of these design choices.

6.1.2 A Simple Typed Language

We extend our language from Chapter 5 with types. We choose syntax for types that is inspired by the TypeScript language, a typed extension of JavaScript. The abstract syntax of our new language is as follows:

\[
\begin{align*}
n &\in \text{Num} \quad \text{numbers (double)} \\
x &\in \text{Var} \quad \text{variables} \\
b &\in \text{Bool} ::= \text{true} \mid \text{false} \quad \text{Booleans} \\
\tau &\in \text{Typ} ::= \text{bool} \mid \text{number} \mid \tau_1 \Rightarrow \tau_2 \quad \text{types} \\
v &\in \text{Val} ::= n \mid b \mid \text{function } p(x: \tau) \ t \ e \quad \text{values} \\
e &\in \text{Expr} ::= x \mid v \mid e_1 \ \text{bop} \ e_2 \mid e_1 \ ? \ e_2 \ : \ e_3 \mid \quad \text{expressions} \\
\text{const } x &= e_1; e_2 \mid e_1(e_2) \\
\text{bop } &\in \text{Bop} ::= + \mid * \mid === \mid !== \mid && \mid || \quad \text{binary operators} \\
p &= x \mid \epsilon \quad \text{function names} \\
t &= : \tau \mid \epsilon \quad \text{return type annotations}
\end{align*}
\]

A type \( \tau \) is either one of the base types \text{bool} or \text{number}, or a function type \( \tau_1 \Rightarrow \tau_2 \), where \( \tau_1 \) and \( \tau_2 \) are again types. A function type describes the signature of a function that has a parameter of type \( \tau_1 \) and returns a value of type \( \tau_2 \).

The new expression language is mostly identical to the untyped language that we considered in Chapter 5. The only difference is that the parameter of a function expression must now be annotated with a type. Moreover, function expressions can be annotated with optional return types. In fact, the return type annotation is mandatory for recursive functions. However, we do not enforce
this property at the level of the grammar rules that define the syntax of our
language. Instead, we will enforce it using the typing rules, which we discuss
below.

Parameter and return types are the only type annotations that the pro-
grammer is required to provide. In all other cases, our type checking algorithm
will be able to automatically infer the type of an expression from the usages of
operators and typed variables in the expression. In particular, variables that
are introduced using `const` declarations do not have to be typed explicitly with
type annotations.

### 6.1.3 Operational Semantics

We use a substitution-based small-step semantics for our typed language. The
corresponding evaluation relation $e \rightarrow e'$ is defined by the rules in Figure 6.1.
Note that the type annotations in function expressions are completely ignored
by the semantics. We will only use these annotations for the type checking
algorithm. The notable difference between our new semantics and the semantics
discussed in Chapter 5 is that we no longer have implicit type conversions.
Moreover, the new semantics has no rules that produce dynamic type errors.
If the new semantics encounters an expression that is not a value and where
we cannot take another evaluation step, such as $1/2$, then the evaluation will
simply get stuck at this point. The goal of our type-checking algorithm is to
detect programs that may get stuck during evaluation and reject them before
they are actually evaluated.

### 6.1.4 Typing Relation

We formalize our type checking algorithm in terms of a typing relation. The
typing relation is defined similarly to the big-step SOS with dynamic scoping
that we discussed in Chapter 5. The main difference is that now types take the
role of values. In particular, the typing relation is defined in terms of a typing
environment $\Gamma : \text{Var} \rightarrow \text{Typ}$ that maps the free variables in an expression to
their types. The typing relation is denoted by the judgment form

$$\Gamma \vdash e : \tau$$

which reads “under typing environment $\Gamma$, the expression $e$ has type $\tau$”. The
inference rules defining the typing relation are given in Figure 6.2.

If a type $\tau$ exists such that $\Gamma \vdash e : \tau$, we say that $e$ is well-typed under $\Gamma$. If $e$
is a closed expression (i.e., $e$ has no free-variables), then the typing environment
does not matter and we simply say that $e$ is well-typed. If the typing relation
fails to infer a type for a given $\Gamma$ and $e$, we say that $e$ has a type error under $\Gamma$.

It is instructive to compare the rules of the typing relation with the corre-
sponding rules for the big-step operational semantics with dynamic binding.
Note that for the typing relation, the order in which the types for subexpres-
sions are inferred is irrelevant. In fact, it is easy to see that the typing rules
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_1 \rightarrow e'_1 )</td>
<td>SearchBop ( e \rightarrow {&amp;&amp;,</td>
</tr>
<tr>
<td>( e_1 &amp; e_2 \rightarrow e'_1 &amp; e_2 )</td>
<td>SearchBop ( e \rightarrow e'_2 )</td>
</tr>
<tr>
<td>( e_1 \rightarrow e'_1 )</td>
<td>SearchIf ( \text{const } x = e_d; e_b \rightarrow \text{const } x = e'_d; e_b )</td>
</tr>
<tr>
<td>( e_1 ? e_2 : e_3 \rightarrow e'_1 ? e'_2 : e_3 )</td>
<td>SearchCall ( e \rightarrow e'_2 )</td>
</tr>
<tr>
<td>( n = n_1 &amp; n_2 ) ( \text{bop} \in {*, *} )</td>
<td>DoArith ( b = v_1 &amp; v_2 ) ( \text{bop} \in {===, !==} )</td>
</tr>
<tr>
<td>( n_1 &amp; n_2 \rightarrow n )</td>
<td>DoEqual ( v_1 &amp; v_2 \rightarrow b )</td>
</tr>
<tr>
<td>( \text{false} &amp; e_2 \rightarrow v_1 )</td>
<td>DoAndFalse ( \text{true} &amp; e_2 \rightarrow e_2 )</td>
</tr>
<tr>
<td>( \text{true}</td>
<td></td>
</tr>
<tr>
<td>( \text{true} ? e_2 : e_3 \rightarrow e_2 )</td>
<td>DoIfThen ( \text{false} ? e_2 : e_3 \rightarrow e_3 )</td>
</tr>
<tr>
<td>( \text{const } x = e_d; e_b \rightarrow e_b[v_d/x] )</td>
<td>DoConst ( v_1 = \text{function} (x: \tau) t e )</td>
</tr>
<tr>
<td>( v_1(v_2) \rightarrow e[v_2/x] )</td>
<td>DoCall ( v_1 = \text{function} x_1(x_2: \tau_2): \tau' e )</td>
</tr>
<tr>
<td>( v_1(v_2) \rightarrow e[v_1/x_1][v_2/x_2] )</td>
<td>DoCallRec ( v_1(v_2) \rightarrow e[v_1/x_1][v_2/x_2] )</td>
</tr>
</tbody>
</table>

Figure 6.1: Small-step operational semantics for typed JakartaScript
are deterministic, i.e., for any $\Gamma$ and $e$, there is at most one type $\tau$ such that $\Gamma \vdash e : \tau$. Thus, we can think of the typing relation $\Gamma \vdash e : \tau$ as a partial function that maps a typing environment $\Gamma$ and an expression $e$ to a type $\tau$. Also note that even though the typing relation uses an environment to keep track of the types of free variables occurring in an expression, the typing relation actually uses static binding rather than dynamic binding. Dynamic binding semantics occurred because the evaluation relation stored function expressions with free variables in the value environment. These function expressions then are later extracted to be evaluated in a different static scope. Hence, we obtained dynamic binding of the free variables in these function expressions. The typing relation, however, only stores types in the typing environment $\Gamma$, not expressions. Hence, the typing environment analyzes all variables of an expression in their static scopes.

### 6.1.5 Limitations

The static typing rules are much less permissive than the rules for the operational semantics with dynamic typing. That is, there are programs that our type checking algorithm considers ill-typed even though they can be safely evaluated according to our dynamically typed semantics of Chapter 5 (i.e., evaluation does not produce a dynamic type error). For example, the following two expressions are not well-typed due to the absence of implicit type conversion in the typing relation:

$$3 + \text{true}$$
$$0 \ ? \ 1 : 2$$

The omission of implicit type conversion is a deliberate design choice in our typed version of JavaScript. Implicit type conversion is often a source of subtle bugs in programs that can be difficult to debug. Statically typed languages therefore often require the programmer to make type conversion explicit by inserting type casts or explicit calls to conversion functions.

Some programs are not well-typed in our type system although they can be safely evaluated and do not rely on implicit type conversion. For example, the following conditional expression is not well-typed because the types of the two branches do not agree:

$$(0 == 1) \ ? \ 1 : \text{false}$$

This is again a deliberate design choice. This time, the motivation is to reduce the complexity of the type checking algorithm. If we wanted the above expression to be well-typed, we would have to introduce a supertype of $\text{number}$ and $\text{bool}$ that describes both $\text{Num}$ and $\text{Bool}$ values. We will consider such type systems later when we discuss subtyping in object-oriented languages.

Perhaps the most severe limitation of our type system is that all types are either base types (i.e., $\text{bool}$ and $\text{number}$) or they are function types that are constructed from base types. A type that is only constructed from base types is called monomorphic. This is in contrast to polymorphic types, which are types
\[
\begin{align*}
\Gamma \vdash b : \text{bool} & \quad \text{TypeBool} \quad \Gamma \vdash n : \text{number} & \quad \text{TypeNum} \quad x \in \text{dom}(\Gamma) & \quad \text{TypeVar} \\
\Gamma \vdash e_1 : \text{bool} & \quad \Gamma \vdash e_2 : \text{bool} & \quad \text{TypeAndOr} \quad bop \in \{\&\&, ||\} \\
\Gamma \vdash e_1 \ bop \ e_2 : \text{bool} & \\
\Gamma \vdash e_1 : \text{number} & \quad \Gamma \vdash e_2 : \text{number} & \quad \text{TypeArith} \quad bop \in \{K, J\} \\
\Gamma \vdash e_1 \ bop \ e_2 : \text{number} & \\
\Gamma \vdash e_1 : \tau & \quad \Gamma \vdash e_2 : \tau & \quad \tau \neq \tau_1 \Rightarrow \tau_2 & \quad \text{TypeEqual} \quad bop \in \{==, !=\} \\
\Gamma \vdash e_1 \ bop \ e_2 : \text{bool} & \\
\Gamma \vdash e_d : \tau_d & \quad \Gamma' = \Gamma[x \mapsto \tau_d] & \quad \Gamma' \vdash e_b : \tau_b & \quad \text{TypeConst} \\
\Gamma' \vdash \text{const } x = e_d; e_b : \tau_b & \\
\Gamma \vdash e_1 : \text{bool} & \quad \Gamma \vdash e_2 : \tau & \quad \Gamma \vdash e_3 : \tau & \quad \text{TypeIf} \\
\Gamma \vdash e_1 \ ? \ e_2 : e_3 : \tau & \\
\Gamma \vdash e_1 : \tau_1 \Rightarrow \tau & \quad \Gamma \vdash e_2 : \tau_2 & \quad \text{TypeCall} \\
\Gamma' \vdash e_1 (e_2) : \tau & \\
\Gamma' = \Gamma[x \mapsto \tau] & \quad \Gamma' \vdash e : \tau' & \quad \text{TypeFun} \\
\Gamma' \vdash \text{function}(x : \tau) \ ; e : \tau \Rightarrow \tau' & \\
\Gamma' = \Gamma[x \mapsto \tau] & \quad \Gamma' \vdash e : \tau' & \quad \text{TypeFunAnn} \\
\Gamma' = \Gamma[x_1 \mapsto \tau_1][x_2 \mapsto \tau_2] & \quad \Gamma' \vdash e : \tau' & \quad \tau_1 = \tau_2 \Rightarrow \tau' & \quad \text{TypeFunRec} \\
\Gamma' \vdash \text{function } x_1(x_2 : \tau_2) : \tau' \ ; e : \tau_1 & \\
\end{align*}
\]

Figure 6.2: Type inference rules
that use type variables to parameterize over other types. For example, generic types in Java and Scala are a form of polymorphic types.

Our simple type system does not support polymorphic types, which means that certain programs cannot be well-typed even though they can be safely evaluated. For example, consider the following program:

```plaintext
const id = function (x) x;
id(false) ? id(2) : id(1)
```

This program can be safely evaluated in our dynamically typed semantics. In particular, no type conversions are needed during evaluation. Nevertheless, there is no type \( \tau \) such that the following expression is well-typed in our new language:

```plaintext
const id = function (x: \tau) x;
id(false) ? id(2) : id(1)
```

If we choose \( \tau = \text{bool} \), then the calls \( \text{id}(2) \) and \( \text{id}(1) \) will yield static type errors. If we choose \( \tau = \text{number} \), then the call \( \text{id}(\text{false}) \) will not be well-typed. Finally, if we choose \( \tau = \tau_1 \Rightarrow \tau_2 \) for any types \( \tau_1 \) and \( \tau_2 \), then none of the calls to \( \text{id} \) will be well-typed.

In Section 6.3 we will consider a more expressive static type system that supports polymorphic types and in which the above program is well-typed. In fact, in this more expressive type system we no longer require that the programmer provides any form of type annotation. All types of expressions (including types of function parameters) can be automatically inferred.

### 6.2 Soundness of Static Type Checking

We have informally stated our intuition that the evaluation of a well-typed expression can never get stuck. We call a static type system with this property *sound* and a programming language with a sound static type system is called *statically type safe*. In the following, we give a formal definition of soundness and prove that our simple type system indeed satisfies this property.

With the definitions of the typing relation and the operational semantics in place, we can define the soundness of our type system as the conjunction of two properties: (1) each well-typed expression is either a value or can take an evaluation step, and (2) each evaluation step preserves well-typedness. Formally:

1. **Progress**: for all \( e \in \text{Expr} \), if \( e \) is closed and well-typed, then either \( e \in \text{Val} \) or \( e \rightarrow e' \) for some \( e' \in \text{Expr} \).

2. **Preservation**: for all \( e, e' \in \text{Expr} \) such that \( e \rightarrow e' \), if \( e \) is closed and well-typed then so is \( e' \).

Together, the progress and preservation properties ensure that the evaluation of a well-typed closed expression will either eventually terminate and produce a value, or go on forever. That is, soundness guarantees that the evaluation...
6 Types

will never get stuck in an expression that is not a value and that cannot take another evaluation step.

6.2.1 Soundness Proof (optional)

In the following, we prove that our type system indeed satisfies progress and preservation. We provide the key ideas of these proofs but leave out some of the details. You are encouraged to fill in these details as an exercise. Even if you do not complete the proofs, you should still try to understand the high-level arguments used in the proof outlines that we provide below. The math is actually quite simple as we only use induction, case splitting, and equational reasoning.

To prove progress, we first need an auxiliary lemma that characterizes values based on their types:

**Lemma 6.1 (Canonical Forms).** Let $v$ be a closed value. Then the following properties hold:

1. If $v$ is of type $\text{bool}$, then $v \in \text{Bool}$.
2. If $v$ is of type $\text{number}$, then $v \in \text{Num}$.
3. If $v$ is of type $\tau \Rightarrow \tau'$, then either
   - $\hat{v} = \text{function}(x:\tau) t e$ where $t = \epsilon$ or $t = :\tau'$, or
   - $\hat{v} = \text{function} x_1(x_2:\tau) :\tau'e$

**Proof (Exercise).** The lemma follows easily from the rules of the typing relation that involve values.

**Theorem 6.2 (Progress).** Let $e \in \text{Expr}$ be a closed expression. If $e$ is well-typed, then either $e \in \text{Val}$ or $e \rightarrow e'$ for some $e' \in \text{Expr}$.

**Proof (Exercise).** Since $e$ is well-typed and closed we have $\emptyset \vdash e : \tau$ for some type $\tau$. The proof goes by induction on the derivation of $\emptyset \vdash e : \tau$. We proceed by cases on the final typing rule used in the derivation.

If the last rule that was used in the derivation of $\emptyset \vdash e : \tau$ is $\text{TypeNum}$, $\text{TypeBool}$, $\text{TypeFunction}$, $\text{TypeFunctionAnn}$, or $\text{TypeFunctionRec}$, then $e$ must be a value and there is nothing to be proved. The rule $\text{TypeVar}$ cannot be the last rule used in the derivation since it would imply that $e = x$ for some variable $x \in \text{Var}$, contradicting the assumption that $e$ is closed. From the remaining cases, we only show the case for the rule $\text{TypeCall}$. The other cases are similar and left as an exercise.

Thus, assume that $\text{TypeCall}$ is the last rule that was used in the derivation of $\emptyset \vdash e : \tau$. From the premises of $\text{TypeCall}$, it follows that $e$ must be of the form $e = e_1(e_2)$ such that $\emptyset \vdash e_1 : (\tau' \Rightarrow \tau)$ and $\emptyset \vdash e_2 : \tau'$ for some $e_1$, $e_2$, $x$, and $\tau'$. We distinguish three subcases:
Case 1 $e_1$ is not a value: since $e_1$ is closed and well-typed, $e_1$ can take a step by induction hypothesis. That is, there exists $e_1'$ such that $e_1 \rightarrow e_1'$. Then by rule \textsc{SearchCall}, we can conclude $e \rightarrow e_1'(e_2)$.

Case 2 $e_1$ is a value but $e_2$ is not: similar to the previous case, we can conclude by induction hypothesis and rule \textsc{SearchCall} that $e \rightarrow e_1(e_2)$ for some $e_2'$.

Case 3 $e_1$ and $e_2$ are both values: it follows from the Canonical Forms Lemma that $e_1$ must be of the forms $e_1 = \text{function}(x:\tau') t e'$ or $e_1 = \text{function} x_1(x_2:\tau') : \tau e'$.

In the first case, we conclude from rule \textsc{DoCall} that $e \rightarrow e'[e_2/x]$. In the second case, rule \textsc{DoCallRec} implies that $e \rightarrow e'[e_1/x_1][e_2/x_2]$.

To prove the preservation property, we first need to state some technical lemmas. We start with two lemmas that allow us to transform typing derivations in specific cases.

First, the Permutation Lemma states that the order in which we extend the typing environment does not matter for a typing derivation as long as the variables for which we extend the environment are distinct.

**Lemma 6.3 (Permutation).** If $\Gamma[x \rightsquigarrow \tau_1][y \rightsquigarrow \tau_2] \vdash e : \tau$ and $x \neq y$, then $\Gamma[y \rightsquigarrow \tau_2][x \rightsquigarrow \tau_1] \vdash e : \tau$.

**Proof (Exercise).** The intuition for why this lemma is correct is that if we extend an environment $\Gamma' = \Gamma[x \rightsquigarrow \tau_1]$ with another binding for a variable $y$, $\Gamma'' = \Gamma'[y \rightsquigarrow \tau_2]$, then the second extension does not interfere with the first extension unless $x$ and $y$ are equal.

Thus, define $\Gamma_1 = \Gamma[x \rightsquigarrow \tau_1][y \rightsquigarrow \tau_2]$ and $\Gamma_2 = \Gamma[y \rightsquigarrow \tau_2][x \rightsquigarrow \tau_1]$. All you need to prove is that $\Gamma_1 = \Gamma_2$. To do so, prove that for all variables $z$, if $z \in \text{dom}(\Gamma_1)$ then $z \in \text{dom}(\Gamma_2)$ and $\Gamma_1(z) = \Gamma_2(z)$, and vice versa. The actual proof is easy. You only need to expand the definition of the extension function $[\cdot \rightsquigarrow \cdot]$ and then case split on $z$ (i.e., whether $z = x$, $z = y$, or $z \neq x$ and $z \neq y$).

The second technical lemma that we will need is the Weakening Lemma. It states that if an expression $e$ is well-typed under some environment $\Gamma$, then it is also well-typed under any environment $\Gamma' = \Gamma[x \rightsquigarrow \tau']$, provided that $x$ does not occur free in $e$. The intuition for this lemma is that if $x$ occurs at all in $e$, then any such occurrence must be a bound occurrence. However, the mapping of $x$ to $\tau'$ in the environment will be overwritten at each defining occurrence of $x$ in $e$. These updated environments will then be used to type the using occurrences of $x$ in the scope of that defining occurrence. Thus, the mapping of $x$ to $\tau'$ will never actually be used in the typing derivation for $e$. 


Lemma 6.4 (Weakening). If $\Gamma \vdash e : \tau$ and $x \notin fv(e)$, then $\Gamma[x \mapsto \tau'] \vdash e : \tau$

Proof (Exercise). The proof goes by induction on the derivation of $\Gamma \vdash e : \tau$ and is quite simple. \qed

The core of the proof of the preservation property is the following Substitution Lemma, which states that well-typedness is preserved under substitution. The proof of this Lemma uses the permutation and weakening lemmas.

Lemma 6.5 (Preservation of Types under Substitutions). If $\Gamma[x \mapsto \tau_2] \vdash e_1 : \tau_1$ and $\Gamma \vdash e_2 : \tau_2$, then $\Gamma \vdash e_1[e_2/x] : \tau_1$.

Proof (Exercise). The proof goes by induction on the derivation of $\Gamma[x \mapsto \tau'] \vdash e_1 : \tau$. We proceed by cases on the final typing rule used in this derivation. The most interesting cases are the ones for variables, procedural abstraction, and const declarations. We consider the rules TypeVar and TypeFunction explicitly and leave the remaining rules as an exercise.

If the last rule in the derivation is TypeVar, then $e_1 = y$ and $\Gamma[x \mapsto \tau_2](y) = \tau_1$. There are two subcases based on whether $y$ is $x$ or another variable. If $y = x$, then $e_1[e_2/x] = y[e_2/x] = e_2$ and $\tau_1 = \tau_2$. Then $\Gamma \vdash e_2 : \tau_2$. If $y \neq x$, then $e_1[e_2/x] = y$ and $\Gamma(y) = \tau_1$. Thus, $\Gamma \vdash e_1[e_2/x] : \tau_1$ follows immediately from rule TypeVar.

If the last rule in the derivation is TypeFunction, then we know that

- $e_1 = \text{function}(y: \tau) \ e$,
- $\tau_1 = \tau \Rightarrow \tau'$, and
- $\Gamma[x \mapsto \tau_2][y \mapsto \tau] \vdash e : \tau'$.

