Principles of Programming Languages
Course Notes

Thomas Wies

December 3, 2023

Preface

This document contains the lecture notes for the NYU undergraduate course CSCI-UA.0480-055, “Principles of Programming Languages”, in fall 2023. 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
- 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

List of Figures

Chapter 1

Scala Basics

1.1 Getting Started

In the following, we assume that you have installed sbt and Intellij Idea with the Scala plugin. If you have not yet done so, please do it now. You can find a link to the installation instructions on the course web site on Brightscape.

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:

```
package greeter

object Hello extends App:
  def main(args: Array[String])
    println("Hello␣World!")
```

This code creates an object `Hello` in the package `greeter`. The object `Hello` contains a method called `main`, which means that `Hello` can serve as the entry point of a Scala application that calls `main` upon start, providing the command line arguments via the parameter `args`. When this application is started, the object `main` prints the message `"Hello␣World!"` on standard output.

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

```
package greeter;

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

You can safe 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 Intelij Idea IDE, you can import the `in-class-code` project following the instructions provided in the `README.md` file of the repository. The repository contains a file `ScalaGreeter.scala`, which you can find in the package `popl.class03`. To compile and run the application, right-click on the file and select “Run 'ScalaGreeter' ”. This should print `"Hello Scala"` in the output view at the bottom of the window.

1.1.2 The Scala REPL and Worksheets in the 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 execute. To make life easier, Scala provides a useful tool called a read-eval-print loop, or *REPL* for short. The Scala REPL is essentially a command line calculator on steroids. 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 execute `scala`. This will start the REPL and a message similar to the following should appear:

```
Welcome to Scala 3.3.0 (18.0.2-ea, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala>
```

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

```
scala> 3 + 4
val res0: Int = 7
```

You can exit the REPL by typing `:quit` or by pressing `Ctrl-d` (respectively, `Cmd-d` on OS X).

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

1. Start Intelij Idea. (In the following, I assume you have previously imported the `in-class-code` project with the `popl` package.)

2. Right-click the `popl` package in the package explorer and choose New/Scala Worksheet. Name the worksheet “MyWorksheet” and press Finish.

3. An empty Scala source file called `MyWorksheet.sc` will open in the editor window.
4. You can now type Scala expressions in the editor window. Each time you click the "Evaluate Worksheet" button at the top of the editor view (the button with the green arrow icon), the expressions in the file are evaluated and the result of the evaluation appear in a separate view next to the editor.
5. If you click the 'Show worksheet settings' button (the button with the wrench icon), you can set the worksheet to Interactive Mode, which will cause all expressions to be evaluated automatically as you type.

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 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:

```
val res0: Int = 7
```

This line includes:

- the keyword **`val`**, indicating that you have defined a new value resulting from evaluating the expression.
- an automatically generated name `res0` that is *bound* to that new value,
- a colon `:`, followed by the type `Int` of the expression,
- 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> res0 * res0
val res1: Int = 9
```

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

```
scala> if res1 > 10 then res0 - 5 else res0 + 5
val 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 is an expression that 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> if res1 > 2 then println(”Large!”)
       else println(”Not␣so␣large!”)
Large!
```

Note that the result value is not automatically bound to a name in this case. The if-then-else expression still 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). 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> val x = 3
val x: Int = 3
scala> x * x
val 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> val x: Int = 3
val x: Int = 3
```

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

```
-- [E052] Type Error: -------------------------------------------------------
1 |x = 5
  |^^^^^
  |Reassignment to val x
```

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

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

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:

```
-- [E007