Since we are allowed to consistently rename bound variables by fresh variables, we may assume that $y \neq x$ and $y \notin fv(e_2)$. Using $y \neq x$ and the Permutation Lemma, we can derive $\Gamma[y \mapsto \tau][x \mapsto \tau_2] \vdash e : \tau'$ from $\Gamma[x \mapsto \tau_2][y \mapsto \tau] \vdash e : \tau'$. Using $y \notin fv(e_2)$ and the Weakening Lemma, we can conclude $\Gamma[y \mapsto \tau] \vdash e_2 : \tau_2$ from $\Gamma \vdash e_2 : \tau_2$. Now, from the induction hypothesis it follows $\Gamma[y \mapsto \tau] \vdash e_2[e_2/x] : \tau'$. Thus, by rule TypeFunction we conclude

$\Gamma \vdash \text{function}(y: \tau) \ (e[e_2/x]) : \tau \Rightarrow \tau'$.

By the definition of substitution and since $y \neq x$, we have

$\text{function}(y: \tau) \ (e[e_2/x]) = (\text{function}(y: \tau) \ e)[e_2/x] = e_1[e_2/x]$

Hence, we conclude $\Gamma \vdash e_1[e_2/x] : \tau_1$. \qed

Theorem 6.6 (Preservation). Let $e, e' \in Expr$ such that $e \rightarrow e'$. If $e$ is closed and well-typed, then so is $e'$.
Proof (Exercise). Since $e$ is closed and well-typed, we must have $\emptyset \vdash e : \tau$ for some type $\tau$. We prove a slightly stronger property than required, namely that if $e \rightarrow e'$, then $e'$ is closed and $\emptyset \vdash e' : \tau$ (i.e., the exact type $\tau$ of $e$ is preserved under evaluation). The proof goes by induction on the derivation of $\emptyset \vdash e : \tau$ using case splitting on the last rule of the derivation. At each step of the induction, we assume that the desired property holds for all subderivations (i.e., whenever $\emptyset \vdash e_1 : \tau_1$ is proved by a subderivation and $e_1 \rightarrow e_1'$, then $e_1'$ is closed and $\emptyset \vdash e_1' : \tau_1$). We leave the details of the proof as an exercise. Hint: In each case, the final typing rule in the derivation of $\emptyset \vdash e : \tau$ determines the top-level syntactic structure of $e$. In turn, this restricts the possible final rules that may have been used in the derivation of $e \rightarrow e'$. By further case splitting on these relevant final rules of the small-step SOS, you can apply the induction hypothesis where needed. The interesting cases are the rules TYPECONST and the typing rules for calls in combination with the relevant do rules of the small-step SOS for these cases. In all these interesting cases, you need to apply the substitution lemma (Lemma 6.5) to complete the proof.

6.3 Parametric Polymorphism (optional)

In the design of our simple type system for JAKARTASCRIPTION we have made several decisions that simplify the type checking and type inference problem. Unfortunately, these decisions also affect the usability and expressiveness of our type system. In particular, we required that the programmer annotates each function with the types of its parameters, and each recursive function with the type of its return value. Annotating programs with types is often tedious. As programmers we prefer to be released from the burden of having to write such annotations explicitly. Moreover, we have observed that the type system rejects certain programs that can be safely executed according to our operational semantics, e.g., programs which make use of polymorphic functions.

In this section, we will study an interesting point in the design space of static type systems that is referred to as the Hindley-Milner type system. This type system solves both of the limitations of our simple type system: (1) it enables type inference without requiring any programmer-provided type annotations and (2) it can deal with programs that make use of polymorphic functions. The Hindley-Milner type system is implemented in a number of programming languages, specifically the ML family of languages, which includes SML and OCaml, as well as related languages such as Haskell.

6.3.1 Type Inference without Type Annotations

Before we introduce the Hindley-Milner type system formally, we explain it through a series of examples.

As a first example, consider the following implementation of the factorial function in JAKARTASCRIPTION:

```
function fac(x) (x == 0 ? 1 : x * fac(x - 1))
```
For this function to be well-typed in our simple type system, we would have to explicitly annotate the parameter \( x \) of function \( \text{fac} \) as well as the function’s return type. However, if we take a closer look at the function’s body:

\[
x === 0 \ ? 1 : x \ast \text{fac}(x - 1)
\]

we observe that there is really only one possible type annotation that can work. Specifically, if we look at the test \( x === 0 \) in the conditional, then we can infer that \( x \) must be of type \text{number} since it is compared with the value 0. Similarly, from the fact that the result of the recursive call \( \text{fac}(x - 1) \) is used in a multiplication operation, we can infer that the return value of \( \text{fac} \) must also be of type \text{number}. The Hindley-Milner type system exploits this idea to infer all types without explicit type annotations.

Types with Type Variables

In certain cases, the operations from which an expression is built may not be specific enough to infer a monomorphic type for the expression (i.e., a type that is either one of the base types \text{number} and \text{bool}, or a function type built from base types). For example, consider the following curried function

\[
\text{apply} = \text{function}(x) (\text{function}(f) (f(x)))
\]

From the subexpression \( f(x) \) of \( \text{apply} \) we can infer that the type of \( f \) must be a function type \( \tau_1 \Rightarrow \tau_2 \) (since \( f \) is called). Moreover the type of \( x \) must be the same as the parameter type \( \tau_1 \) of that function type (since \( x \) is the argument of the call to \( f \)). However, we don’t have enough information to determine the specific type \( \tau_1 \) and what the specific return type \( \tau_2 \) of \( f \) is. In fact, for the evaluation of a call to the function \( \text{apply} \) the specific types of the parameter and return value of \( f \) don’t matter. We say that \( \text{apply} \) is parameterized in the types \( \tau_1 \) and \( \tau_2 \).

To deal with type parameterization, we extend our language of types with type variables that serve as placeholders for other types. We use Greek letters to denote such type variables. The new type language is as follows:

\[
\begin{align*}
\alpha & \in TVar \\
\tau & \in Typ ::= \text{bool} | \text{number} | \alpha | \tau_1 \Rightarrow \tau_2
\end{align*}
\]

If a type \( \tau \) does not contain any type variables, it is called \textit{monomorphic}, otherwise it is called \textit{polymorphic}. We can think of polymorphic types as types that parameterize over monomorphic types. For example, the type \( \alpha \Rightarrow \text{bool} \) stands for the monomorphic types \( \text{bool} \Rightarrow \text{bool}, \text{number} \Rightarrow \text{bool}, (\text{bool} \Rightarrow \text{bool}) \Rightarrow \text{bool}, \) etc.

A Complete Example

To infer the type of a given expression \( e \) we now proceed in three steps:
1. Associate a fresh type variable with each subexpression occurring in $e$.

2. Generate a set of equality constraints over types from the syntactic structure of $e$. These typing constraints relate the introduced type variables with each other and impose restrictions on the types that they stand for.

3. Solve the generated typing constraints. If a solution of the constraints exists, the expression is well-typed and we can read off the types of all subexpressions (including $e$ itself) from the computed solution. Otherwise, if no solution exists, $e$ has a type error.

We explain these three steps using our initial example:

$$x \equiv 0 \ ? \ 1 : x \ast \text{fac}(x - 1)$$

Let us call this expression $e$. We generate the type variables and typing constraints for the subexpressions of $e$ in one go:

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Type Variable</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\alpha_x$</td>
<td>$\alpha_0 \equiv \text{number}$</td>
</tr>
<tr>
<td>$0$</td>
<td>$\alpha_0$</td>
<td>$\alpha_0 \equiv \text{number}$</td>
</tr>
<tr>
<td>$x \equiv 0$</td>
<td>$\alpha_{eq}$</td>
<td>$\alpha_x \equiv \alpha_0$, $\alpha_{eq} \equiv \text{bool}$, $\alpha_x \equiv \text{number}$</td>
</tr>
<tr>
<td>$1$</td>
<td>$\alpha_1$</td>
<td>$\alpha_1 \equiv \text{number}$</td>
</tr>
<tr>
<td>$\text{fac}$</td>
<td>$\alpha_{fac}$</td>
<td>$\alpha_{call} \overset{\text{call}}{\Rightarrow} (\alpha_x \Rightarrow \alpha_{call})$</td>
</tr>
<tr>
<td>$x - 1$</td>
<td>$\alpha_\ast$</td>
<td>$\alpha_x \equiv \text{number}$, $\alpha_1 \equiv \text{number}$, $\alpha_\ast \equiv \text{number}$</td>
</tr>
<tr>
<td>$\text{fac}(x - 1)$</td>
<td>$\alpha_{call}$</td>
<td>$\alpha_{fac} \overset{\text{call}}{\Rightarrow} (\alpha_x \Rightarrow \alpha_{call})$</td>
</tr>
<tr>
<td>$x \ast \text{fac}(x - 1)$</td>
<td>$\alpha_{\ast e}$</td>
<td>$\alpha_{eq} \equiv \text{bool}$, $\alpha_{\ast e} \equiv \alpha_1$, $\alpha_{\ast e} \equiv \alpha_\ast$</td>
</tr>
<tr>
<td>$x \equiv 0 \ ? \ 1 : x \ast \text{fac}(x - 1)$</td>
<td>$\alpha_{\ast e}$</td>
<td>$\alpha_{eq} \equiv \text{bool}$, $\alpha_{\ast e} \equiv \alpha_1$, $\alpha_{\ast e} \equiv \alpha_\ast$</td>
</tr>
</tbody>
</table>

Each row lists one of $e$’s subexpressions, the type variable that stands for the type of that subexpression, and a list of equality constraints that constrain the type variable with the type variables of other subexpressions. To avoid notational confusion with actual equality on types in our mathematical meta language, we use the symbol $\equiv$ to equate types in the typing constraints. Note that the subexpressions $x$ and $1$, which occur multiple times in $e$, are only listed once.

We explain two of the rows in the above table in more detail. The subexpression $x - 1$ has the associated type variable $\alpha_\ast$. Since we know that the arguments of the binary operator $-$ must be of type $\text{number}$, we obtain two typing constraints $\alpha_x = \text{number}$ and $\alpha_1 = \text{number}$, where $\alpha_x$ and $\alpha_1$ are the type variables associated with $x$ and $1$, respectively. Similarly, we know that the result of operator $-$ is again a value of type $\text{number}$. Hence, we obtain the additional constraint $\alpha_\ast = \text{number}$. Another particularly interesting case is the call expression $\text{fac}(x - 1)$. From this expression we can infer that $\text{fac}$ must be a function whose parameter type matches the type of the subexpression $x - 1$ (which is the argument of the call) and whose result type matches the type
of the entire call expression (which is denoted by $\alpha_{call}$). This information is captured by the typing constraint

$$\alpha_{fac} \doteq (\alpha_\rightarrow \alpha_{call}) .$$

If we collect the typing constraints that have been generated for all of $e$’s subexpressions from the table above, we obtain the following set of constraints:

1. $\alpha_0 \doteq \text{number}$
2. $\alpha_x \doteq \alpha_0$
3. $\alpha_{eq} \doteq \text{bool}$
4. $\alpha_1 \doteq \text{number}$
5. $\alpha_\rightarrow \doteq \text{number}$
6. $\alpha_{fac} \doteq (\alpha_\rightarrow \alpha_{call})$
7. $\alpha_{call} \doteq \text{number}$
8. $\alpha_s \doteq \text{number}$
9. $\alpha_{eq} \doteq \text{bool}$
10. $\alpha_{ite} \doteq \alpha_1$
11. $\alpha_{ite} \doteq \alpha_\rightarrow$

We denote this set of constraints by $C$. To see whether $e$ is well-typed, we have to solve $C$. That is, we have to find a mapping $\sigma : TVar \rightarrow Typ$ from type variables to types, such that if we substitute the type variables occurring in each constraint according to $\sigma$, then the two sides of each constraint become equal. That is, $\sigma$ must satisfy that for all constraints $\tau_1 \doteq \tau_2 \in C$, $\tau_1 \sigma = \tau_2 \sigma$. Here, $\tau \sigma$ denotes substitution of the type variables occurring in a type $\tau$ according to $\sigma$. We refer to the problem of finding such a solution $\sigma$ for a given set of typing constraints $C$ as the unification problem. A solution $\sigma$ of a unification problem instance $C$ is called a unifier of $C$.

We can compute a unifier $\sigma$ from the set of constraints $C$ using a simple iterative algorithm. The algorithm starts with a trivial candidate solution $\sigma_0 = \emptyset$ and then processes the equality constraints one at a time, extending $\sigma_0$ to an actual unifier of $C$. We describe this algorithm below (see Figure 6.4). In the following, we show how it works for our concrete example.

As noted, we start with the trivial candidate solution $\sigma_0 = \emptyset$ and process the constraints one at a time, extending $\sigma_0$ as we go along. The order in which the constraints are processed does not matter. We choose to process them in the order given above. That is, the first constraint that we consider is

1. $\alpha_0 \doteq \text{number}$

Observe that the left-hand side of this constraint is a type variable $\alpha_0$. Thus, to solve this specific constraint, all we need to do is map $\alpha_0$ to the type on the right-hand side, which is $\text{number}$. We therefore define $\sigma_1 = \sigma_0[\alpha_0 \mapsto \text{number}]$. Observe, that $\sigma_1$ is indeed a unifier of constraint 1, since

$$\alpha_0 \sigma_1 = \text{number} = \text{number} \sigma_1$$

More generally, the unification algorithm will maintain the invariant that after processing the $i$th constraint, the current candidate solution $\sigma_i$ will unify all previously processed constraints 1 to $i$. 
We maintain another invariant in our algorithm, namely that any type variable that is assigned by our current candidate solution no longer occurs in any of the constraints that still need to be processed. To ensure this invariant, we have to apply the candidate unifier $\sigma_i$ to the unprocessed constraints after each extension. That is, in our example, we substitute $\alpha_0$ by \texttt{number} in constraints 2 to 11, which gives us the new set of constraints:

$$
\begin{align*}
2. \quad & \alpha_x \equiv \texttt{number} \\
3. \quad & \alpha_{eq} \equiv \texttt{bool} \\
4. \quad & \alpha_1 \equiv \texttt{number} \\
5. \quad & \alpha_\cdot \equiv \texttt{number} \\
6. \quad & \alpha_{\texttt{fac}} \equiv (\alpha_\cdot \Rightarrow \alpha_{\texttt{call}}) \\
7. \quad & \alpha_{\texttt{call}} \equiv \texttt{number} \\
8. \quad & \alpha_\cdot \equiv \texttt{number} \\
9. \quad & \alpha_{eq} \equiv \texttt{bool} \\
10. \quad & \alpha_{ite} \equiv \alpha_1 \\
11. \quad & \alpha_{ite} \equiv \alpha_\cdot 
\end{align*}
$$

We continue processing the constraints in the given order. The constraints 2 to 6 are similar to constraint 1. We extend the candidate unifier $\sigma_0$ for each of these cases as described above to obtain the following remaining constraints and current candidate unifier $\sigma_6$:

$$
\begin{align*}
7. \quad & \alpha_{\texttt{call}} \equiv \texttt{number} \\
8. \quad & \alpha_\cdot \equiv \texttt{number} \\
9. \quad & \texttt{bool} \equiv \texttt{bool} \\
10. \quad & \alpha_{ite} \equiv \texttt{number} \\
11. \quad & \alpha_{ite} \equiv \texttt{number} \\
\end{align*}
$$

The case for constraint 7 is similar to the previous cases, except that the type variable $\alpha_{\text{call}}$ now also appears in the type to which $\alpha_{\text{fac}}$ is mapped by the current candidate unifier $\sigma_6$. In order to maintain our invariant that the candidate unifier unifies all constraints processed so far, we also have to apply the mapping $\alpha_{\text{call}} \mapsto \texttt{number}$ to $\sigma_6$ before we extend $\sigma_6$ with this new mapping. After we do this, we obtain the new mapping:

$$
\begin{align*}
\sigma_7 = \{ \alpha_0 \mapsto \texttt{number}, \quad & \sigma_6 = \{ \alpha_0 \mapsto \texttt{number}, \\
\alpha_x \mapsto \texttt{number}, \quad & \alpha_x \mapsto \texttt{number}, \\
\alpha_{eq} \mapsto \texttt{bool}, \quad & \alpha_{eq} \mapsto \texttt{bool}, \\
\alpha_1 \mapsto \texttt{number}, \quad & \alpha_1 \mapsto \texttt{number}, \\
\alpha_\cdot \mapsto \texttt{number}, \quad & \alpha_\cdot \mapsto \texttt{number}, \\
\alpha_{ite} \mapsto \texttt{number}, \quad & \alpha_{ite} \mapsto \texttt{number}, \\
\alpha_{\texttt{fac}} \mapsto (\texttt{number} \Rightarrow \alpha_{\text{call}}) \} \\
\alpha_{\text{call}} \mapsto \texttt{number} \}
\end{align*}
$$
The unprocessed constraints 8 to 11 remain unchanged since \( \alpha_{\text{call}} \) does not appear in any of these. The case for constraint 8 is again similar to the first cases, so we process it to obtain the new candidate unifier \( \sigma_8 = \sigma_7[\alpha_s \mapsto \text{number}] \).

Constraint 9 is again interesting. It is now of the form \( \text{bool} \equiv \text{bool} \). The two sides of this constraint are already unified, so there is no need to extend \( \sigma_8 \). We thus simply define \( \sigma_9 = \sigma_8 \) and proceed with constraint 10. Processing constraint 10 again extends the candidate unifier, after which constraint 11 becomes trivially unified. After processing all constraints we obtain the following mapping

\[
\sigma_{11} = \{ \alpha_0 \mapsto \text{number}, \alpha_x \mapsto \text{number}, \alpha_{eq} \mapsto \text{bool}, \alpha_1 \mapsto \text{number}, \\
\alpha_x \mapsto \text{number}, \alpha_{fac} \mapsto (\text{number} \Rightarrow \text{number}), \\
\alpha_{call} \mapsto \text{number}, \alpha_{s} \mapsto \text{number}, \alpha_{ite} \mapsto \text{number} \}
\]

Observe that this mapping is indeed a unifier for the original set of constraints. Thus, we have shown that the expression \( e \) is well-typed.

**Inferring Polymorphic Types**

Next, we apply the type inference algorithm to our polymorphic example:

\[
\text{apply} = \text{function}(x) (\text{function}(f) (f(x)))
\]

From the expression \( \text{apply} \) we collect the following subexpressions with associated type variables and typing constraints:

\[
\begin{array}{llll}
  f & \alpha_f & \rightarrow & \\
  x & \alpha_x & \rightarrow & \\
  f(x) & \alpha_{call} & \alpha_f \equiv (\alpha_x \Rightarrow \alpha_{call})
\end{array}
\]

\[
\begin{array}{llll}
  \text{function}(f) (f(x)) & \alpha_{fun} & \alpha_{fun} = (\alpha_f \Rightarrow \alpha_{call})
\end{array}
\]

\[
\begin{array}{llll}
  \text{function}(x) (\text{function}(f) (f(x))) & \alpha_{apply} & \alpha_{apply} = (\alpha_x \Rightarrow \alpha_{fun})
\end{array}
\]

That is, we need to solve the following unification problem to show that \( \text{apply} \) is well-typed:

1. \( \alpha_f \equiv (\alpha_x \Rightarrow \alpha_{call}) \)
2. \( \alpha_{fun} \equiv (\alpha_f \Rightarrow \alpha_{call}) \)
3. \( \alpha_{apply} \equiv (\alpha_x \Rightarrow \alpha_{fun}) \)

Starting with the trivial candidate unifier \( \sigma_0 = \emptyset \) we process the first two constraints as described before to obtain the following updated candidate unifier:

\[
\sigma_2 = \{ \alpha_f \mapsto \alpha_x \Rightarrow \alpha_{call} \\
\alpha_{fun} \mapsto \alpha_x \Rightarrow \alpha_{call} \Rightarrow \alpha_{call} \}
\]

The remaining constraint 3 now looks as follows:
3. $\alpha_{\text{apply}} = (\alpha_x \Rightarrow (\alpha_x \Rightarrow \alpha_{\text{call}}) \Rightarrow \alpha_{\text{call}})$

Processing this remaining constraint yields the actual unifier of the original set of constraints:

$$\sigma_3 = \{ \alpha_f \mapsto \alpha_x \Rightarrow \alpha_{\text{call}}, \alpha_{\text{fun}} \mapsto (\alpha_x \Rightarrow \alpha_{\text{call}}) \Rightarrow \alpha_{\text{call}}, \alpha_{\text{apply}} \mapsto (\alpha_x \Rightarrow (\alpha_x \Rightarrow \alpha_{\text{call}}) \Rightarrow \alpha_{\text{call}}) \}$$

We obtain the inferred type of $\text{apply}$ by looking up the type to which the associated type variable $\alpha_{\text{apply}}$ is mapped by the unifier $\sigma_3$. Note that this type is polymorphic as it still contains the type variables $\alpha_x$ and $\alpha_{\text{call}}$. This tells us that we can safely call $\text{apply}$ with any arguments $x$ and $f$, as long as $f$ is a function whose parameter type matches the type of $x$. In particular, both of the following specific usages of $\text{apply}$ are safe:

$$\text{apply}(3)(\text{function}(x)(x + 2))$$
$$\text{apply}(\text{true})(\text{function}(x)(x || \text{false}))$$

Detecting Type Errors

So far we have only considered cases in which the type inferences succeeded. The question remains what happens if an expression is not well-typed and how to detect this during unification. To this end, we consider another example:

$$x ? x + 1 : 3$$

From this expression, we generate the following typing constraints:

1. $\alpha_x \equiv \text{number}$
2. $\alpha_x \equiv \text{number}$
3. $\alpha_1 \equiv \text{number}$
4. $\alpha_3 \equiv \text{number}$
5. $\alpha_x \equiv \text{bool}$
6. $\alpha_{\text{ite}} \equiv \alpha_x$
7. $\alpha_{\text{ite}} \equiv \alpha_3$

After processing the first 4 constraints we obtain the following candidate unifier

$$\sigma_4 = \{ \alpha_x \mapsto \text{number}, \alpha_x \mapsto \text{number}, \alpha_1 \mapsto \text{number}, \alpha_3 \mapsto \text{number} \}$$

and the remaining set of constraints now looks as follows:

5. $\text{number} \equiv \text{bool}$
Continuing with constraint 5, we detect a problem. The constraint is now of the form \texttt{number} = \texttt{bool}. Since \texttt{number} and \texttt{bool} are two distinct monomorphic types, there exists no unifier that can make these distinct types equal. This means that the generated unification problem has no solution. We therefore abort and report a type error in the original expression. If we look at the original expression from which we generated the typing constraints, we observe that the problem comes from the two usages of \( x \). First, we use \( x \) in the test of the conditional expression, which means that \( x \) must have type \texttt{bool}. Then we use \( x \) again in the “then” branch as an argument to \( \texttt{K} \), which means that \( x \) must also have type \texttt{number} – a contradiction.

**Self application and Occurrence Check**

There is one specific kind of type error that has to do with the restrictions on the degree of polymorphism that we allow in our type language. To explain this issue, consider the following expression where we call a variable \( x \) on itself:

\[
x(x)
\]

We refer to this kind of expression as self application. From this expression, we generate a single typing constraint:

\[
\alpha_x = \alpha_x \Rightarrow \alpha_{\text{call}}
\]

On first sight, it appears that this constraint has a simple solution given by the following mapping:

\[
\sigma = \{ \alpha_x \mapsto \alpha_x \Rightarrow \alpha_{\text{call}} \}
\]

However, observe that \( \sigma \) is not actually a unifier of the constraint since applying \( \sigma \) to the two sides of the constraint does not make the two sides equal:

\[
\alpha_x \sigma = \alpha_x \Rightarrow \alpha_{\text{call}} \\
\neq (\alpha_x \Rightarrow \alpha_{\text{call}}) \Rightarrow \alpha_{\text{call}} \\
= (\alpha_x \Rightarrow \alpha_{\text{call}})\sigma
\]

In fact, there is no mapping of the type variables \( \alpha_x \) and \( \alpha_{\text{call}} \) to any type in our language such that the two sides of the constraint would be equal. The expression \( x(x) \) is therefore not well-typed. The reason for this restriction is that type variables in polymorphic types stand for monomorphic types only rather than arbitrary types. That is, in our type language we do not consider polymorphic types that are parameterized by other polymorphic types. Such more general polymorphic types are referred to as higher-ranked polymorphic types. While we could make our type system more general and allow higher-ranked polymorphic types, we would no longer be able to solve the type inference problem and the programmer would again have to provide type annotations.
We can detect situations such as self application by introducing an additional check in our unification algorithm. When we process a constraint that is of the form \( \alpha \rightarrow \tau \), we first check whether \( \alpha \) occurs in \( \tau \) before we extend the current candidate unifier with the mapping \( \alpha \mapsto \tau \). If \( \alpha \) occurs in \( \tau \), we abort the algorithm and report a type error. We refer to this additional check as the occurrence check.

### 6.3.2 The Hindley-Milner Type System

We formalize the Hindley-Milner type system using a simplified version of the language from Chapter 5. Specifically, we drop equality operators and \texttt{const} declarations from the syntax. We will discuss later how these constructs can be added back to the language. The simplified grammar of our new language is as follows:

\[
\begin{align*}
n & \in \text{Num} \quad \text{numbers (double)} \\
x & \in \text{Var} \quad \text{variables} \\
b & \in \text{Bool} ::= \text{true} \mid \text{false} \quad \text{Booleans} \\
v & \in \text{Val} ::= n \mid b \mid \text{function } p(x) \ e \\
e & \in \text{Expr} ::= x \mid v \mid e_1 \ bop \ e_2 \mid e_1 \ ? \ e_2 \ : \ e_3 \mid e_1(e_2) \quad \text{expressions} \\
bop & \in \text{Bop} ::= + \mid * \mid &\& \mid |\mid | \quad \text{binary operators} \\
p & ::= x \mid \epsilon \quad \text{function names}
\end{align*}
\]

Compared to the language from Section 6.1, the new language does not support type annotations for function abstractions. We could allow such annotations as a form of optional code documentation. However, they are not needed for the type inference. Hence, we omit them for the sake of simplicity.

#### Typing Constraint Generation

We describe the actual constraint generation for the type inference using a modified version of the typing relation of our simple type system. The new typing relation is denoted by judgments of the form:

\[
\Gamma \vdash e : \alpha \mid C
\]

The typing environment \( \Gamma \) and the expression \( e \) are the input of the relation. The output computed by the relation is the type variable \( \alpha \) and the set of typing constraints \( C \). Informally, this judgment states that under typing environment \( \Gamma \), the expression \( e \) is well-typed, provided that the typing constraints \( C \) have a solution \( \sigma \). In particular, given a solution \( \sigma \) of \( C \), the type of \( e \) for this solution is \( \sigma(\alpha) \).

The typing environment \( \Gamma \) is simply a mapping from (expression) variables to type variables, i.e., \( \Gamma : \text{Var} \rightarrow \text{TVar} \). We need the typing environment to make sure that we associate the same type variable \( \Gamma(x) \) with each free occurrence of a variable \( x \) in \( e \).
The inference rules that define the typing constraint generation relation are given in Figure 6.3. It is instructive to compare these rules with the corresponding rules for the typing relation of our simple type system discussed in Section 6.1. Also note that the constraints that we generated for the examples in the previous section exactly correspond to those obtained by the rules in Figure 6.3 (modulo renaming of type variables).

### Unification Algorithm

Figure 6.4 formalizes the unification algorithm that we used to solve the generated typing constraints in the examples. The algorithm is split into two functions: `unify` and `unifyOne`. The function `unify` takes a set of typing constraints \( C \) and a candidate unifier \( \sigma \) and returns an extension of \( \sigma \) to an actual unifier of \( C \), or \( \perp \) if no such unifier exists. The actual unification is done by the function `unifyOne`, which extends the given candidate unifier to a unifier for a single constraint \( \tau \). The function `unify` applies the function `unifyOne` to the individual constraints in \( C \), one at a time. The defining cases for `unify` and

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\alpha \text{ fresh}}{\Gamma \vdash b : \alpha \mid { \alpha = \text{bool} }} )</td>
<td>HMBOOL</td>
</tr>
<tr>
<td>( \frac{\alpha \text{ fresh}}{\Gamma \vdash n : \alpha \mid { \alpha = \text{number} }} )</td>
<td>HNUM</td>
</tr>
<tr>
<td>( \frac{}{\Gamma \vdash x : \Gamma(x) \mid \emptyset} )</td>
<td>HMVAR</td>
</tr>
<tr>
<td>( \frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \quad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \quad \text{bop} \in { &amp; &amp;,</td>
<td></td>
</tr>
<tr>
<td>( \frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \quad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \quad \text{bop} \in { +, * } \quad \alpha \text{ fresh}}{\Gamma \vdash e_1 \text{ bop} e_2 : \alpha \mid C_1 \cup C_2 \cup { \alpha = \text{number}, \alpha_1 = \text{number}, \alpha_2 = \text{number} }} )</td>
<td>HMARITH</td>
</tr>
<tr>
<td>( \frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \quad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \quad \Gamma \vdash e_3 : \alpha_3 \mid C_3 \quad \alpha \text{ fresh}}{\Gamma \vdash e_1 ? e_2 : e_3 : \alpha \mid C_1 \cup C_2 \cup C_3 \cup { \alpha = \text{bool}, \alpha = \alpha_2, \alpha = \alpha_3 }} )</td>
<td>HMIF</td>
</tr>
<tr>
<td>( \frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \quad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \quad \alpha \text{ fresh}}{\Gamma \vdash e_1(e_2) : \alpha \mid C_1 \cup C_2 \cup { \alpha_1 \vdash (\alpha_2 \Rightarrow \alpha) }} )</td>
<td>HMCALL</td>
</tr>
<tr>
<td>( \frac{\Gamma' = \Gamma[x \mapsto \alpha_1] \quad \Gamma' \vdash e : \alpha_2 \mid C \quad \alpha, \alpha_1 \text{ fresh}}{\Gamma \vdash \text{function}(x) e : \alpha \mid C \cup { \alpha \vdash (\alpha_1 \Rightarrow \alpha_2) }} )</td>
<td>HMFUN</td>
</tr>
<tr>
<td>( \frac{\Gamma' = \Gamma[x \mapsto \alpha][y \mapsto \alpha_1] \quad \Gamma' \vdash e : \alpha_2 \mid C \quad \alpha, \alpha_1 \text{ fresh}}{\Gamma \vdash \text{function} x(y) e : \alpha \mid C \cup { \alpha = (\alpha_1 \Rightarrow \alpha_2) }} )</td>
<td>HMUNREC</td>
</tr>
</tbody>
</table>

Figure 6.3: Typing constraint generation rules for the Hindley-Milner type system
unify(C) = unify(C, ∅)  
unify(∅, σ) = σ  
unify(C, ⊥) = ⊥  
unify({τ₁ ⊑ τ₂} ∪ C, σ) =  
let σ' = unifyOne(τ₁ = τ₂, σ) in  
if σ' = ⊥ then ⊥ else unify(Cσ', σ')  
unifyOne(α ⊑ α, σ) = σ  
unifyOne(α ⊑ τ, σ) =  
if α ∈ tv(τ) then ⊥ else (λβ. σ(β)[τ/α])|α → τ]  
unifyOne(τ = α, σ) = unifyOne(α = τ, σ)  
unifyOne((τ₁ ⇒ τ₂) = (τ'_₁ ⇒ τ'_₂), σ) =  
unify({τ₁ ⊑ τ'_₁, τ₂ ⊑ τ'_₂}, σ)  
unifyOne(number ⊑ number, σ) = σ  
unifyOne(bool ⊑ bool, σ) = σ  
unifyOne(τ₁ ⊑ τ₂, σ) = ⊥  

Figure 6.4: A simple unification algorithm

unifyOne should be read top-down. The first matching case applies, similar to a match expression in a Scala program.

In the case for α = τ of unifyOne, the actual extension of the candidate unifier σ happens. The test α ∈ tv(τ) implements the occurrence check. Here, we denote by tv the function that takes a type expression τ and returns the set of type variables occurring in τ, e.g., tv(α ⇒ β) = {α, β}. If the occurrence check fails, unifyOne returns ⊥ to indicate that the constraint cannot be unified. If the check succeeds, unifyOne returns the extension of the candidate unifier σ, which is given by the expression:

\[ (λβ. σ(β)[τ/α])|α → τ] \]

The notation λx.e stands for an anonymous function. That is, the candidate unifier is a function σ'' that is obtained from σ in two steps. First, σ is updated to σ' by applying the mapping α → τ to all current mappings β → σ(β) in σ using type variable substitution:

\[ σ' : TVar \rightarrow Typ \]
\[ σ'(β) = σ(β)[τ/α] \]

Note that this definition of σ' is equivalent to the following definition which defines σ' in terms of an anonymous function:

\[ σ' = λβ. σ(β)[τ/α] \]
Then $\sigma''$ is obtained from $\sigma'$ by extending it with the new mapping $\alpha \mapsto \tau$:

$$
\sigma'' : TVar \rightarrow Typ
$$

$$
\sigma'' = \sigma'[\alpha \mapsto \tau].
$$

The case $(\tau_1 \Rightarrow \tau_2) = (\tau'_1 \Rightarrow \tau'_2)$ of `unifyOne` handles situations in which we need to unify a constraint that equates two function types. In this case, we simply need to recursively solve a new unification problem for the set of constraints $C' = \{\tau_1 = \tau'_1, \tau_2 = \tau'_2\}$. The recursion is well-defined (i.e., `unify` and `unifyOne` always terminate) since we only decompose function types in typing constraints but never create new function types during unification.

The final “catch-all” case of `unifyOne` handles all the cases $\tau_1 = \tau_2$ where $\tau_1$ and $\tau_2$ cannot be unified, such as `bool = number`, etc.

If we analyze the complexity of our simple unification algorithm we observe that it is worst-case quadratic in the number of typing constraints. This is because each time we extend the candidate unifier $\sigma$ for a constraint $\alpha \cdot = \tau$, we have to iterate over both the existing mappings in $\sigma$ as well as the unprocessed constraints. Practical implementations of the Hindley-Milner type system use more efficient unification algorithms that use a union-find data structure to represent the candidate unifier. Using this data structure the extension of the candidate unifier can be implemented more efficiently. In fact, these practical algorithms run in quasilinear time.

**Parametric Polymorphism**

We have not yet fully described how we can actually use expressions with polymorphic types in our programs and how such expressions are handled by the type inference algorithm. In the Hindley-Milner type system, the introduction of expressions with polymorphic types is closely tied to `const` declarations. Consider the following example

```const```
``id``
```const``
```function``
```id``
```function``
```id``
```id``
```id``
```id``
```id``
```id``
```id``
```id``
```id``
```id``
```id``
```id``

Recall from our discussion in Section 6.1.5 that this expression cannot be typed in our simple type system with annotated monomorphic types. However, it is well-typed in the Hindley-Milner type system.

The constraint generation for a `const` declaration works as follows. First, we generate the constraints for the defining expression of the declared variable as described before. In the example, the defining expression of `id` is `function(x) x`, which generates the following single typing constraint:

$$
\alpha_{id} = \alpha_x \Rightarrow \alpha_x
$$

Here, $\alpha_{id}$ represents the actual type of `id`. When we generate the typing constraints for the body of the `const` declaration, we do not simply reuse the same type variable $\alpha_{id}$ for each usage of `id` in the body. Instead, we use a fresh
Parametric Polymorphism (optional)

copy of the type variable and generate a fresh copy of the constraints obtained from the defining expression for each usage of \textit{id}. Here, “fresh copy” means that we consistently substitute all the type variables in the constraints by fresh type variables. The copying of constraints ensures that the inferred type for each usage of \textit{id} in the body of the declaration is consistent with the constraints imposed by \textit{id}’s definition. However, different usages of \textit{id} in the body do not interfere. In total, we end up with the following set of constraints for the complete expression in our example:

1. $\alpha_{id} \vdash \alpha_{x} \Rightarrow \alpha_{x}$
2. $\alpha_{id,1} \vdash \alpha_{\text{true}} \Rightarrow \alpha_{\text{call},1}$
3. $\alpha_{id,1} \vdash \alpha_{x,1} \Rightarrow \alpha_{x,1}$
4. $\alpha_{\text{true}} \vdash \text{bool}$
5. $\alpha_{\text{call},1} \vdash \text{bool}$
6. $\alpha_{id,2} \vdash \alpha_{1} \Rightarrow \alpha_{\text{call},2}$
7. $\alpha_{id,2} \vdash \alpha_{x,2} \Rightarrow \alpha_{x,2}$
8. $\alpha_{1} \vdash \text{number}$
9. $\alpha_{0} \vdash \text{number}$
10. $\alpha_{\text{call},1} \vdash \text{bool}$
11. $\alpha_{\text{ite}} \vdash \alpha_{\text{call},2}$
12. $\alpha_{\text{ite}} \vdash \alpha_{0}$

Note that the constraints 3 and 7 are the fresh copies of constraint 1 for the two usages of \textit{id} in the body of the \texttt{const} declaration. The resulting unification problem has a solution, which means that the expression is indeed well-typed.

To see the limitations of the kind of parametric polymorphism that is supported by the Hindley-Milner type system, contrast the expression

\texttt{const id = function(x)x; id(true) ? id(1) : 0}

with the expression

\texttt{(function(id)(id(true) ? id(1) : 0))(function(x)x)}

From a semantic point of view the two expressions are equivalent. We have simply replaced the \texttt{const} declaration of \textit{id} by function abstraction over \textit{id} in the body of the declaration, followed by a call that immediately binds \textit{id}’s defining expression to the parameter of the obtained function. As we have seen,
the first expression is well-typed. However, the second one is not. The problem is that during the typing constraint generation for the second expression, the two occurrences of \textit{id} in the function body are bound to the same type variable. That is, the generated constraints require that \textit{id} is at the same time of type

\[
\text{bool} \Rightarrow \text{bool}
\]

and of type

\[
\text{number} \Rightarrow \text{number}
\]

This is not possible. In order to support such expressions in the type system, we would have to consider the more general form of higher-ranked parametric polymorphism. The price we would have to pay for this generality is that type inference would no longer be possible in all cases.

The constraints that come from the defining expression of a \texttt{const} variable \(x\) are copied for each usage of \(x\) in the body of the declaration of \(x\). This means that the size of the generated constraints can grow exponentially with the nesting depth of \texttt{const} declarations in defining expressions. In fact, the type inference problem for the Hindley-Milner type system with \texttt{const} declarations is known to be \textsc{EXPTIME}-complete. So this exponential blow-up cannot be avoided in general. Actual implementations of the type system avoid this blow-up in practice by solving the generated constraints on the fly instead of separating the constraint generation phase from the unification phase.
Chapter 7

Imperative Programming

All the languages that we have studied so far have been purely functional. That is, the evaluation of expressions in these languages is side-effect free and (recursive) function call is the main computational device that drives the evaluation. Functional programming is contrasted by imperative programming where the main computational device is state mutation. We have argued that the functional programming paradigm has certain advantages over the imperative paradigm. In particular, the absence of side-effects makes it much easier to reason about the behavior of a program. Nevertheless, imperative programming is important in practice, specifically for performance critical code, as imperative language primitives map more directly to the underlying hardware architecture that ultimately executes a program. In this chapter, we will study the central language primitives of imperative programming languages.

7.1 Variables and Assignments

We extend our simple language from the previous chapter with the two central primitives of imperative programming languages: mutable variables and assignments. That is, we introduce state and mutation.

Before we formalize our new language extension, we study several examples of JavaScript programs that make use of mutable variables to better understand how these primitives work.

In JavaScript, mutable variables are declared by var declarations, which are like const declarations except that the keyword const is replaced by var:

\[ \text{var } x = e_d; e_b \]

The difference to a const declaration is that the binding of \( x \) to the value obtained from \( e_d \) can be modified in the body \( e_b \) using assignments

\[ x = e \]

For example, the following program declares a mutable variable \( x \) which is initialized to the value 1 and subsequently modified to \( x + 1 \):
This program evaluates to 2 since the final occurrence of x evaluates to the new value that x is updated to by the assignment.

As in most imperative programming languages, assignments are expressions that evaluate to a value – the value obtained from the right side of the assignment. For example, the following code also evaluates to 2.

```javascript
var x = 1;
x = x + 1;
x
```

In particular, assignments can be nested inside other assignments. For example, the following program nests assignments to x and y by first assigning x to the value 3 and then subsequently assigning the same value to y

```javascript
var x = 2;
var y = 2;
y = x = 3;
x + y
```

This program thus evaluates to 6.

At first, it may seem as if we could eliminate `var` declarations by replacing them with `const` declarations and also replacing every assignment to a variable x by a new `const` declaration that redeclares x with a new value, shadowing the previous binding of x. For example, we can rewrite our first program as follows:

```javascript
const x = 1;
{ const x = x + 1;
  x
}
```

The additional curly braces are needed because the scope of a `const` declarations in JavaScript is the entire basic block of statements in which the declaration occurs. By wrapping the second declaration in curly braces we start a new basic block. This program still evaluates to 2.

Unfortunately, the proposed elimination technique for `var` declarations only works for a straight-line sequence of declarations and assignments. If assignments are nested within function bodies, then the result of evaluation changes if we replace assignments by `const` declarations. For example, consider the following program

```javascript
var x = 2;
const f = function (y) { x = y; return x; };
f(3);
x
```

This program evaluates to 3. This is because when f is called, the nested assignment mutates the variable x to 3. This mutation is globally observable in
the entire scope of x. Hence, when we access x after the call returns, we obtain the new value S. On the other hand, consider the program

```
const x = 2;
const f = function (y) { const x = y; return x; }
f(3);
x
```

This program evaluates to 2 since the scope of x declared by the const declaration inside of f is restricted to the body of f. Thus, the occurrence of the variable x after the call to f refers to the declaration on the first line. Hence, we obtain the value S.

The fact that variable assignments have globally observable side-effects is one of the reasons why imperative programs are more difficult to reason about.

### 7.1.1 A Simple Language with Variables and Assignments

We start from the language in Chapter 6.1 and extend it with variable declarations, assignments, addresses, and a dereference operator. The grammar of the extended language is as follows:

- \( n \in \text{Num} \)  
  numbers (double)
- \( b \in \text{Bool} ::= \text{true} \mid \text{false} \)  
  Booleans
- \( a \in \text{Addr} = \mathbb{N} \)  
  addresses
- \( x \in \text{Var} \)  
  variables
- \( \tau \in \text{Typ} ::= \text{bool} \mid \text{number} \mid x:\tau_1 \Rightarrow \tau_2 \)  
  types
- \( v \in \text{Val} ::= n \mid b \mid a \mid \text{function} p(x:\tau) t e \)  
  values
- \( e \in \text{Expr} ::= x \mid v \mid e_1 \text{bop} e_2 \mid \text{uop} e_1 \mid e_1 \text{?} e_2 : e_3 \mid \text{mut} x = e_d; e_b \mid e_1(e_2) \)  
  expressions
- \( \text{bop} \in \text{Bop} ::= + \mid * \mid \&\& \mid | \mid === \mid !== \mid = \)  
  binary operators
- \( \text{uop} \in \text{Uop} ::= \ast \)  
  unary operators
- \( p ::= x \mid \epsilon \)  
  function names
- \( t ::= \tau \mid \epsilon \)  
  return type annotations
- \( \text{mut} \in \text{Mut} ::= \text{const} \mid \text{var} \)  
  mutabilities

Note that we introduce variable declarations by generalizing const declarations in our previous language to a declaration that is parameterized by a mutability, \( \text{mut} \in \text{Mut} \). A mutability mut is either var or const. We introduce addresses, \( a \in \text{Addr} \), as a new type of values that denote locations in memory. We define \( \text{Addr} = \mathbb{N} \). However, the specific definition of \( \text{Addr} \) is immaterial. We only rely on the fact that \( \text{Addr} \) is an infinite set (i.e., we never run out of

1Even if we assumed dynamic binding semantics, this program would still evaluate to 2.
addresses when we need fresh ones for memory allocation). The assignment operator \(=\) is a binary operator and the dereference operator \(*\) is a unary operator.

The role of addresses \(a\) and dereference expressions \(*e\) will become clear below. These primitives are included in the expression language because they arise during evaluation. However, there is no way to explicitly write these expressions in the source program (i.e., they are not part of the language’s concrete syntax). These primitives are an example of an enrichment of program expressions as an intermediate form solely for evaluation. Their role is thus similar to the value \texttt{typeerror} that we previously used to indicate a dynamic type error during evaluation.

### 7.1.2 Operational Semantics

Next, we adapt our operational semantics from Section 6.1 to account for the addition of mutable variables and assignments. We will see that adding state and mutation to our language forces us to do a rather global refactoring of our operational semantics.

**Modeling State**

We model the state of an expression using a mapping \(M\) that we refer to as the memory. The memory is both an input and output of the big-step evaluation relation. That is, the big-step judgment form is now as follows:

\[
\langle M, e \rangle \Downarrow \langle M', v \rangle
\]

This judgment says informally, “In memory \(M\), expression \(e\) evaluates to value \(v\) and the new memory \(M'\).” The presence of a memory \(M\) that gets updated during evaluation is the hallmark of imperative computation.

In our current language, we can only assign values to variables. If we think of this restriction in terms of computer systems architecture, this means that a memory \(M\) only needs to model the stack of activation records for function calls, which stores the values of all local variables that are currently in scope. It is therefore tempting to define \(M\) as a partial mapping from variable names to values, similar to our notion of environment that we used in the dynamic binding semantics discussed in Section 5.3.1. However, we want to maintain a static binding semantics in our new language. We therefore introduce one level of indirection to model memory access and mutation with static binding correctly. For this purpose, we introduce addresses and the dereference operator.

In our model, a memory \(M\) is a partial function from addresses to values, \(M : \text{Addr} \rightarrow \text{Val}\). We can then think of a \texttt{var} variable \(x\) as a \texttt{const} variable that stores an address to a memory location in \(M\). Whenever we use a \texttt{var} variable \(x\) in an expression, we implicitly dereference the address \(a\) stored in \(x\) to retrieve the value \(M(a)\) at the associated memory location in \(M\).
Inference Rules

The big-step evaluation relation is defined by the rules shown in Figures 7.1 and 7.2. As usual, the rules are syntax-driven. The rules for the new imperative language primitives are given in Figure 7.1. Observe the interplay between the rules EvalVarDecl, EvalDerefVar, and EvalAssignVar. The rule EvalVarDecl handles var declarations. It first evaluates the defining expression \( e_d \), as usual, to obtain its value \( v_d \) and an updated memory state \( M_d \). Then, a fresh memory address \( a \) is allocated to store the value \( v_d \) in \( M_d \). Allocation of a fresh address is modeled by nondeterministically choosing some address \( a \) that satisfies the condition \( a \not\in \text{dom}(M_d) \). Then, the memory state \( M_d \) is updated accordingly to obtain the new memory state \( M' = M_d[a \mapsto v_d] \). Next, the body \( e_b \) is updated by substituting all free occurrences of the declared variable \( x \) by the dereference expression \( \star a \). This ensures that the rules EvalDerefVar and EvalAssignVar look-up, respectively, modify the content of the memory location \( a \) that we associate with \( x \) when \( e_b \) is evaluated. This is in contrast to the EvalConstDecl rule for const declarations, where occurrences of \( x \) in \( e_b \) are replaced directly by the defining value \( v_d \). Note that the evaluation rule for assignments evaluates an assignment expression to the value \( v \) obtained from the expression on the right side of the assignment, as expected.

You may wonder why we need to introduce the dereference operations \( \star a \) in the EvalVarDecl rule instead of replacing \( x \) in \( e_b \) directly by \( a \). We need the additional dereference operation to model memory look-up, correctly. Since we defined addresses as values, the EvalDerefVar operation would otherwise have to evaluate a value and replace it by another value. This would conflict with the EvalVal rule. Also, we will see in Chapter 8 that we will be forced to treat addresses as values, once we extend our language with mutable objects.

The rules for the non-imperative constructs are given in Figure 7.2. The rules are essentially identical to those given in Section 5.3.4 except that we now have to thread the memory state \( M \) through all evaluation steps.

You might have noticed that in our operational semantics, the memory \( M \) only grows and never shrinks during the course of evaluation. Our semantics only ever allocates memory and never deallocates. This choice is fine in a
Figure 7.2: Big-step operational semantics of non-imperative primitives. The only changes compared to Figure 5.3 are the threading of the memory and the omission of implicit type conversions.
mathematical model, but a production run-time system must somehow enable collecting garbage–allocated memory locations that are no longer used by the running program. Collecting garbage may be done manually by the programmer (as in C and C++) or automatically by a conservative garbage collector (as in JavaScript, Scala, Java, C#, and Python).

7.1.3 Type Checking

Finally, we adapt the typing relation from Section 6.1 to account for the new language primitives. The interesting new case are assignment expressions $e_1 = e_2$. Specifically, the following expression should be considered well-typed:

\[ \text{var } x = 3; x = 5 \]

whereas the next expression should not be well-typed:

\[ \text{const } x = 3; x = 5 \]

That is, we should only allow assignments to variables that have actually been declared by a var declaration. In order to be able to distinguish the two cases above during typing, we have to provide additional information in the typing environment. Namely, in addition to the type of every free variable $x$ that occurs in the expression being typed, the typing environment must also record $x$’s mutability. We thus modify the signature of typing environments $\Gamma$ as follows:

\[ \Gamma : \text{Var} \rightarrow \text{Mut} \times \text{Typ} \]

The inference rules that define the new typing relation $\Gamma \vdash e : \tau$ are given in Figures 7.3 and 7.4. The rules in Figure 7.3 are identical to the corresponding rules discussed in Section 6.1 except that the signature of $\Gamma$ has changed. However, this change is irrelevant for these rules. The new, respectively, modified rules are all summarized in Figure 7.4. All variable binding constructs (i.e., declarations and function expressions) now also store the mutability of the declared name in the typing environment, together with the actual type inferred or annotated in the declaration. Note that function parameters and the names of recursive functions are considered to have const mutability. That is, function parameters cannot be reassigned in the function body. This is in contrast to JavaScript, where function parameters have var mutability. We will extend our language with the ability to reassign function parameters when we discuss parameter passing modes in Section 7.2.

The only rule that uses the mutability information in the typing environment is the rule for assignments, TYPEASSIGNVAR. This rule ensures that the left-hand side of an assignment is always a variable and that this variable has indeed been introduced using a var declaration, rather than a const declaration or a function abstraction. Further note that the rule also checks that the two sides of an assignment agree on the type $\tau$. That is, we do not allow a variable $x$ to be reassigned to a value whose type is different from the type
Figure 7.3: Type checking rules for non-imperative primitives (no changes compared to Figure 6.2).

of $x$’s initialization expression. This is in contrast to JavaScript, which allows such reassignments since JavaScript is dynamically typed. The restriction to type-consistent reassignment in the rule TypeAssignVar is crucial for proving the preservation property of the new static typing relation.

There are no rules for typing addresses $a \in Addr$ and dereference operations $* e$ since these expressions are only introduced during evaluation.

### 7.2 Parameter Passing Modes

In this section, we will systematically explore the different language design choices for implementing parameter passing in function calls.

#### 7.2.1 Parameter Passing Variants

We extend our JAKARTASCRIPT fragment with four parameter passing modes:

- call by value
- call by name
- call by variable, and
- call by reference.
We distinguish these parameter passing modes by prefixing function parameters with a dedicated keyword that determines the mode. Before we introduce the formal semantics of these modes, we explain the differences between them using a series of examples.

Call by Value

So far, we have evaluated function calls using call by value semantics. In this mode, before the actual function call happens, the argument expression is first reduced to a value, which is then passed to the function body. In our new language, we denote call by value parameters by prefixing them with the keyword \texttt{const}. As an example, consider the following program, which defines a function \texttt{f} with a call-by-value parameter \texttt{x}:

```javascript
const f = function(const x: number) (x + x);
const y = 3;
const r = f(y + 1);
console.log(y);
console.log(r);
```

This program will print 3 and 8. Since the call to \texttt{f} in this program is side-effect free, we cannot observe the specific evaluation order of the call by value semantics. To make the evaluation order explicit, consider the following modified version of the program:
const \( f \) = \( \text{function}(\text{const } x: \text{number}) \ (x + x) \);
var \( y \) = 3;
const \( r \) = \( f(y = y + 1) \);
console.log(\( y \));
console.log(\( r \));

This program will print 4 and 8. The assignment expression \( y = y + 1 \) that is the argument to the call to \( f \) is executed once before the call. The value 4, which is the result of the assignment, is then passed into the function to compute the result of the call, which is 8.

Call by Name

One alternative to the call by value passing mode is to change the evaluation order of function calls and call arguments so that the function call happens before the argument is evaluated. If we define the semantics in such a way that we reevaluate the argument each time the parameter is used in the function body, we speak of call by name parameter passing.

In our new extension of \textit{JakartaScript}, we indicate call by name parameters by prefixing the parameter with the keyword \textit{name}. As a first example, consider the following modified version of our first program above where we change the call by value parameter \( x \) in function \( f \) to a call by name parameter:

\[
\text{const } f = \text{function} (\text{name } x: \text{number}) \ (x + x) ; \\
\text{const } y = 3 ; \\
\text{const } r = f(y = y + 1) ; \\
\text{console.log}(y) ; \\
\text{console.log}(r) ;
\]

This program will print 3 and 8 just like in the version with the call by value parameter. Again, we have to introduce side effects in the call to \( f \) in order to make the evaluation order explicit. To this end, consider the call by name version of the second program above:

\[
\text{const } f = \text{function} (\text{name } x: \text{number}) \ (x + x) ; \\
\text{var } y = 3 ; \\
\text{const } r = f(y = y + 1) ; \\
\text{console.log}(y) ; \\
\text{console.log}(r) ;
\]

This program will now print 5 and 9 because the assignment \( y = y + 1 \) is executed two times during the evaluation of \( f \)–once for each usage of the call by name parameter \( x \) in \( f \). Depending on the control flow in the body of the called function, the argument to a call by name parameter may not be evaluated at all. For instance, consider the following variant of our program:

\[
\text{const } f = \text{function} (\text{const } b: \text{bool}) \{
\text{  function} (\text{name } x: \text{number}) \ (b ? x + x : 0) 
\}
\]
This program will print 3 and 0 because the call to `f` will not evaluate the “then” branch \( x + x \) of the conditional expression in the body of `f`. Hence, the argument \( y = y + 1 \) that is passed by name to `x` is never used in the call and hence never evaluated.

Call by name parameter passing is a very useful programming feature. For example, suppose we have a function whose argument value is only used in the function body if certain conditions are satisfied, e.g., a logging function might only use its argument value if the program is run in debugging mode. In such cases, we would like to avoid the evaluation of the argument altogether in the cases where the value is not actually used. We can easily achieve this by using a call by name parameter.

**Simulating Call by Name.** Unfortunately, many programming languages only support call by value parameters. However, if a language supports higher-order functions and function abstraction, we can simulate call by name using call by value. The idea is to delay the execution of the argument until after the function call happened by wrapping the argument in a function abstraction. Whenever the argument is used in the body of the called function, it has to be explicitly unwrapped using an auxiliary function call that recalculates the argument value.

To see how this works, consider the following JavaScript program:

```javascript
const f = function(x) { x() + x();
var y = 3;
const r = f(function () { y = y + 1});
console.log(y);
console.log(r);
```

This program prints 5 and 9, just like our second example for call by name parameter passing. Observe that we turned the call by name parameter `x` of type `number` into a call by value parameter of a function type. That is, `x` is now a function that takes no parameters. Each call to `x` in `f`'s body will cause the wrapped argument expression to be reevaluated, which gives us the same behavior as a call by name parameter.

**Call by Variable**

The difference between call by value and call by name parameters is the order in which the call and the arguments to the call are evaluated. However, we can make a more fine-grained distinction in our semantics of parameter passing, even if we fix the evaluation order to call by value semantics. Specifically, we can distinguish between parameters that are treated as constant values.
throughout the execution, and parameters that are essentially treated like mutable variables and can be reassigned new values. We refer to the latter type of parameters as \textit{call by variable} parameters. We indicate such parameters by prefixing the parameter name with the keyword \texttt{var}. Essentially, the difference between a \texttt{const} parameter and a \texttt{var} parameter is that inside the function body, a \texttt{var} parameter is treated as if it was declared by a \texttt{var} declaration instead of a \texttt{const} declaration. The following example highlights this difference:

\begin{verbatim}
const f = function(var x: number) (x = x + 1; x);
var y = 3;
const r = f(y);
console.log(y);
console.log(r);
\end{verbatim}

This program will print 3 and 4. The parameter variable \texttt{x} is reassigned inside of the body of \texttt{f}. However, the effect of this assignment is not observable outside of \texttt{f}.

Note that in JavaScript (and most other imperative programming languages) function calls are implemented using call by variable semantics. Often, these languages do not make the fine-grained distinction between call by variable and call by value that we make here and both modes are simply referred to as call by value.

\textbf{Call by Reference}

Finally, we can consider a combination of call by name and call by variable. We refer to this mode as \textit{call by reference}. This mode is indicated by the keyword \texttt{ref}. The following program highlights the difference between call by variable and call by reference:

\begin{verbatim}
const f = function(ref x: number) (x = x + 1; x);
var y = 3;
const r = f(y);
console.log(y);
console.log(r);
\end{verbatim}

This program will print 4 and 4. Unlike in the previous program where we passed the argument to \texttt{f} by variable, the assignment to \texttt{x} is now treated as an assignment to the mutable variable \texttt{y}, which is passed to \texttt{x} in the call to \texttt{f}. Note that we can only pass assignable expressions as arguments to call by reference parameters. We refer to such expressions as \textit{location expressions}. In our current language, only mutable variables are location expressions. For example, the following program should be rejected by our type checker because \texttt{f} attempts to reassign \texttt{y} which has been declared as an immutable \texttt{const} variable:

\begin{verbatim}
const f = function(ref x: number) (x = x + 1; x);
const y = 3;
const r = f(y); // type error because y is not assignable
console.log(y);
\end{verbatim}
console.log(r);

### 7.2.2 A Simple Language with Parameter Passing Modes

The abstract syntax of our extended language is as follows:

\[
\begin{align*}
    n & \in \text{Num} & \text{numbers (double)} \\
    b & \in \text{Bool} ::= \text{true} | \text{false} & \text{Booleans} \\
    a & \in \text{Addr} = \mathbb{N} & \text{addresses} \\
    x & \in \text{Var} & \text{variables} \\
    \tau & \in \text{Typ} ::= \text{bool} | \text{number} | (\text{mode} \ \tau_1) \Rightarrow \tau_2 & \text{types} \\
    v & \in \text{Val} ::= n | b | a | \text{function} \ p(\text{mode} \ x : \tau) \ t \ e & \text{values} \\
    e & \in \text{Expr} ::= x | v | e_1 \ bop \ e_2 | uop \ e_1 | e_1 ? e_2 : e_3 | \text{expressions} \\
    \text{mut} \ x = e_d; e_b | e_1(e_2) & \\
    bop & \in \text{Bop} ::= + | * | \&\& | \| | = & \text{binary operators} \\
    uop & \in \text{Uop} ::= * & \text{unary operators} \\
    p & ::= x | \epsilon & \text{function names} \\
    t & ::= \tau | \epsilon & \text{type annotations} \\
    \text{mut} & \in \text{Mut} ::= \text{const} | \text{var} & \text{mutabilities} \\
    \text{mode} & \in \text{PMode} ::= \text{const} | \text{var} | \text{name} | \text{ref} & \text{passing modes}
\end{align*}
\]

The only change compared to the language of Section 7.1 is that we now explicitly declare the parameter passing mode for each function abstraction. The parameter passing mode also becomes part of the type signature of a function.

### 7.2.3 Operational Semantics

Figure 7.5 shows the new big-step evaluation rules for function calls with the different parameter passing modes. The rules for the other language constructs are as before. Note that the rule \text{SEARCHCALL}_2 only applies to the call-by-value and call-by-variable modes. For a call to a function whose parameter is passed by reference, the type system will ensure that the argument of the call is always of the form \text{*} \ a. In this case, the argument \text{*} \ a should not be evaluated before the call, so that the address can be passed into the function body. Similarly, for a function whose parameter is passed by name, the argument should not be evaluated before the call.

### 7.2.4 Type Checking

The new type checking rules for the different types of function abstractions and call expressions are given in Figure 7.6. When the typing environment is extended with the parameter in the different \text{TYPEFUN} rules, the given parameter passing mode is mapped to the appropriate mutability using the function \text{mut}. 
\[
\begin{align*}
\langle M, e_1 \rangle \Downarrow \langle M', v_1 \rangle & \quad v_1 = \textbf{function} \langle \textbf{const} \; x : \tau_2 \rangle \; t \; e \\
\langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle & \quad e' = e[v_2/x] \quad \langle M'', v_2 \rangle \Downarrow (M''', v) \\
& \quad \langle M, e_1(e_2) \rangle \Downarrow (M''', v) \\
\langle M, e_1 \rangle \Downarrow \langle M', v_1 \rangle & \quad v_1 = \textbf{function} \langle x_1 \; \textbf{const} \; x_2 : \tau_2 \rangle \; t \; e \\
\langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle & \quad e' = e[v_1/x_1][v_2/x_2] \quad \langle M'', v_2 \rangle \Downarrow (M''', v) \\
& \quad \langle M, e_1(e_2) \rangle \Downarrow (M''', v) \\
\langle M, e_1 \rangle \Downarrow \langle M', v_1 \rangle & \quad v_1 = \textbf{function} \langle x_1 \; \textbf{var} \; x_2 : \tau_2 \rangle \; t \; e \\
\langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle & \quad a \not\in \text{dom}(M'') \quad e' = e[\ast \; a/x] \\
& \quad \langle M''[a \mapsto v_2], e' \rangle \Downarrow (M''', v) \\
& \quad \langle M, e_1(e_2) \rangle \Downarrow (M''', v) \\
\langle M, e_1 \rangle \Downarrow \langle M', v_1 \rangle & \quad v_1 = \textbf{function} \langle x_1 \; \textbf{name} \; x_2 : \tau_2 \rangle \; t \; e \\
& \quad e' = e[v_1/x_1][v_2/x_2] \quad \langle M', e' \rangle \Downarrow (M''', v) \\
& \quad \langle M, e_1(e_2) \rangle \Downarrow (M''', v) \\
\langle M, e_1 \rangle \Downarrow \langle M', v_1 \rangle & \quad v_1 = \textbf{function} \langle x_1 \; \textbf{ref} \; x_2 : \tau_2 \rangle \; t \; e \\
& \quad e' = e[\ast \; a/x] \quad \langle M', e' \rangle \Downarrow (M''', v) \\
& \quad \langle M, e_1(\ast \; a) \rangle \Downarrow (M''', v) \\
\langle M, e_1 \rangle \Downarrow \langle M', v_1 \rangle & \quad v_1 = \textbf{function} \langle x_1 \; \textbf{ref} \; x_2 : \tau_2 \rangle \; t \; e \\
& \quad e' = e[v_1/x_1][\ast \; a/x_2] \quad \langle M', e' \rangle \Downarrow (M''', v) \\
& \quad \langle M, e_1(\ast \; a) \rangle \Downarrow (M''', v)
\end{align*}
\]

Figure 7.5: New inference rules that define the big-step semantics of function call expressions for the different parameter passing modes
\[\begin{align*}
\Gamma \vdash e_1 : (\text{mode } \tau_2) \Rightarrow \tau & \quad \Gamma \vdash e_2 : \tau_2 \quad \text{mode} \notin \{\text{ref}\} \\
\Gamma \vdash (e_1(e_2)) : \tau & \quad \text{TYPECALL} \\
\end{align*}\]
\[\begin{align*}
\Gamma \vdash e : (\text{ref } \tau_2) \Rightarrow \tau & \quad x \in \text{dom}(\Gamma) \quad \Gamma(x) = (\text{var}, \tau_2) \\
\Gamma \vdash e(x) : \tau & \quad \text{TYPECALLRef} \\
\end{align*}\]
\[\begin{align*}
\Gamma' = \Gamma[x \mapsto (\text{mut(mode)}, \tau_2)] & \quad \Gamma' \vdash e : \tau \\
\Gamma' \vdash \text{function}(\text{mode } x : \tau_2) e : (\text{mode } \tau_2) \Rightarrow \tau & \quad \text{TYPEFUN} \\
\end{align*}\]
\[\begin{align*}
\Gamma' = \Gamma[x \mapsto (\text{mut(mode)}, \tau_2)] & \quad \Gamma' \vdash e : \tau \\
\Gamma' \vdash \text{function}(\text{mode } x : \tau_2) : \tau e : (\text{mode } \tau_2) \Rightarrow \tau & \quad \text{TYPEFUNANN} \\
\end{align*}\]
\[\begin{align*}
\Gamma' = \Gamma[x_1 \mapsto (\text{const}, \tau_1)][x_2 \mapsto (\text{mut(mode)}, \tau_2)] & \quad \Gamma' \vdash e : \tau \\
\Gamma' \vdash e : \tau & \quad \tau_1 = (\text{mode } \tau_2) \Rightarrow \tau \\
\Gamma' \vdash \text{function } x_1 (\text{mode } x_2 : \tau_2) : \tau e : \tau_1 & \quad \text{TYPEFUNREC} \\
\end{align*}\]
\[\begin{align*}
\text{mut(const)} &= \text{mut(name)} = \text{const} \\
\text{mut(var)} &= \text{mut(ref)} = \text{var} \\
\end{align*}\]

Figure 7.6: New type checking rules for function and call expressions

Note that at the moment mutable variables are the only expressions that are allowed as arguments to functions whose parameters are passed by reference. This is because in our current language mutable variables are the only expressions that evaluate to references to memory locations.

### 7.2.5 Custom Control Constructs with Call by Name

One useful feature of call-by-name parameters is that it can be combined with curried higher-order functions to define custom control constructs. We discuss how this can be done in our current JAKARTA SCRIPT fragment as well as in Scala.

**JAKARTA SCRIPT while loop.** Loops are one of the most important control constructs in imperative languages. We do not have loops built into our current JAKARTA SCRIPT fragment. However, using call-by-name parameter passing, we can write recursive functions that can be used almost as if they were loops:

```javascript
const while = function while(name cond: bool):
    (name Undefined) => Undefined 
{
    return function (name body: Undefined) {
                return cond ? (body, while(cond)(body)) : undefined
    }
```
In the above JAKARTA SCRIPT program, we define a curried function `while` that takes a value of type `boolean`, the loop condition, and a value of type `Undefined`, the loop body, to implement the semantics of a `while` loop in JavaScript. By passing the condition and body by name, they are reevaluated each time a call to the nested function of `while` is evaluated. This way, we obtain the proper semantics of a `while` loop.

When we use the `while` function, the only syntactic difference to an actual `while` loop in JavaScript is that the loop body has to be wrapped in an extra pair of parenthesis. The reason for this is that we are actually calling the function that resulted from the first call to `while`.

Another minor issue is that we have to terminate the body of the loop with an explicit `undefined` value. If we instead wrote:

```javascript
var x = 0;
while(x < 10){
  console.log(x);
  x = x + 1;
  undefined
}
```

then this program would be rejected by our type system. This is because the type of a sequence of expressions is calculated as the type of the last expression in the sequence. In the case of the loop body in the second program, this is now the assignment expression `x = x + 1`. However, the type of an assignment expression is the type of the right side of the assignment, which is `number`. This type is incompatible with the type `Undefined`, which is the expected type of the argument to the function that is returned by `while`.

**Scala repeat/until loop.** In Scala, a parameter can be declared as pass-by-name by putting an arrow in front of the parameter type:

```scala
def f(x: => T) e
```

Thus, a call-by-name parameter in Scala can be thought of as a function that is called with no argument and then returns a value of type `T`. This syntax is reminiscent of our encoding of call-by-name parameters using call-by-value parameters with function types that take no parameters.

We can combine call-by-name parameters with Scala’s object system and its condensed method call syntax. This gives us a powerful technique for defining custom language primitives that can be used as if they were built into the language.
For example, some languages such as Pascal support repeat/until loops:

```scala
repeat body until (cond)
```

These loops execute `body` once, and then repeatedly execute it until the loop condition `cond` becomes true. Although, Scala does not have repeat.until loops built in, we can easily write a class that provides us with such a construct:

```scala
class repeat(body: => Unit) {
  def until(cond: => Boolean): Unit = {
    body
    if (!cond) until(cond)
  }
}
```

```scala
object repeat {
  def apply(body: => Unit) = new repeat(body)
}
```

The class `repeat` takes the loop body as a parameter and then defines a method `until` that takes the loop condition to implement a repeat/until loop using recursion. The type `Unit` is Scala’s equivalent of the type `Undefined` in JAKARTAJS. Since both the loop body and condition are passed by name, we obtain the expected behavior. The companion object of `repeat` defines a factory method to create new `repeat` instances, saving us the explicit calls to `new`.

We can then use this class as follows:

```scala
var x = 0
repeat {
  x = x + 1
} until (x == 10)
```

It now seems as if repeat/until is indeed an in-built language construct. However, this code is just a syntactically more compact but semantically equivalent version of the following nested sequence of method calls:

```scala
var x = 0
repeat.apply({
  println(x)
  x = x + 1
}).until(x == 10)
```

In particular, the first call goes to the `apply` method of the companion object of `repeat`, the subsequent `until` call then goes to the newly created `repeat` instance that is returned by the call to `apply`.

### 7.3 Monads

We have seen that introducing state and mutation complicates the operational semantics of our language because the memory state now becomes part of the in-
put and output of our big-step evaluation relation. Even for the non-imperative primitives in our language, we now have to thread the memory state through the individual evaluation steps. This increases the amount of “plumbing” we have to do in our interpreter implementation. In this section, we will study a class of data structures referred to as monads. Specifically, we will learn about the so called state monad. The state monad allows us to encapsulate the additional computational overhead for threading the memory state in our interpreter and to avoid exposing this complexity in the actual implementation of the step relation.

### 7.3.1 A Stateful JakartaScript Interpreter

We start our exposition with a straightforward Scala implementation of the small-step operational semantics that we formalized in class 18. As a first step, we define the abstract syntax of our language with variables and assignments using case classes, as usual. The Scala code is shown in Figure 7.7. We focus on the new imperative primitives of our language, the other primitives are elided. The type `mem` represents memory states. It is defined as an alias to the type `map{addrL val}`, which represents partial mappings from addresses to values.

We can now directly translate the small-step operational semantics into Scala code. Specifically, we implement the step relation $M, e \vdash M', e' \rightarrow$ from the previous class by a function `step` with the following signature:

```scala
def step(m: Mem, e: Expr): (Mem, Val)
```

That is, `step` takes the input memory state and an expression and returns the new memory state and value obtained from evaluating the given expression in the input memory state.

In Figure 7.8, we show a stub of the implementation of the `eval` function of the interpreter. We only show the implementation of the `EVALVALUNUMUS`, and `EVALPLUS` rule. The code closely follows the corresponding inference rules of the big-step evaluation relation. As we can see, the additional threading of the memory state $m$ dilutes the simplicity of the stateless implementation of our purely functional JAKARTASCRIPT variants. The goal of this section is to restore that simplicity.

### 7.3.2 Monads and for Expressions in Scala

As a precursor for the simplification of our stateful interpreter, we study Scala’s `for` expressions. The `for` expression primitive provides a generic way for iterating over collections of data and for building new collections from existing ones. The following example shows how a `for` expression can be used to iterate over a list $l$ to build a new list by applying some function to the elements of $l$:

```scala
scala> val l = List(2,5)
l: List[Int] = List(2,5)

scala> for (x <- l) yield x + 1
```
sealed abstract class Expr
sealed abstract class Val extends Expr
...

/** mutabilities */
sealed abstract class Mut
case object MConst extends Mut
case object MVar extends Mut

/** declarations */
case class Decl(mut: Mut, x: String, e1: Expr, e2: Expr) extends Expr

/** binary operators */
case class BinOp(bop: Bop, e1: Expr, e2: Expr) extends Expr

sealed abstract class Bop
case object Times extends Bop
case object Plus extends Bop
case object Assign extends Bop

/** unary operators */
case class UnOp(uop: Uop, e1: Expr, e2: Expr) extends Expr

sealed abstract class Uop
case object Deref extends Uop

.../*/
case class Addr(a: Int) extends Val

/** memory states */
type Mem = Map[Addr, Val]

Figure 7.7: Representing in Scala the abstract syntax of JAKARTASCRIPT with variables and assignments
def eval(m: Mem, e: Expr): (Mem, Val) = {
    def eToNum(m: Mem, e: Expr): (Mem, Int) = {
        val (mp, v) = eval(m, e)
        v match {
            case Num(n) => (mp, n)
            case _ => throw StuckError(e)
        }
    }
    e match {
        /** rule EvalVal */
        case v: Val => (m, v)
        /** rule EvalUMinus */
        case UnOp(UMinus, e1) =>
            val (mp, n1) = eToNum(e1)(m)
            (mp, Num(- n1))
        /** rule EvalPlus */
        case BinOp(Plus, e1, e2) =>
            val (mp, n1) = eToNum(m, e1)
            val (mpp, n2) = eToNum(m, e2)
            (mpp, Num(n1 + n2))
        ...
    }
}

def evaluate(e: Expr): Val = {
    val (_, v) = eval(Mem.empty, e)
    v
}

Figure 7.8: Partial implementation of the eval function of the stateful interpreter
When we look at the result of the above for expression, we can see that it is really computing a map over the list \( l \). In fact, the Scala compiler simply rewrites the for expression

\[
\text{for } (x <- l) \text{ yield } x + 1
\]

to the following expression, which calls map on \( l \) with an anonymous function that is built from the yield part of the for expression:

\[
l \text{ map } (x => x + 1)
\]

Thus, a for expression is really just syntactic sugar for a call to map.

One useful feature of for expressions is that they can be nested. As an example, the following nested for expression computes the “Cartesian product” of the list \( l \) with itself:

\[
\text{scala> for } (x <- l; y <- l) \text{ yield } (x, y)
\]

\[
\text{res1: List[(Int, Int)] = List((2,2), (2,5), (5,2), (5,5))}
\]

If we naively expand this for expression to map calls as described above, we obtain the following result:

\[
\text{scala> l map (x => l map (y => (x, y)))}
\]

\[
\text{res2: List[List[Int]] = List(List(2,2), List(2,5), List(5,2), List(5,5))}
\]

The result value res2 is a list of list of pairs, rather than a list of pairs. In order to obtain res1 from res2, we need to flatten res2 by concatenating the inner lists to a single list of pairs. Incidentally, the List class provides a method flatten that does just that:

\[
\text{scala> res2.flatten}
\]

\[
\text{res3: List[Int] = List((2,2), (2,5), (5,2), (5,5))}
\]

For convenience, the class List also provides a method flatMap that corresponds to the composition of map and flatten. Using flatMap we can express the nested for expression that produced res1 more compactly as follows:

\[
\text{scala> l flatMap (x => l map (y => (x, y)))}
\]

\[
\text{res4: List[Int] = List((2,2), (2,5), (5,2), (5,5))}
\]

This translation pattern generalizes to for expressions of arbitrary nesting depth. In general, the Scala compiler will translate a for expression of the form

\[
\text{for } (x_n <- e_n; \ldots; x_2 <- e_2; x_1 <- e_1) \text{ yield } e_0
\]

to the expression

\[
e_n \text{ flatMap } (x_n => \ldots e_2 \text{ flatMap } (x_2 => e_1 \text{ map } (x_1 => e_0))\ldots)
\]
Monads. The use of for expressions is not restricted to the List type. It works for any type that provides a map and a flatMap method with the appropriate signatures. For example, the Option type also provides these functions and can hence be used in for expressions:

```scala
scala> for (x <- Some(0)) yield x + 1
res5: Option[Int] = Some(1)
```

```scala
scala> for (x <- None) yield x + 1
res6: Option[Int] = None
```

We refer to a class that has appropriate map and flatMap methods as a monad. One can think of a monad as an abstract data type that implements a container for data and provides generic functions for transforming this data without extracting it from the container. Using for expressions we can then conveniently express a sequence of such transformations that operate directly on the contained data.

The monad-as-container correspondence is easy to see for the type List and also for the type Option. The latter can be thought of as a list of length at most 1. In general, monads can be more abstract and it is sometimes more difficult to understand the nature of the contained data. We will see one such example in the next section. Many of the classes that are provided by the Scala standard API are monads, including all of the collection classes. Some of the more interesting monads provided by Scala are Try and Future.

The Theory of Monads. As an aside, the term monad is lent from category theory, a branch of mathematics that is concerned with the theory of mathematical structures and the morphisms between them. The programming language and category theoretic concepts of a monad are closely related. In category theory, monads are defined in terms of certain algebraic laws that relate the flatMap and map functions. For example, these laws codify how map can be expressed in terms of flatMap and vice versa. These laws also ensure that for expressions in Scala behave the way they are expected to behave.

7.3.3 The State Monad

To simplify the implementation of our stateful interpreter given in Figure 7.8, we introduce the state monad. The state monad allows us to hide the threading of the memory state between the different calls to the step function. In other words, the state monad takes care of the plumbing so that we can focus our attention on the interesting bits of the interpreter implementation. Before we introduce the state monad, we first refactor the interpreter given in Figure 7.8 into a semantically equivalent version that will help us understand what it is that the state monad actually does.

Curried Interpreter. The transformation that we apply to the interpreter is as follows: we modify the signature of the eval function from:
def eval(e: Expr): Mem => (Mem, Val) = {
  def eToNum(e: Expr): Mem => (Mem, Int) = { m =>
    val (mp, v) = eval(e)
    v match {
      case Num(n) => (mp, n)
      case _ => throw StuckError(e)
    }
  }
  e match {
    /** rule EvalVal */
    case v: Val => { m => (m, v) }
    /** rule EvalUMinus */
    case UnOp(UMinus, e1) => { m =>
      val (mp, n1) = eToNum(e1)(m)
      (mp, Num(- n1))
    }
    /** rule EvalPlus */
    case BinOp(Plus, e1, e2) => { m =>
      val (mp, n1) = eToNum(e1)(m)
      val (mpp, n2) = eToNum(e2)(mp)
      (mpp, Num(n1 + n2))
    }
    ...
  }

  def evaluate(e: Expr): Val = {
    val (_, v) = eval(e)(Mem.empty)
    v
  }
}

Figure 7.9: Curried version of the interpreter shown in Figure 7.8
def eval(m: Mem, e: Expr): (Mem, Val)

to

def eval(e: Expr): Mem => (Mem, Val)

More precisely, we turn eval into a curried function that first takes the input expression e and then returns a new function. This new function can then be applied to the input memory state m to compute the output state and the result value of the evaluation (mp, v), as before. From a caller's point of view, the eval function behaves exactly as before, except that every call eval(m, e) has to be replaced by a curried call eval(e)(m). The new curried implementation of the interpreter is shown in Figure 7.9. The new version looks more complicated than before. Though, as is often in life, things first have to get worse before they get better.

Implementing the State Monad. The state monad is a parameterized type State[S, R]. The essence of State[S, R] is that it is a container that encapsulates a function of type S => (S, R), which can be seen as a computation that takes an input state of type S and returns an output state together with a result value of type R. Looking at the result type Mem => (Mem, Val) of the curried eval function, we can see that it is precisely a computation of this form. That is, we can encapsulate the result of a call eval(e) in a state monad State[Mem, Val]. The map and flatMap functions of this monad will then take care of threading the memory state m through a sequence of calls to eval.

The implementation of the state monad is shown in Figure 7.10. The encapsulated computation of type S => (S, R) is held in the field run, which is a parameter of the class State. We discuss the map and flatMap methods in more detail.

The map method transforms the given State[S, R] into a State[S, P] by composing a stateless computation f: R => P with the encapsulated stateful computation run. It is instructive to compare the body of the map method with the implementation of the EvalUMinus rule in eval shown in Figure 7.9. We can see that the latter can be obtained from the former by substituting

- run by eToNum(e1),
- s by m,
- sp by mp,
- r by n1, and
- f by { n1 => Num(- n1)}.

That is, the implementation of EvalUMinus corresponds to a call to map on the result of eToNum(e1) encapsulated in a state monad.

Similar to map, the flatMap method transforms the given State[S, R] into a State[S, P] by composing run with another computation f. The difference to
sealed class State[S,R](run: S => (S,R)) {
    def apply(s: S) = run(s)

    def map[P](f: R => P): State[S,P] =
        new State[S,P]({ s =>
            val (sp, r) = run(s)
            (sp, f(r))
        })

    def flatMap[P](f: R => State[S,P]): State[S,P] =
        new State[S,P]({ s =>
            val (sp, r) = run(s)
            f(r)(sp) // same as f(r).apply(s)
        })
    }

object State {
    def insert[S,R](r: R): State[S,R] =
        new State[S,R]({ s => (s, r) })
}

Figure 7.10: Implementation of the state monad

map is that now f is itself stateful and returns a state monad. If we again carefully
analyze the code in Figure 7.9 we observe that the case implementing the rule
for \texttt{EvalPlus} corresponds to a call to \texttt{flatMap} on the result of \texttt{etoNum(e1)}
encapsulated in a \texttt{State[Mem,Val]}. Here, \texttt{etoNum(e1)} takes the role of \texttt{run} in
\texttt{flatMap} and f is replaced the function

\[
\begin{array}{l}
\{ \text{n1 => new State[Mem, Val]({mp => val (mpp, n2) = etoNum(e2)
                                                   (mpp, Num(n1 + n2)) })} \\
\end{array}
\]

Finally, we add a companion object for the State class that provides a
factory method insert for inserting a result value r into a state monad. The
encapsulated computation simply returns r together with the unmodified input
state s.

Using the State Monad. With the implementation of the state monad in
place, we can refactor the interpreter in Figure 7.9 as shown in Figure 7.11. In
particular, we are using map and flatMap to thread the memory state through
the recursive calls to eval as described above. The threading of the memory
def eval(e: Expr): State[Mem, Val] = {
def eToNum(e: Expr): State[Mem, Val] =
  eval(e) map { v =>
    v match {
      case Num(n) => n
      case _ => throw StuckError(e)
    }
  }

  e match {
    /** rule EvalVal */
    case v: Val => State.insert(v)
    /** rule EvalUMinus */
    case UnOp(UMinus, e1) =>
      eToNum(e1) map { n1 =>
        Num(- n1)
      }
    /** rule EvalPlus */
    case BinOp(Plus, e1, e2) =>
      eToNum(e1) flatMap { n1 =>
        eToNum(e2) map { n2 =>
          Num(n1 + n2)
        }
      }
  }

  ...
}

def evaluate(e: Expr): Val = {
  val (_, v) = eval(e)(Mem.empty)
  v
}

Figure 7.11: Monadic version of the interpreter shown in Figure 7.9
def eval(e: Expr): State[Mem, Val] = {
    def eToNum(e: Expr): State[Mem, Val] =
        for (v <- eval(e)) yield v match {
            case Num(n) => n
            case _ => throw StuckError(e)
        }

    e match {
        /** rule EvalVal */
        case v: Val => State.insert(v)
        /** rule EvalUMinus */
        case UnOp(UMinus, e1) =>
            for {
                n <- eToNum(e1)
            } yield Num(-n1)
        /** rule EvalPlus */
        case BinOp(Plus, e1, e2) =>
            for {
                n1 <- eToNum(e1)
                n2 <- eToNum(e2)
            } yield Num(n1 + n2)
            ...
    }
}

def evaluate(e: Expr): Val = {
    val (_, v) = eval(e)(Mem.empty)
    v
}

Figure 7.12: Variant of the interpreter shown in Figure 7.11 with for expressions

states is now completely hidden inside of the state monad. Note that the rule EvalVal in eval is implemented with a call to the insert method of State's companion object. To simplify our implementation further, we can additionally hide the map and flatMap calls using for expressions as shown in Figure 7.12.
Chapter 8

Object-Oriented Programming

In this chapter, we will study the basic primitives of object-oriented programming languages. We will extend Jakartascript with a simple object system. We will stay faithful to the semantics of objects in Javascript while restricting the new language features to the bare minimum. Certain features of Javascript’s object system, such as prototype objects, will not be supported directly by our language extension. Instead, we will study how we can implement the essential features of object-oriented (OO) programming languages by using only the very simple primitives of our minimal language extension. We will then extend our simple type system from Chapter 6 with subtyping and study how subtyping relates to the central notion of object-oriented programming: the substitution principle.

8.1 Objects and Fields

We start from the language that we discussed in Section 7.1 and add object literals and field dereference operations. For now, we omit types from our new language. We first describe the semantics of the new language constructs informally.

8.1.1 A Simple Untyped Language with Objects

An object literal is a comma-separated sequence of field names with mutabilities (const or var) and initialization expressions surrounded by braces:

\[
\{\text{mut}_1 f_1 \colon e_1, \ldots, \text{mut}_n f_n \colon e_n\}
\]

When an object literal is evaluated, we create an object. An object is a mapping from field names to values. Evaluating an object literal creates an object that
maps the fields $f_i$ to the values $v_i$ obtained from evaluating the initialization expressions $e_i$. The created object is stored in memory and the object literal itself evaluates to the address of that memory location.

To access the value of a field $f$ in an object, we use field dereference operations, $e.f$. Here $e$ must evaluate to an address $a$ of a memory location that contains an object. The operation $e.f$ then evaluates to the value of the field $f$ in the object stored at address $a$.

A field of an object is mutable if it is prefixed with the mutability `var`. On the other hand, if it is prefixed with mutability `const` then it is immutable. Mutable fields can be reassigned new values using assignment expressions of the form $e_1.f = e_2$. The effect of such an assignment is that the object at the memory location given by $e_1$ will be modified by setting its field $f$ to the value obtained from $e_2$. In JavaScript, there are no mutability modifiers for fields in object literals and all fields have `var` mutability. For compatibility with JavaScript we make the field mutability modifiers optional and the mutability of a field defaults to `var` if the modifier is omitted. We add `const` fields to our language because they give us more flexibility once we extend the language with a type system.

**Aliasing.** Since objects are referenced with an extra level of indirection through an address, two program variables can reference the same object. We refer to such a situation as aliasing. With mutation, aliasing is now observable as demonstrated by the following example:

```javascript
const x = { var f : 1 };
const y = x;
x.f = 2;
y.f
```

This program will evaluate to 2 because `x` and `y` are aliases (i.e., reference the same object). The interaction between aliasing and mutation makes programs more difficult to reason about and is often the source of subtle bugs.

**Abstract Syntax.** For the time being, we remove types from our language. We will discuss typing issues related to objects in Section 8.3. The abstract
syntax of our new language is given by the following grammar:

\[
\begin{align*}
\text{n} & \in \text{Num} & \text{numbers (double)} \\
\text{b} & \in \text{Bool} ::= \text{true} | \text{false} & \text{Booleans} \\
\text{a} & \in \text{Addr} = \mathbb{N} & \text{addresses} \\
\text{x} & \in \text{Var} & \text{variables} \\
\text{f} & \in \text{Fld} & \text{field names} \\
\text{v} & \in \text{Val} ::= \text{n} | \text{b} | \text{a} | \text{function} p(x) e & \text{values} \\
\text{lv} & \in \text{LVal} ::= * \text{a} | \text{a}.f & \text{location values} \\
\text{e} & \in \text{Expr} ::= x | v | e_1 \text{bop} e_2 | \text{uop} e_1 | e_1 ? e_2 : e_3 | \text{mut} x = e_d; e_b | e_1(e_2) | e.f \{ \text{mut}_1 f_1 : e_1, \ldots, \text{mut}_n f_n : e_n \} & \text{expressions} \\
\text{le} & \in \text{LExpr} ::= x | e.f & \text{location expressions} \\
\text{bop} & \in \text{Bop} ::= + | * | \&\& | || | = & \text{binary operators} \\
\text{uop} & \in \text{Uop} ::= * & \text{unary operators} \\
\text{p} & ::= x | \epsilon & \text{function names} \\
\text{mut} & \in \text{Mut} ::= \text{const} | \text{var} & \text{mutabilities}
\end{align*}
\]

We introduce location expressions, \( \text{le} \in \text{LExpr} \), as a separate semantic domain. Location expressions are the expressions that we allow to occur on the left side of assignment expressions. Similarly, we introduce location values to denote the expressions that location expressions in assignments reduce to during evaluation.

### 8.1.2 Operational Semantics

Figure 8.1 shows the big-step evaluation rules for the new language constructs. A memory state \( M \) is now a partial function from addresses to values and object values. We formalize this by introducing a semantic domain of memory contents:

\[ k \in \text{Con} ::= v | \{ f_1 : v_1; \ldots; f_n : v_n \} \]

and define

\[ M \in \text{Mem} = \text{Addr} \rightarrow \text{Con} \]

Note that the fields of object values in memory have no mutabilities. The type system that we will discuss in Section 8.3 will statically ensure that immutable fields cannot be reassigned. Hence, we can erase the mutability information at evaluation-time.

The EvalObj rule formalizes the allocation of a fresh memory location to store the result of evaluating an object literal. The EvalDerefFld rule formalizes the semantics of field dereference via memory look-up. Finally, the rule
\[
M_0 = M \quad \text{for all } i \in [1, n] : \langle M_{i-1}, e_i \rangle \Downarrow \langle M_i, v_i \rangle \\
a \notin \text{dom}(M_n) \quad M' = M_n[a \mapsto \{f_1:v_1, \ldots, f_n:v_n\}] \\
\langle M, e \rangle \Downarrow \langle M', a \rangle \quad \text{EVALOBJ} \\
\langle M, e \rangle \Downarrow \langle M', a \rangle \quad M(a) = \{\ldots, f:v, \ldots\} \quad \text{EVALDEREFLD} \\
\langle M, e_1 \rangle \Downarrow \langle M', a \rangle \ \langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle \\
M''(a) = \{\ldots, f:v, \ldots\} \quad M'' = M''[a \mapsto \{\ldots, f:v_2, \ldots\}] \quad \text{EVALASSIGNFLD}
\]

Figure 8.1: New evaluation rules for the language primitives related to objects

EVALASSIGNFLD formalizes the assignment to a field by updating the corresponding field of the object value at the given memory location.

The evaluation rules for the remaining language constructs are as in Section 7.1.2 except that we do not have type annotations and that memories now store both values and object values.

8.2 Basic Features of OO Languages

We now show how the essential features of object-oriented languages can be realized using the language primitives that we introduced above. We discuss simple objects, classes and data encapsulation, inheritance, method overriding, calls to super class methods, and open recursion. We use the concrete syntax of JavaScript so that you can easily try out the examples using a JavaScript interpreter.

8.2.1 Methods

We start by introducing simple objects. An object in object-oriented programs encapsulates state and provides methods to access and manipulate this state. We also refer to the state of the object as the representation of the object.

As an example, we implement an object that encapsulates a counter value \(x\). The counter object provides two methods: a method \(\text{get}\) that reads the current counter value, and a method \(\text{inc}\) that increments the counter. We can declare such an object in our language using an object that has two fields holding functions that implement the get and inc methods:

```javascript
const rep = { x: 0 }; 
const counter = { 
  get: function() { return rep.x; }, 
  inc: function() { rep.x = rep.x + 1; }
};
```
You may wonder why we declared the field \(x\) that holds the counter value in a separate object, \(\text{rep}\), instead of adding it directly to the \(\text{counter}\) object. There are two reasons for this. First, if \(x\) was a field of \(\text{counter}\), then any client who has a reference to \(\text{counter}\) would be able to modify the state of the counter directly by assigning new values to the field \(x\). This would break the OO philosophy of hiding the representation of an object from the clients of the object. Most object-oriented languages provide mechanisms for controlling the visibility of fields. We do not have such mechanisms in place, yet. Hence, we put the representation of the \(\text{counter}\) object in another object \(\text{rep}\) that is read and written by the \(\text{get}\) and \(\text{inc}\) methods. A client of the \(\text{counter}\) object who does not have a direct reference to the \(\text{rep}\) object will not be able to modify the state of \(\text{counter}\) directly.

The second reason why we cannot add the field \(x\) to \(\text{counter}\) directly is that the methods \(\text{inc}\) and \(\text{get}\) must be able to access \(x\). However, we cannot access a field of an object before the object has been created. Most OO languages (including JavaScript) support access to the fields of an object from inside the object’s methods through a special variable called \(\text{this}\) (or \(\text{self}\)). The \(\text{this}\) reference is only bound to the actual object instance after the object has been created. This feature is called open recursion. Instead of building this feature directly into our language, we will later see how to implement it using the language primitives that we already have at our disposal.

Here is a simple client of our counter object that calls \(\text{inc}\) three times and then returns the new counter value:

```javascript
const counterClient = function(c) {
  c.inc();
  c.inc();
  c.inc();
  return c.get();
};
```

Evaluating `counterClient(counter)` will return 3, as expected.

### 8.2.2 Classes and Encapsulation

So far, we have only constructed a single object instance. Constructing objects this way is rather tedious. Moreover, we have not yet achieved actual encapsulation of the state of our counter object, since we still have a global variable \(\text{rep}\) to the counter representation. Hence, clients can still by-pass the methods of the object and access the internal state directly. We will solve both of these problems at once.

Many OO languages are based on classes. A class can be thought of as a template that describes how new instances of a specific type of objects can be created. The class mechanisms of most languages tend to be rather complicated because they are often the only language construct for coarse-grain structuring of programs. Hence, classes often provide many features that are orthogonal to their main purpose. We here focus on the basic features provided by a class.
We implement a counter class as a function that takes the internal representation of a counter object and then returns a counter object instance:

```javascript
const counterClass = function(rep) {
  return {
    get: function() { return rep.x; },
    inc: function() { rep.x = rep.x + 1; }
  }
};
```

Using the function `counterClass` we can now define a constructor that creates new counter objects on demand:

```javascript
const newCounter = function() {
  const rep = {x: 0};
  return counterClass(rep);
};
```

The scope of the reference to the `rep` object of the newly created counter object is now restricted to the body of the constructor. Hence, the clients of counter objects can no longer read or modify the counter values directly without calling the `get` and `inc` methods.

Also, note that each counter object has its own internal representation. For example, the following code evaluates to 0 because the counter value of `counter2` is not modified by the calls to `inc` on `counter`:

```javascript
const counter = newCounter();
const counter2 = newCounter();
counterClient(counter);
counter2.get()
```

**Inheritance**

One of the most important features of OO languages is that they provide an inheritance mechanisms to augment objects with additional functionality in a modular fashion.

As an example, we define a new class of reset counter objects that extends the counter class. A reset counter provides the same functionality as a counter object. In addition, it provides a method `reset` that sets the internal counter value back to 0. The following function implements our reset counter class:

```javascript
const resetCounterClass = function(rep) {
  const _super = counterClass(rep);
  return {
    get: _super.get,
    inc: _super.inc,
    reset: function() { rep.x = 0; }
  }
};
```
The function `resetCounterClass` first calls `counterClass` with the given `rep` object to create a `counter` object `_super`. Then it creates a reset counter object using the `get` and `inc` methods of `_super` and adds the `reset` method. Thus, the reset counter class effectively inherits the `get` and `inc` methods from `counterClass`. We refer to the extended class as the `subclass` and to the class that is extended as the `superclass`.

The constructor for reset counter objects is almost identical to that of counter objects:

```javascript
const newResetCounter = function() {
    const rep = {x: 0};
    return resetCounterClass(rep);
};
```

### Substitution Principle

One of the key properties of object-oriented programs is that objects of sub-classes can be safely used in any context that expects an object of a superclass. Here, “safe” means “type safe”, i.e., the evaluation of a program in which an object of a superclass has been replaced by an object of a subclass cannot get stuck.

For example, we can safely call `counterClient` on a reset counter object as in the following program:

```javascript
const counter = newResetCounter();
counterClient(counter);
counter.reset();
counter.get()
```

This program will evaluate to `0`.

We refer to the principle of safe replacement of superclass objects by subclass objects as the substitution principle. We will study this principle more thoroughly when we add a type system to our object-oriented version of JAKARTA SCRIPT.

### Extending the Internal Representation and Overriding Methods

When we derive a subclass from a superclass we cannot only add new methods but we can also add new fields to the internal representation. Moreover, we can override the methods of the superclass with new methods that change the behavior of subclass instances.

As an example, we define a class of backup counter objects. This class extends the reset counter class with a new method `backup` that stores the current counter value in an additional field `y`. The field `y` is added to the internal representation. The backup counter class then overrides the `reset` method so that the counter is set back to the backed up value stored in field `y`:
const backupCounterClass = function(rep) {
    const _super = resetCounterClass(rep);
    return {
        get: _super.get,
        inc: _super.inc,
        reset: function() { rep.x = rep.y; },
        backup: function() { rep.y = rep.x; }
    }
};

const newBackupCounter = function() {
    const rep = {x: 0, y: 0};
    return backupCounterClass(rep);
};

8.2.3 Calling Superclass Methods

When we extend a superclass with new methods or when we override existing methods, it is often useful to implement the new functionality by calling the existing methods provided by the superclass.

For example, suppose we want to implement a variant of our backup counter class that backs up the counter value each time inc is called. Then, we can implement the new inc method using the backup and inc methods of the backup counter class as follows:

const funnyBackupCounterClass = function(rep) {
    const _super = backupCounterClass(rep);
    return {
        get: _super.get,
        inc: function() { _super.backup(); _super.inc(); },
        reset: _super.reset,
        backup: _super.backup
    }
};

Note that the call _super.inc() will go to the inc method defined in the function backupCounterClass.

8.2.4 Open Recursion

Finally, we consider the problem of open recursion via a self reference this. Open recursion is often useful, e.g., to implement one method using calls to other methods of the same class. For example, suppose we want to implement a counter object that provides a set method, which can be used to set the counter to a given value. Then we can implement inc using calls to get and set on the same object instance.
There are essentially two ways how we can implement open recursion in our language. First, we can make the function that implements our counter class recursive so that it first creates an object instance where the behavior of the methods is only partially defined. Then it calls itself recursively to “tie the knot”, binding the created instance to this to get the intended behavior. However, this approach is rather inefficient because we will end up constructing multiple object instances.

There exists a more efficient solution that closely follows the actual implementation of open recursion in OO languages. In this solution we tie the recursive knot by using the indirection provided by references to the memory in combination with state mutation via assignments.

The trick is to change the class function so that it takes a reference, say _this, to an object in memory that already provides all the necessary fields. The function implementing the counter class then simply modifies the _this object by updating its fields to the appropriate values:

```javascript
const setCounterClass = function(_this) {
  _this.x = 0;
  _this.get = function() { return _this.x; };
  _this.set = function(i) { _this.x = i; };
  _this.inc = function() { _this.set(_this.get() + 1); };
  return _this;
};
```

The constructor method now simply needs to create a dummy object instance that provides the correct fields, initialized with some placeholder values of the appropriate types. The reference to the dummy instance is then passed to the class function, which initializes the fields to the correct values:

```javascript
const newSetCounter = function() {
  const dummy = {
    x: 0,
    get: function() { return 0; },
    set: function(i) {},
    inc: function() {}
  );
  return setCounterClass(dummy);
};
```

## 8.3 Subtyping

In this section, we will extend the type system of JA\textsc{kartaScript} to account for objects and provide typing rules that formalize our intuition of the substitution principle in object-oriented programs.
8 Object-Oriented Programming

\[ \Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n \]
\[ \Gamma \vdash \{ \text{mut}_1 f_1 : e_1, \ldots, \text{mut}_n f_n : e_n \} : \{ \text{mut}_1 f_1 : \tau_1, \ldots, \text{mut}_n f_n : \tau_n \} \]

\[ \Gamma \vdash e : \{ \text{mut}_1 f_1 : \tau_1, \ldots, \text{mut}_n f_n : \tau_n \} \]
\[ f = f_i \quad \tau = \tau_i \quad i \in [1, n] \]
\[ \Gamma \vdash e.f : \tau \]

\[ \Gamma \vdash e_1 : \{ \ldots \text{var} f : \tau \ldots \} \quad \Gamma \vdash e_2 : \tau \]
\[ \Gamma \vdash e_1.f \equiv e_2 : \tau \]

Figure 8.2: Typing rules for objects

8.3.1 Typing Objects

First, we extend our type language with a dedicated type \textbf{any} and a type constructor for object types:

\[ \tau \in \text{Typ} ::= \ldots | \textbf{any} | \{ \text{mut}_1 f_1 : \tau_1, \ldots, \text{mut}_n f_n : \tau_n \} \]

The type \textbf{any} will simplify certain aspects of our type system. We will discuss it in more detail later. An object type is like an object expression, except that the expressions of the individual fields are replaced by types.

The rules that extend our typing relation \( \Gamma \vdash e : \tau \) to account for objects are given in Figure 8.2. The rule \text{TypeObj} infers the object type of an object literal \( \{ \text{mut} f : e \} \) by recursively inferring the types of the initialization expressions \( e \) of the fields \( f \). The rules for field dereference, \text{TypeDerefFld}, and assignment to a field, \text{TypeAssignFld}, are straightforward. Note that \text{TypeAssignFld} requires that the type of the assigned expression is equal to the type of the field that is being assigned to. Moreover, the updated field must have mutability \textbf{var}. This requirement is analogous to the rule \text{TypeAssignVar} for typing assignments to variables.

Substitution Principle Revisited

Consider the following simple program:

```javascript
const fun = function(x: { f: number }) {
    return 2 * x.f;
};
fun({ f: 1 })
```

In this program it should be safe to replace the argument \( \{ f:1 \} \) of the call to function \( \text{fun} \) by, e.g., \( \{ f:1,g:2 \} \). The new program can still be safely evaluated. In fact, it will reduce to exactly the same value. However, the call \( \text{fun}({ f:1,g:2 }) \) is not well-typed according to our current typing rules. The problem is the rule \text{TypeCall}, which requires that the type of the argument in a call expression precisely matches the type of the function parameter that
it is passed to. This condition is violated in the call `fun({f: number, g: number})`, since the argument has type `{f: number, g: number}` whereas the parameter of function `fun` has type `{f: number}`. In the following, we extend our typing rules so that the modified call to `fun` is well-typed.

### 8.3.2 Structural Subtyping

In order to extend our type system to account for the substitution principle, we identify the situations when we are allowed to replace a value of one type with a value of another type without violating type safety. That is, we still want to guarantee that the evaluation of a well-typed program will never get stuck. We formalize the substitution principle by introducing the *subtype relation*. The subtype relation is a binary relation on types, denoted \( \tau <: \tau' \). The judgment form \( \tau <: \tau' \) states that values of type \( \tau' \) can be safely substituted by values of type \( \tau \). Informally, this means that all the things that can be done with values of type \( \tau' \) can also be done with values of type \( \tau \). If \( \tau <: \tau' \) holds, we say that \( \tau \) is a *subtype* of \( \tau' \) or, conversely, that \( \tau' \) is a *supertype* of \( \tau \).

We incorporate the subtype relation into our type system by extending our typing rules with the rule `TypeSub` shown in Figure 8.3. The rule states that if an expression \( e \) has the inferred type \( \tau \) and \( \tau \) is a subtype of \( \tau' \), then \( e \) also has type \( \tau' \). The rule `TypeSub` thus captures the essence of the substitution principle.

It remains to define the actual subtype relation. We here use *structural subtyping* to define this relation. This means that we define subtyping by comparing the structure of the type expressions. For instance, TypeScript, a typed version of JavaScript, uses structural subtyping in its type checker. An alternative to structural subtyping is *nominal subtyping*, which is used by most class-based object oriented languages including Java. In nominal subtyping, the subtype relation must be defined explicitly in the program using class inheritance. The question whether one type is a subtype of another can then be checked by looking up the relationship of the corresponding class names in the inheritance tree. Scala uses a combination of nominal and structural subtyping.

Our subtype relation is defined by the rules in Figure 8.4. We discuss the individual rules in more detail.

**General properties of subtyping.** First, we state some basic properties of the subtype relation. The type `any` takes a special role in our new type system. It describes all values and is thus a supertype of all other types. This property is captured by the rule `SubAny`. Many typed object-oriented languages have such a most general supertype. For example, in Scala this type is called `Any`. Here is the type rule expressing this:

\[
\frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}
\]

**Figure 8.3:** The subtyping rule
Java does not have a most general supertype because primitive types such as \texttt{int} do not have proper supertypes in Java. However, Java has an object type that is a supertype of all other object types, namely the type \texttt{Object}.

From the perspective of writing programs, the type \texttt{any} is not very useful on its own. If we determine that the type of an expression is \texttt{any}, then the only way we can interact with the resulting value is to pass it around, e.g., to a function that expects a value of type \texttt{any}. Nevertheless, the type \texttt{any} is still useful. As we will see in the next section, having a most general supertype simplifies some aspects of our type inference algorithm. Moreover, most object-oriented languages provide cast operators that enable the programmer to down-cast a value from a supertype to a subtype. The safety of such casts is then checked at evaluation time. The combination of having a most general supertype and down-casts provides a simple way of writing polymorphic code in object-oriented languages. In fact, in early versions of the Java language, the use of down-casts in combination with the most general object type \texttt{Object} was the only available mechanism for implementing generic data structures\footnote{Generics were not included in the Java language before version 5.0.}

Next, every type should be a subtype of itself, i.e., the subtype relation is reflexive. This is captured by the rule \texttt{SubRefl}. Similarly, the subtype relation should be transitive, i.e., if $\tau_1$ is a subtype of $\tau_2$ and $\tau_2$ a subtype of $\tau_3$, then $\tau_1$ should also be a subtype of $\tau_3$. This is captured by the rule \texttt{SubTrans}.

**Subtyping of objects.** The crucial part of the subtype relation are the rules that govern when an object type is a subtype of another object type. These rules formalize our intuition of what subtyping in object-oriented languages is really about. There are three notions of object subtyping that we can distinguish. First, we can obtain a subtype of an object type by adding more fields to

\begin{align*}
\tau <: \texttt{any} & \quad \text{\texttt{SubAny}} & \tau <: \tau & \quad \text{\texttt{SubRefl}} & \tau_1 <: \tau_2 <: \tau_3 & \quad \text{\texttt{SubTrans}} \\
\{\text{mut } f:\tau, \text{mut' } g:\tau'\} <: \{\text{mut } f:\tau\} & \quad \text{\texttt{SubObjWidth}} \\
\tau <: \tau' & \quad \text{\texttt{SubRefl}} \\
\{\ldots \text{const } f:\tau\ldots\} <: \{\ldots \text{const } f:\tau'\ldots\} & \quad \text{\texttt{SubObjDepth}} \\
\{\ldots \text{var } f:\tau\ldots\} <: \{\ldots \text{const } f:\tau\ldots\} & \quad \text{\texttt{SubObjMut}} \\
\{\ldots \text{mut } f:\tau, \text{mut' } g:\tau'\ldots\} <: \{\ldots \text{mut' } g:\tau', \text{mut } f:\tau\ldots\} & \quad \text{\texttt{SubObjPerm}} \\
\tau'_1 <: \tau_1 <: \tau_2 <: \tau'_2 & \quad \text{\texttt{SubFun}} \\
(\tau_1 \Rightarrow \tau_2) <: (\tau'_1 \Rightarrow \tau'_2) & \quad \text{\texttt{SubFun}}
\end{align*}

Figure 8.4: The inference rules that define the subtyping relation
it. This is captured by the rule \texttt{SUBOBJWIDTH}. Width subtyping closely corresponds to class inheritance since an inherited class may extend its superclass with additional fields and methods. In languages with nominal subtyping such as Java, width subtyping is typically the only way to put two different types in the subtype relation.

In our structural subtyping relation, we consider additional forms of object subtyping. Specifically, consider the following program:

```javascript
const f = function(x: {const y: {var z: number}}) {
    return x.y.z;
};
f({const y: {z: 0, w: 2}})
```

The function \(f\) expects an argument that is an object with a \texttt{const} field \(y\), which itself stores an object with a field \(z\) of type \texttt{number}. When we call \(f\), we provide an object whose field \(y\) provides both a field \(z\) and a field \(w\) of type \texttt{number}. Thus, the call can be safely evaluated. We can capture this in our type system by making the type \(\{\texttt{const} y: {z: number}\}\) a subtype of \(\{\texttt{const} y: {z: number, w: number}\}\). Note that this subtype relationship is not yet covered by the rule \texttt{SUBOBJWIDTH}.

More generally, we can obtain a subtype of an object type by replacing the type \(\tau\) of one of its \texttt{const} fields by a subtype \(\tau'\) of \(\tau\). This is captured by the rule \texttt{SUBOBJDEPTH}. We refer to this form of subtyping as \textit{depth subtyping}. Note that we only allow depth subtyping for \texttt{const} fields but not for \texttt{var} fields. We explain the rationale for this restriction in more detail below. If we want to use depth-subtyping for a \texttt{var} field, we first need to demote it to a \texttt{const} field. This idea is captured by the \texttt{SUBOBJMUT} rule. That is, this rule captures the fact that a \texttt{var} field has more capabilities than a \texttt{const} field (it can be both read and written instead of just read).

Finally, we observe that the order in which we list the fields in an object does not matter for the safety of evaluation. Hence, we can obtain a subtype of an object type by permuting its fields. This is captured by the rule \texttt{SUBOBJPERM}. Note that this subtyping rule affects the efficiency of the interpreter implementation. If field permutation is disallowed, then the relative position of the value for each field in an object is fixed a priori. This means that field dereference and field assignment operations can be implemented efficiently using direct memory accesses. On the other hand, if field permutation is allowed, then the representation of objects in memory involve an additional level of indirection. That is, each object must be represented as a map from fields to values. A field dereference operation or assignment operation then involves a look-up of the appropriate entry for the given field in the object. Most object-oriented languages that allow field permutation implement objects as hash maps to obtain constant-time field look-up operations. However, a hash map look-up is still more expensive than a direct memory access. Most compiled OO languages therefore disallow field permutation for the sake of efficiency whereas interpreted languages allow field permutations because it gives more flexibility to the programmer.

The three object subtyping rules can be stitched together using the tran-
sitivity rule \text{SubTrans}. For example, the following derivation shows that the object type \{\texttt{const}\ g: \texttt{bool}, \texttt{var}\ f: \texttt{bool}\} is a subtype of \{\texttt{var}\ f: \texttt{bool}\}:

\[
\{\texttt{const}\ g: \texttt{bool}, \texttt{var}\ f: \texttt{bool}\} <: \{\texttt{var}\ f: \texttt{bool}, \texttt{const}\ g: \texttt{bool}\} \quad \text{SubObjPerm}
\]

\[
\{\texttt{var}\ f: \texttt{bool}\} <: \{\texttt{var}\ f: \texttt{bool}\} \quad \text{SubObjWidth}
\]

\[
\{\texttt{const}\ g: \texttt{bool}, \texttt{var}\ g: \texttt{bool}\} <: \{\texttt{var}\ f: \texttt{bool}\} \quad \text{SubTrans}
\]

\textbf{Subtyping of functions.} \textsc{JakartaScript} supports function values. This raises the question whether we can formalize a subtyping rule for function types that captures the substitution principle for function values. That is, when can a function expression safely replace a function expression of another type? We are thus looking for a rule that places two function types in the subtype relation:

\[
(\tau_1 \Rightarrow \tau_2) <: (\tau'_1 \Rightarrow \tau'_2)
\]

The question is then how the parameter types \(\tau_1\) and \(\tau'_1\), respectively, the result types \(\tau_2\) and \(\tau'_2\) should be related so that this subtype relationship is safe. To this end, consider the following example program:

1. ```
   \texttt{const}\ f1 = \texttt{function}\ (x: \{\texttt{var}\ y: \texttt{number}\})\{ 
   \texttt{return}\ {z: x.y + x.y}; 
   \};
   
   \texttt{const}\ f2 = \texttt{function}\ (x: \{\texttt{var}\ y: \texttt{number}\})\{ 
   \texttt{return}\ {z: x.y + x.y, w: \texttt{true}}; 
   \};
   
   \texttt{const}\ x = f1({y: 2});
   
   x.z
   ```

The type of \(f1\)'s return value is \{\texttt{var}\ z: \texttt{number}\} and the type of \(f2\)'s return value is \{\texttt{var}\ z: \texttt{number}, \texttt{var}\ w: \texttt{bool}\}. Clearly, it is safe to replace the call to \(f1\) on line 7 by a call to \(f2\) because all the capabilities of the object returned by \(f1\) (reading and updating field \(z\)) are also provided by the object returned by \(f2\). This observation generalizes: it is safe to substitute a function \(f1\) by a function \(f2\) if the \(f2\) provides stronger guarantees about its return type than \(f1\). That is, the first requirement for the subtype relationship

\[
(\tau_1 \Rightarrow \tau_2) <: (\tau'_1 \Rightarrow \tau'_2)
\]

to hold is that \(\tau_2 <: \tau'_2\) holds. We say that function types are \textit{covariant} in their result types to indicate that the subtype relation keeps the same direction when we recurse into the result types.

What about the types of the function parameters \(\tau_1\) and \(\tau'_1\)? It may seem tempting to also require \(\tau_1 <: \tau'_1\). However, this would be unsafe as the following program demonstrates:

1. ```
   \texttt{const}\ f1 = \texttt{function}\ (x: \{\texttt{var}\ y: \texttt{number}\})\{ 
   \texttt{return}\ x.f + x.f;
   ```
The parameter type of \( f_2 \) is a subtype of \( f_1 \)'s parameter type. Observe that it is not safe to replace the call to \( f_1 \) on line 8 with a call to \( f_2 \). We pass an object to \( f_1 \) that only provides a field \( y \) but function \( f_2 \) accesses fields \( y \) and \( z \) of its argument. However, we can safely replace the call to \( f_2 \) on line 9 with a call to \( f_1 \). The object passed object to \( f_2 \) has fields \( y \) and \( z \) but \( f_1 \) only accesses field \( y \). Again this observation generalizes: it is always safe to replace a function by another function that makes fewer demands on the type of its argument. That is, for

\[
(\tau_1 \Rightarrow \tau_2) <: (\tau'_1 \Rightarrow \tau'_2)
\]

to hold we must require \( \tau'_1 <: \tau_1 \). We say that function types are *contravariant* in their parameter type to indicate that the subtype relation is inverted when we recurse into the parameter types. The resulting subtyping rule for function types is given by the rule SubFun.

### 8.3.3 Subtyping and Assignments

The purpose of a type system is to detect problems in programs that may lead to run-time errors during evaluation. So far, we have been extra careful in the design of our static type systems so that we obtain strong correctness guarantees for well-typed programs. Namely, we want to guarantee that the evaluation of a well-typed program will never get stuck. It turns out that with the introduction of subtyping we have introduced a flaw in our type system and this safety guarantee no longer holds. This flaw results from the interaction between the subtyping rule and the typing rule for assignment expressions. To understand this problem, consider the following program:

```javascript
const x = { f: { g: true } };  
x.f = {};  
x.f.g
```

When we evaluate this program starting with some memory state \( M \), then after evaluating the defining expression of \( x \), we get a memory state like this

\[
M' = M[a_1 \mapsto \{ g: \text{true} \}][a_2 \mapsto \{ f: a_1 \}]
\]

and we are left to evaluate the expression:

```javascript
a_2.f = {};  
a_2.f.g
```
After evaluating the assignment expression, we obtain the memory state

\[ M'' = M'[a_3 \mapsto \{\}] | a_2 \mapsto \{f:a_3\}] \]

Note that we have allocated a new object at address \( a_3 \) and have updated the value of the field \( f \) of the object at \( a_2 \) to point to \( a_3 \). We are now left with the expression

\[ a_2 . f . g \]

Evaluating the first field dereference operation \( a_2 . f \) in \( M'' \) will yield the address \( a_3 \). The final field dereference operation that is left to be evaluated will now be \( a_3 . g \). However, at this point we are stuck because the object stored at \( a_3 \) in \( M'' \) is the empty object \( \{\} \). Thus, the EVALDEREFFLD rule for evaluating field dereference operations does not apply because \( \{\} \) has no field \( g \).

Unfortunately, the above program is well-typed according to our current typing relation. This means that our typing relation does not have the progress property, which says that the evaluation of well-typed programs never gets stuck. Consequently, our current type system is not safe.

To see that the program is indeed well-typed. Let us look more closely at the critical step in the type inference. After typing the defining expression of \( x \), the typing environment \( \Gamma \) stores the inferred type of \( x \), which is:

\[ \Gamma(x) = (\text{const}, \{\text{var } f : \{\text{var } g : \text{bool}\}\}) \]

The types of the expressions \( x . f = \{} \) and \( x . f . g \) are inferred independently using the binding of \( x \) in \( \Gamma \). It is easy to see that \( x . f . g \) is well-typed under this binding of \( x \). To see that the assignment \( x . f = \{} \) is also well-typed we look at the complete subderivation for that expression:

\[
\begin{array}{c}
\Gamma(x) = (\text{const}, \{\text{var } f : \{\text{var } g : \text{bool}\}\}) \\
\Gamma \vdash x : \{\text{var } f : \{\text{var } g : \text{bool}\}\} \quad \text{TYPEVAR} \\
\Gamma \vdash x . f : \{\text{var } g : \text{bool}\} \quad \text{TYPEDEREFFLD} \\
\{\text{var } g : \text{bool}\} <: \{} \quad \text{SUBOBJWIDTH} \\
\Gamma \vdash \text{type } \{\} \quad \text{TYPESUB} \\
\Gamma \vdash \{\} : \{\} \quad \text{TYPEOBJ} \\
\Gamma \vdash x . f = \{} : \{} \\
\text{TYPEASSIGNFLD}
\end{array}
\]

We can see that the TYPESUB rule provides the “glue” between the TYPEASSIGNFLD rule and the TYPEDEREFFLD rule. It allows us to relax the type \( \{g : \text{bool}\} \) inferred for \( x . f \), the left side of the assignment, to its supertype \( \{\} \), which is the type that we infer for the right side of the assignment.

The problem with our current type system is that it allows the rule TYPESUB to be used too liberally. The typing rules do not account for the fact that location expressions \( e . f \) can play two different roles during program evaluation. An \( e . f \) expression that is evaluated by the EVALDEREFFLD rule serves as a source of a value that is read from memory. On the other hand, an expression \( e . f \) that occurs on the left side of an assignment and is evaluated by the
EVALASSIGNFLD rule serves as a sink of a value that is stored into memory. In a source position, it is always safe to replace a location expression e.f of type \( \tau \) by an expression of a subtype \( \tau' \). That is, source positions are typed covariantly. However, for a sink position, it is only safe to replace \( e.f \) by an expression that is a supertype of \( \tau \), i.e., sink positions must be typed contravariantly. In other words, when we write a value into memory, then that value must at least provide all the capabilities that are guaranteed by the type of the memory location to which we write. For example, when the type of \( e.f \) is that of a reset counter object, then we should not be allowed to reassign it to a counter object. Otherwise, we may later read \( e.f \) and call the reset method on the retrieved object, since we expect a reset counter object. We would then get stuck because the counter object does not have a reset method. Our current typing rules allow \( e.f \) expressions in sink positions to be typed covariantly, which violates type safety.

In the next section, we will repair our type system by fusing the TYPESUB rule with those typing rules where subtyping is actually needed to capture the substitution principle. In particular, we will restrict the use of subtyping in the TYPEASSIGNFLD rule so that the above program is no longer well-typed.

**Depth-Subtyping of Mutable Fields.** The reason for restricting the depth-subtyping rule to \texttt{const} fields is related to the issue that \texttt{var} fields can serve as sinks of values. If we allowed depth subtyping of \texttt{var} fields, we would introduce another loop hole in our type system. The following modified version of the problematic program above demonstrates this issue:

```javascript
const y = { f: { g: true });
const fun = function(x: { var f: {} }) { x.f = {};
}fun(y);
y.f.g
```

The type of \( y \) is \( \{ \text{var} f: \{ \text{var} g: \text{bool} \} \} \). Evidently, the function \( \text{fun} \) is well-typed since its parameter \( x \) has type \( \{ \text{var} f: {} \} \) and we are assigning \( x.f \) to \( {} \).

If we allow depth-subtyping of \texttt{var} fields, the call \( \text{fun}(y) \) is also well-typed since the type of \( y \) is then a subtype of \( \{ \text{var} f: {} \} \). Finally, the sequence of field dereferences \( y.f.g \) is again well-typed. Thus, if we relaxed the depth-subtyping rule, this program would be accepted by the type checker. However, during evaluation of the program, we will again get stuck when we try to evaluate the final dereference operation on field \( g \).

Although the problem with depth subtyping of objects is similar to the problem with the combination of the rules TYPEASSIGNFLD and TYPESUB that we discussed earlier, the two issues are orthogonal. Even if we restrict the usage of the TYPESUB in the TYPEASSIGNFLD rule, allowing depth subtyping of \texttt{var} fields will still be unsafe. For this reason we disallow depth-subtyping of \texttt{var} fields. Consequently, the above program will be rejected by the type checker because the call \( \text{fun}(y) \) is not well-typed.

If we want to use depth-subtyping for mutable field \( f \) of an object type, then we first have to demote the mutability of that field from \texttt{var} to \texttt{const} using the
rule \textsc{SubObjMut}. For example, consider the following program:

\begin{verbatim}
const y = { f: { g: true } };
const fun = function(x: { const f: {} } { x.f };
fun(y);
y.f.g
\end{verbatim}

This program can be safely evaluated. It is also well-typed because the following subtype relationship holds

\begin{verbatim}
{ var f: { var g: bool } } <: { const f: {} }
\end{verbatim}

This subtype relationship enables us to type the call to \texttt{fun}. This relationship can be derived using one application of the rule \textsc{SubObjMut} and one application of the rule \textsc{SubObjDepth}. These two applications can then be combined using the rule \textsc{SubTrans}.

On the other hand, consider the following program, which is a combination of the two programs above:

\begin{verbatim}
const y = { f: { g: true } };
const fun = function(x: { const f: {} } { x.f = {}; };
fun(y);
y.f.g
\end{verbatim}

Like the first program, the evaluation of this program will get stuck when we try to evaluate the final field dereference operation. The program is also not well-typed, however, for different reasons than the first program. Like the second program, the call to \texttt{fun} is now well-typed. Though, in the new program the body of the function \texttt{fun} is no longer well-typed because \texttt{fun} now tries to assign a new value to the \texttt{const} field of object \texttt{x}.

**Array Subtyping.** Many existing typed object-oriented languages have weaknesses in their type systems that stem from the incorrect handling of subtyping and assignments. Sometimes these issues are oversights in the language design but other times they are deliberate design decisions. For example, TypeScript deliberately allows covariant subtyping of location expressions in sink positions. While this may be desirable for a scripting languages that is not used for writing critical programs, it can be problematic for other programming languages. A prime example of this is the handling of array subtyping in Java.

In Java, arrays are typed covariantly, i.e., if a Java type \texttt{S} is a subtype of a type \texttt{T}, then \texttt{S[]} (the type of an array of elements of type \texttt{S}) is a subtype of \texttt{T[]}.

This treatment of arrays is unsafe because arrays can serve both as sources and sinks of values, as demonstrated by the following program:

\begin{verbatim}
1 String[] a = new String[1];
2 Object[] b = a; // a and b are now aliased
3 b[0] = new Object(); // a[0] now stores an Object
4 a[0].indexOf('c'); // indexOf not a method of Object
\end{verbatim}
This Java program is accepted by the Java type checker. However, the call to `indexOf` on line 4 would fail when the program is executed because instances of class `Object` do not have an `indexOf` method. To prevent this problem, Java checks at run-time whether the type of a value that is stored into an array is a subtype of the array’s element type. This check is performed before each array store operation is executed. The array store check fails on line 3, since the element type of the array referenced by `b` is that of `a`, which is `String`, and the type of the stored value, which is `Object`, is not a subtype of `String`. At this point, the run-time environment raises an `ArrayStoreException` and aborts the execution of the program.

The covariant treatment of arrays in Java was introduced as a convenience for programmers in the initial version of the language. The rationale for this decision was that Java 1.0 did not yet support generic types. Covariant arrays provided programmers with more flexibility to write polymorphic code, even without generics. However, in retrospect, this decision is considered a mistake in the language’s design. The overhead of the additional dynamic type checks for array store operations make the usage of arrays in Java quite expensive.

The correct treatment of arrays in a type system with subtyping is to consider them invariant in their element type. This means that, if `S` is a subtype of `T`, then it does not follow that `S[]` is also a subtype of `T[]` nor that `T[]` is a subtype of `S[]`. In this case, the program above is rejected by the type checker because the assignment of `a` to `b` on line 2 is ill-typed. Additional run-time checks for array store operation are then no longer needed. In Scala, arrays are typed invariantly.

### 8.3.4 Type Inference

We have learned how we can incorporate the substitution principle of object-oriented languages into our type system by extending it with a subtype relation. It remains to discuss how we can actually implement a type inference algorithm for the new typing relation. The problem with the current definition of the subtype relation and the subtyping rule is that these rules are not syntax-directed. Hence, when we try to implement the type inference algorithm by reading the inference rules bottom-up, we cannot decide how to apply the new rules just by looking at the syntactic structure of expressions. For example, reconsider the subtyping rule itself:

\[
\frac{\Gamma \vdash e : \tau' \quad \tau' \prec : \tau}{\Gamma \vdash e : \tau} \text{ TypeSub}
\]

Note that the rule applies to any expression `e` without restrictions on `e`’s syntactic structure. For each top-level syntactic constructor used in a program expression, our typing relation also has an ordinary rule that does not involve subtyping. Given an expression `e`, we could use this ordinary typing rule to infer the type `\tau'` of `e` that occurs in the premise of the rule `TypeSub` in an attempt to compose the two rules together. However, even if we did that, we
would still need to guess the supertype \( \tau \) of \( \tau' \) in the rule TYPE\( \text{SUB} \). How can we implement this rule without having to guess any of the types involved?

For the subtype relation itself the situation is similar. Consider the rule for transitivity of subtyping:

\[
\frac{\tau_1 \triangleleft \tau_2 \quad \tau_2 \triangleleft \tau_3}{\tau_1 \triangleleft \tau_3} \quad \text{SUB\( \text{TRANS} \)}
\]

In order to apply this rule during type inference (reading the rule bottom-up), we would have to guess the intermediate type \( \tau_2 \) to make the transitive step between \( \tau_1 \) and \( \tau_3 \). Again, it is unclear how to implement this rule without having to guess any types.

We will solve these problems by modifying the subtyping and typing rules so that they become again syntax-directed.

**Syntax-Directed Subtyping**

We start by redefining the subtype relation. Let us first analyze how the transitivity rule is used in a subtyping derivation. To this end, reconsider the example derivation from the previous section:

\[
\begin{align*}
\{\text{const } g: \text{bool}, \text{var } f: \text{bool}\} & \triangleleft \{\text{var } f: \text{bool}, \text{const } g: \text{bool}\} \quad \text{SUB\( \text{OBJ} \text{PERM} \)} \\
\{\text{var } f: \text{bool}, \text{var } g: \text{bool}\} & \triangleleft \{\text{var } f: \text{bool}\} \quad \text{SUB\( \text{OBJ} \text{WIDTH} \)} \\
\{\text{const } g: \text{bool}, \text{var } f: \text{bool}\} & \triangleleft \{\text{var } f: \text{bool}\} \quad \text{SUB\( \text{TRANS} \)}
\end{align*}
\]

We can see that we use the transitivity rule to combine several applications of the object subtyping rules—here, the rules SUB\( \text{OBJ} \text{PERM} \) and SUB\( \text{OBJ} \text{WIDTH} \). It turns out that these are the only situations where the transitivity rule is actually needed. Thus, we can eliminate the transitivity rule by merging it with the object subtyping rules SUB\( \text{OBJ} \text{PERM} \), SUB\( \text{OBJ} \text{WIDTH} \), SUB\( \text{OBJ} \text{DEPTH} \), and SUB\( \text{OBJ} \text{MUT} \) into a single rule for object subtyping. The resulting object subtyping rule, called SUB\( \text{OBJ} \), is shown in Figure 8.5.

Now, all of our subtyping rules are syntax-directed. By looking at the structure of the typing expression, we know which rule to apply. Note that the rule SUB\( \text{REFL} \) can be restricted to the cases where \( \tau \) is one of the basic types number and bool.

**Syntax-Directed Typing with Subtyping**

Our solution for making our typing relation syntax-directed is similar: we eliminate the subtyping rule TYPE\( \text{SUB} \) by merging it with those syntax-directed typing rules where subtyping is really needed. The modified typing rules are shown in Figure 8.6. For the syntactic primitives where no new rule is provided, the old typing rule still applies. We discuss the new typing rules in more detail.
\( \tau \llany \) SubAny \quad \tau \ll\tau \) SubRefl

\[
\begin{align*}
\tau_2 &\ll\tau' \\
\tau_1 &\ll\tau_1 \\
(\tau_1 \Rightarrow \tau_2) &\ll(\tau'_1 \Rightarrow \tau'_2) \quad \text{SubFun}
\end{align*}
\]

\[
\{g_1, \ldots, g_m\} \subseteq \{f_1, \ldots, f_n\}
\quad \text{for all } i, j, \text{ if } f_i = g_j, \text{ then } \text{mut}_j = \text{const} \text{ and } \tau_i \ll \tau'_j \\
or \quad \text{mut}_i = \text{mut'}_j \text{ and } \tau_i = \tau'_j
\quad \text{SubObj}
\]

Figure 8.5: Algorithmic subtyping rules for JAKARTA SCRIPT

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_2 &\Rightarrow \tau \\
\Gamma \vdash e_2 : \tau_2' &\ll\tau_2 \\
\Gamma \vdash e_1(e_2) : \tau &\quad \text{TypeCall}
\end{align*}
\]

\[
\begin{align*}
\Gamma' = \Gamma[x \mapsto (\text{const}, \tau_2)] &\quad \Gamma' \vdash e : \tau' \\
\Gamma' \vdash e : \tau &\ll\tau' &\quad \text{TypeFunAnn}
\end{align*}
\]

\[
\begin{align*}
\Gamma' = \Gamma[x_1 \mapsto \tau_1, x_2 \mapsto (\text{const}, \tau_2)] &\quad \Gamma' \vdash e : \tau' \\
\tau_1 &\ll(\tau_2 \Rightarrow \tau) &\quad \Gamma' \vdash e : \tau' \\
\Gamma \vdash \text{function } x_1(x_2:\tau_2) : \tau &\ll \tau &\quad \text{TypeFunRec}
\end{align*}
\]

\[
\begin{align*}
\Gamma(x) = (\var, \tau') &\quad \Gamma \vdash e : \tau \\
\Gamma \vdash x = e : \tau &\quad \text{TypeAssignVar}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 : \{ \ldots \var f : \tau' \ldots \} &\quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash e_1, f = e_2 : \tau &\quad \text{TypeAssignFld}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 : \text{bool} &\quad \Gamma \vdash e_2 : \tau_2 &\quad \Gamma \vdash e_3 : \tau_3 \\
\tau_2 \sqcup \tau_3 &\ll \tau &\quad \text{TypeIf}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 &\quad \Gamma \vdash e_2 : \tau_2 \\
\tau_1 \sqcup \tau_2 &\not\ll \text{any} \\
\tau_1, \tau_2 &\text{ have no function types} &\quad bop \in \{==, !=\} \\
\Gamma \vdash e_1 \ bop \ e_2 : \text{bool} &\quad \text{TypeEqual}
\end{align*}
\]

Figure 8.6: Typing rules with subtyping
**Function Calls.** The rule `TypeCall` is the new rule for typing function call expressions. Observe that the new rule no longer requires that the argument type and parameter type of the called function are equal. Instead, the rule now incorporates the subtyping rule directly by requiring that the argument type can be a subtype of the parameter type.

**Type Annotations.** In the cases for function expressions with type annotations, we have to make a design decision. We can either keep our old typing rules, which require the programmer to annotate functions with the exact return type of the function body, or we can relax the condition and only require that the annotated type is a supertype of the actual type of the function body. We choose the latter because it provides a useful mechanism for information hiding. For instance, consider the following typed variant of our counter class with open recursion from Section 8.2.4:

```javascript
const counterClass =
  function(this: { var x: number,
    var get: () => number,
    var inc: () => () })
  : { var get: () => number, var inc: () => () }
  {
    this.x = 0;
    this.get = function() { return this.x; };
    this.inc = function() { this.x = this.x + 1; };
    return this;
  };
```

The annotated return type for the function `counterClass` is a supertype of the actual type of the returned expression `this`. The effect is that the return type hides the field `x` of `this`, which we use for the representation of the internal state of the counter object. A client that uses an instance of the counter class can now only access the field `x` by calling the methods `get` and `inc`. The type annotation hides the field `x`, i.e., the internal representation of the object, from the client.

**Assignment Expressions.** The rules `TypeAssignVar` and `TypeAssignFld` are the new rules for typing assignment expressions. Note that the rules require that the inferred type `τ'` of the assigned location expression is a supertype of the type `τ` inferred for the right side of the assignment. Thus, location expressions are typed contravariantly. These modified rules plug the loop hole in our previous type system with the unrestricted subtyping rule `TypeSub`. In particular, our problematic program from Section 8.3.3 is now rejected by the new typing relation:

```javascript
const x = {f: {g: true}};
x.f = {};
x.f.g
```
The assignment expression on line 2 is ill-typed because the type `{}` is not a subtype of the type `{var g: bool}`.

**Conditional Expressions.** Perhaps the most interesting new typing rule is the rule `TypeIf` for conditional expressions $e_1 ? e_2 : e_3$. In our earlier version of this rule, we required that the inferred types of the “then” branch $e_2$ and the “else” branch $e_3$ must agree. With the subtyping rule we can relax the inferred types of the two branches to a common supertype. For example, with subtyping the following program is well-typed:

```plaintext
const o = true ? {x: 0, y: true} : {x: 2, z: 0};
2 * o.x
```

The inferred type of the “then” branch `{x:0, y: true}` is

{var x: number, var y: bool}

and the inferred type of the “else” branch `{x: 2, z: 0}` is

{var x: number, var z: number}

While the two types do not agree, they have a common supertype

{var x: number}

Thus, we can apply the rule `TypeSub` to each branch, relaxing the inferred type to the common supertype. We can then successfully type the conditional expression with the old `TypeIf` rule. Moreover, the common supertype is sufficient to successfully type the body of the `const` declaration, since the expression $2 * o.x$ only accesses the field x of o, which it expects to be of type `number`. Hence, the above program is well-typed.

How do we incorporate the rule `TypeSub` directly into the rule `TypeIf`? We want to be able to relax the inferred types of the two branches to a common supertype, but which supertype should we pick? For example, in the above program another common supertype is the empty object type `{}`. This supertype could also be used to merge the two branches in the conditional. Hence, the conditional expression would still be well-typed. However, we would then fail to type the subexpression $o.x$ in the body of the `const` declaration if the inferred type for o was the empty object type `{}`. When we merge the inferred types of the two branches, we have to preserve as much information as possible that is common to the two types. More precisely, the type of a conditional expression should be the *least common supertype* of the types inferred for its two branches.

We refer to the least common supertype of two types $\tau_1$ and $\tau_2$ as the *join* of the two types, which we denote by $\tau_1 \sqcup \tau_2$. The intuition is that we find $\tau_1 \sqcup \tau_2$ by walking up the lattice of the subtyping relation from both $\tau_1$ and $\tau_2$ until we find the first common supertype. For example, we have

{var x: number, var y: bool} $\sqcup$ {var x: number, var z: number} = {var x: number}

Hence, the type that the new `TypeIf` rule infers for the expression
true ? {x: 0, y: true} : {x: 2, z: 0}

is {var x:number}. You may ask yourself why the inferred type is not

{var x:number, var y:bool}

since we know that the above conditional expression will always take the “then” branch. The reason is that the typing rules must consider the general case. In general, we cannot predict the truth value of branching conditions statically without evaluating the program. The typing relation therefore always assumes that both branches could be taken and infers the type that captures the maximal information that is common to the types of both branches. This type is obtained by taking the join of the types that are inferred for the two branches. We discuss the computation of joins in detail below.

In our subtype relation, joins always exist because every two types \( \tau_1 \) and \( \tau_2 \) have at least one common supertype, the type any. For example, the following expression will have type any:

true ? 3 : false

If our type inference algorithm infers the type any for an expression, then this means that the rest of the program cannot do much with the result value of such expressions, except passing it to functions whose parameters have type any.

Equality Expressions. Finally, the TypeEqual rule is a slightly relaxed version of our earlier typing rule for equality and disequality expressions. Instead of requiring that we only compare expressions \( e_1 \) and \( e_2 \) of the same type, we now require that the inferred types \( \tau_1 \) and \( \tau_2 \) of the two expressions must have some common supertype that is different from any. We express this condition by stating that \( \tau_1 \sqcup \tau_2 \neq \text{any} \). This relaxation allows equality comparisons between objects of different object types while disallowing comparisons of values between different primitive types or function types.

Computing Joins and Meets

Joins. A join of two types \( \tau \) and \( \tau' \) is a type \( \tau'' \) that satisfies the following two properties: (1) \( \tau'' \) is a supertype of both \( \tau \) and \( \tau' \) and (2) any other supertype \( \tau_s \) of \( \tau \) and \( \tau' \) must also be a supertype of \( \tau'' \).

In our type system, joins are not unique. That is, given \( \tau \) and \( \tau' \), we can have two distinct types \( \tau_1 \) and \( \tau_2 \) such that both satisfy the conditions (1) and (2) above. For example, consider the types:

\[
\tau = \{x:\text{number}, y:\text{bool}; z: \text{bool}\}
\]

\[
\tau' = \{x:\text{number}, y: \text{number}; z: \text{bool}\}
\]

Then both of the following types are joins of \( \tau_1 \) and \( \tau_2 \):

\[
\tau_1 = \{x: \text{number}, z: \text{bool}\}
\]

\[
\tau_2 = \{z: \text{bool}, x: \text{number}\}.
\]
Note that we have $\tau_1 \triangleleft \tau_2$ and $\tau_2 \triangleleft \tau_1$. In fact, $\tau_1$ and $\tau_2$ are equal up to permutation of their fields. This is true in general: any two joins of two types $\tau$ and $\tau'$ will be equal up to permutation of fields in their object type subexpressions. In our discussion, we will treat these different joins as equal and simply write $\tau \sqcup \tau'$ for the join of $\tau$ and $\tau'$.

The inference rules in Figure 8.7 describe how the join function $\sqcup$ can be computed using recursion on the structure of type expressions. We discuss the individual rules in more detail below.

**Joins of Basic Types.** If one of the two types $\tau$ and $\tau'$ is a basic type, then computing their join $\tau \sqcup \tau'$ simply reduces to checking whether the two types are equal or not. This is because basic types only have any as a proper supertype. The corresponding case analysis is captured by the rules JoinBasic, JoinAny$_1$, and JoinAny$_2$.

**Joins of Object Types.** Next, we discuss the case where $\tau$ and $\tau'$ are both object types. The join computation for object types is handled by the JoinObj rules. First, note that the join of two object types cannot be any because all object types have the empty object type ${\{}$ as a common supertype. Intuitively, we obtain the actual join $\tau''$ by taking all fields $h$ that are common to both $\tau$ and $\tau'$, and merging the types of $h$ appropriately. The join computation thus works by recursion over the fields of $\tau$. The base case is captured by the rule JoinObjEmp. The remaining rules capture the cases where $\tau$ has at least one field $h$. The recursion then goes over the left-most field $h$ of $\tau$ using further case analysis. Let us first consider the case where $h$ only occurs in $\tau$ but not $\tau'$. This case is captured by the rule JoinObjNo. In this case, $h$ will not be included in the resulting join $\tau''$ and we simply obtain $\tau''$ by recursively computing the join for the remaining fields of $\tau$ with $\tau'$. If $h$ occurs in both $\tau$ and $\tau'$, we have to make another case distinction. In the first subcase, we consider the situation where $h$ is a var field in both $\tau$ and $\tau'$, and the types of $h$ are identical. Then we can simply add $h$ as a var field of that type to the resulting join $\tau''$. This case is captured by the rule JoinObjVar$_{\text{eq}}$. In the remaining cases, $h$ must either have a different type or a different mutability in $\tau$ and $\tau'$. These two cases are captured by the rule JoinObjMUT$_{\neq}$. To obtain any common object supertype of $\tau$ and $\tau'$ that still contains field $h$, we must use depth-subtyping for the field $h$, change the mutability of $h$ from var to const, or both. Either way, the join $\tau''$ of $\tau$ and $\tau'$ will have a const field $h$ whose type is the join of the types of $h$ in $\tau$ and $\tau'$.

**Joins of Function Types.** Finally, we consider the case of computing the join of two function types $\tau_1 \Rightarrow \tau_2$ and $\tau'_1 \Rightarrow \tau'_2$. The join of two function types

---

2More formally, we consider the quotient of Typ under the equivalence relation that is induced by the subtype relation. Then $\tau \sqcup \tau'$ is the least upper bound of the equivalence classes represented by $\tau$ and $\tau'$ in the partial order that is induced by $\triangleleft$ on this quotient. The least upper bounds of a partial order are always unique if they exist.
\[ \tau \in \{ \text{bool, number, any} \} \quad \text{JOINBasic}_{=} \]
\[ \tau \neq \tau' \quad \tau \in \{ \text{bool, number, any} \} \quad \text{JOINBasic} \]
\[ \tau \neq \tau' \quad \tau' \in \{ \text{bool, number, any} \} \quad \text{JOINBasic}_2 \]
\[ \{ \} \sqcup \{ \text{mut} \ g : \tau_g \} = \{ \} \quad \text{JOINObjEmp} \]
\[ \{ \text{mut} \ h : \tau_h, \text{mut} \ f : \tau_f \} \sqcup \{ \text{mut} \ g : \tau_g, \text{mut} \ g' : \tau_{g'} \} = \{ \text{mut} \ k : \tau_k \} \quad \text{JOINObjMut} \]
\[ \{ \text{var} \ h : \tau_h, \text{mut} \ f : \tau_f \} \sqcup \{ \text{mut} \ g : \tau_g, \text{var} \ h : \tau_h, \text{mut} \ g' : \tau_{g'} \} = \tau'' \quad \text{JOINObjVar}_{=} \]
\[ \{ \text{mut} \ f : \tau_f \} \sqcup \{ \text{mut} \ g : \tau_g, \text{mut} \ g' : \tau_{g'} \} = \{ \text{mut} \ k : \tau_k \} \quad \text{JOINObjNo} \]
\[ h \notin \{ \} \quad \{ \text{mut} \ f : \tau_f \} \sqcup \{ \text{mut} \ g : \tau_g \} = \tau'' \quad \text{JOINObjNo} \]
\[ \tau_1 \sqcap \tau'_1 = \tau''_1 \quad \tau_2 \sqcup \tau'_2 = \tau''_2 \quad \tau''_1 \Rightarrow \tau''_2 \quad \text{JOINFunMeet} \]
\[ \tau_1 \sqcap \tau'_1 = \bot \quad (\tau_1 \Rightarrow \tau_2) \sqcup (\tau'_1 \Rightarrow \tau'_2) = \text{any} \quad \text{JOINFunMeet} \]
\[ \{ \text{mut} \ f : \tau_f \} \sqcup (\tau_1 \Rightarrow \tau_2) = \text{any} \quad \text{JOINFunObj} \]
\[ (\tau_1 \Rightarrow \tau_2) \sqcup \{ \text{mut} \ f : \tau_f \} = \text{any} \quad \text{JOINFunObj} \]

Figure 8.7: Rules for computing joins.
Subtyping 181

is either any or again a function type of the form $\tau''_1 \Rightarrow \tau''_2$. The two cases are captured by the rules JOINFUNMEET and JOINFUNANY.

We first discuss the rule JOINFUNMEET where the computed join is of the form $\tau''_1 \Rightarrow \tau''_2$. How do we compute the types $\tau''_1$ and $\tau''_2$ in this case? Since function types are covariant in their result types, it is easy to see that we must have $\tau''_2 = \tau_2 \cup \tau'_2$. The following program exemplifies this intuition:

```javascript
const f1 = function () { return { x: 0, y: true } }; const f2 = function () { return { x: 1, z: 1 } }; const f = true ? f1 : f2; f().x
```

The inferred type for $f$ must be such that all subsequent usages of $f$ are safe, regardless of whether $f$ is evaluated to $f1$ or $f2$. The expression $f().x$ accesses the field $x$ of $f$'s return value. This is safe since the objects returned by $f1$ and $f2$ both have a field $x$ of type number. However, if we replaced $f().x$ by $f().z$, then the program would only be safe if the conditional expression defining $f$ always took the “else” branch. The inferred return type for $f$ should capture the maximal information that is common to the return types of $f1$ and $f2$, i.e., we have to compute the join of the two return types.

For the parameter types of two joined function types the situation is reversed since parameter types are contravariant. Consider the following example:

```javascript
const f1 = function (o: { var x: number, var y: number }) { return o.x + o.y }; const f2 = function (o: { var x: number, var z: number }) { return o.x + o.z }; const f = true ? f1 : f2; f({x: 1, y: 2, z: 3})
```

The inferred type of $f1$ is

$$o: \{ \text{var } x:\text{number}, \text{var } y:\text{number} \} \Rightarrow \text{number}$$

and the inferred type of $f2$ is

$$o: \{ \text{var } x:\text{number}, \text{var } z:\text{number} \} \Rightarrow \text{number}$$

The call to $f$ must be safe regardless of whether $f$ is bound to $f1$ or $f2$. Note that $f1$ accesses fields $x$ and $y$ of the passed object whereas $f2$ accesses fields $x$ and $z$. Hence, the object that is passed to $f$ must at least have the fields $x$, $y$, and $z$. Computing the join of the parameter types of $f1$ and $f2$ yields

$$\{ \text{var } x:\text{number} \}$$

Thus, if we defined the parameter type of the join of the types of $f1$ and $f2$ to be $\{ \text{var } x:\text{number} \}$, then the call $f({x: 1})$ would be well-typed, even though it is unsafe.
Instead of computing the type that captures the maximal information that is common to both parameter types, we have to compute the type that captures all the information provided in each of the two types. This means that we have to compute their greatest common subtype. In our example above, the greatest common subtype of the types `{var x: number, var y: number}` and `{var x: number, var z: number}` is the type

```plaintext
{var x: number, var y: number, var z: number}
```

Thus the inferred type for f is:

```plaintext
{var x: number, var y: number, var z: number} => number
```

We refer to the greatest common subtype of two types \( \tau \) and \( \tau' \) as their meet, which we denote by \( \tau \sqcap \tau' \).

Note that unlike joins, the meet of two types does not always exist. For example, the types `number` and `bool` have no common subtype and thus no meet. We write \( \tau \sqcap \tau' = \bot \) to indicate that the meet of \( \tau \) and \( \tau' \) does not exist. If the meet of the parameter types of two function types does not exist, then their only common supertype is `any`. This case is captured by the rule `JoinFunAny`.

Finally, the join of an object type and a function type is always `any`. This case is covered by the rules `JoinObjFun` and `JoinFunObj`.

Due to the contravariant subtyping of parameter types in function types, we have to compute joins and meets simultaneously using mutual recursion. Whenever we compute the join of two function types, we switch to computing the meets of their parameter types. Similarly, whenever we compute the meet of two function types, we switch back to computing the joins of their parameter types. This mutual recursion is well-founded (i.e., the recursion will always terminate) since we recurse into smaller subexpressions of the considered types. We next explain the computation of meets in detail.

**Meets.** As indicated in our discussion above, the meet of two types is the order-theoretic dual of their join. Thus, a meet \( \tau'' \) of two types \( \tau \) and \( \tau' \) is a type that satisfies the following two conditions: (1) \( \tau'' \) is a common subtype of \( \tau \) and \( \tau' \); and (2) any other common subtype \( \tau_s \) of \( \tau \) and \( \tau' \) is also a subtype of \( \tau'' \). Similar to joins, meets are unique up to reordering of fields in object type expressions. So we denote by \( \tau \sqcap \tau' \) one of the representatives of all these equivalent meets if any such meet exists at all. Figure 8.8 provides inference rules for computing meets using structural recursion on the type expressions. The rules are very similar to the rules for joins. In particular, the rule for computing meets of function types, `MeetFunJoin`, is the dual of the rule `JoinFunMeet`. That is, we obtain `MeetFunJoin` from `JoinFunMeet` by replacing meets by joins and vice versa. Note that there is no equivalent for the `JoinFunAny` rule. If the meet of the result types of two function types does not exist, then the meet of these function types also does not exist. In general, there are no explicit

\footnote{That is, meets are the greatest lower bounds of the partial order that is induced by the subtype relation.}
\[\tau \in \{\text{bool, number, any}\} \implies \tau \cap \tau = \tau\]

\[\text{MeetBasic} = \]

\[\text{any} \cap \tau = \tau \quad \text{MeetAny}_1 \quad \tau \cap \text{any} = \tau \quad \text{MeetAny}_2\]

\[\varnothing \cap \{\text{mut}_g \ g: \tau_g\} = \{\text{mut}_g \ g: \tau_g\} \quad \text{MeetObjEmp}\]

\[\{\text{mut}_f \ f: \tau_f\} \cap \{\text{mut}_g \ g: \tau_g, \text{mut}_{g'} \ g': \tau_{g'}\} = \{\text{mut}_k \ k: \tau_k\} \quad \tau_h \cap \tau_{h'} = \tau_h'' \quad \tau = \{\text{const} \ h: \tau_h, \text{mut}_k \ k: \tau_k\}\]

\[\{\text{const} \ h: \tau_h, \text{mut}_f \ f: \tau_f\} \cap \{\text{mut}_g \ g: \tau_g, \text{const} \ h': \tau_{h'}, \text{mut}_{g'} \ g': \tau_{g'}\} = \tau\]

\[\{\text{mut}_f \ f: \tau_f\} \cap \{\text{mut}_g \ g: \tau_g, \text{mut}_{g'} \ g': \tau_{g'}\} = \{\text{mut}_k \ k: \tau_k\} \quad \tau = \{\text{var} \ h: \tau_h, \text{mut}_k \ k: \tau_k\}\]

\[\{\text{mut}_h \ h: \tau_h, \text{mut}_f \ f: \tau_f\} \cap \{\text{mut}_g \ g: \tau_g, \text{mut}_h \ h: \tau_{h'}, \text{mut}_{g'} \ g': \tau_{g'}\} = \tau\]

\[\{\text{mut}_f \ f: \tau_f\} \cap \{\text{mut}_g \ g: \tau_g\} = \{\text{mut}_k \ k: \tau_k\} \quad h \notin \{g\} \quad \tau'' = \{\text{mut}_h \ h: \tau_h, \text{mut}_k \ k: \tau_k\}\]

\[\{\text{mut}_h \ h: \tau_h, \text{mut}_f \ f: \tau_f\} \cap \{\text{mut}_g \ g: \tau_g\} = \tau''\]

\[\tau_1 \cup \tau_1' = \tau_1'' \quad \tau_2 \cap \tau_2' = \tau_2'' \quad \tau'' = \tau_1' \Rightarrow \tau_2'' \quad (\tau_1 \Rightarrow \tau_2) \cap (\tau_1' \Rightarrow \tau_2') = \tau''\]

\[\text{MeetFunJoin}\]

Figure 8.8: Rules for computing meets.

rules for the cases where the meet of two types \(\tau\) and \(\tau'\) does not exist. Our convention is that if there is no type \(\tau''\) such that we can derive \(\tau \cap \tau' = \tau''\) with the rules in Figure 8.8, then we define \(\tau \cap \tau' = \perp\).

In the following, we discuss the rules for computing meets of object types in more detail.

**Meets of Object Types.** In order to compute the meet \(\tau'' = \tau \cap \tau'\) of two object types \(\tau\) and \(\tau'\), we proceed as follows. Again, the computation of the meet recurses over the left-most field of \(\tau\). The individual cases are captured by the MeetObj rules. First, if a field \(h\) of \(\tau\) does not occur in \(\tau'\), then it is simply added to the resulting meet \(\tau''\), and vice versa for the fields that occur in \(\tau'\) but not in \(\tau\). These cases are covered by the rules MeetObjNO and MeetObjEMP. The remaining rules consider the cases where the left-most field \(h\) of \(\tau\) also occurs in \(\tau'\). In this case, we first check whether \(h\) has mutability var in both types. If yes, then the types \(\tau_h\) and \(\tau_h'\) of \(h\) in \(\tau\) and \(\tau'\) must be equal to obtain any common subtype (rule MeetObjVar=). If they are not equal, then the meet \(\tau''\) does not exist. If at least one of the two mutabilities of
$h$ is `const`, then we can still use depth-subtyping to obtain a common subtype if $\tau_h$ and $\tau'_h$ are not equal. In this case, the type of $h$ in $\tau''$ is computed recursively by taking the meet of the types $\tau_h$ and $\tau'_h$. If this meet does not exist, then the meet $\tau''$ again does not exist. If the meet of $\tau_h$ and $\tau'_h$ exists, then the mutability of $h$ in $\tau''$ is `const` if $h$ has `const` mutability in both $\tau$ and $\tau'$ (rule MEETOBJCONST), and `var` otherwise (rule MEETMUT≠).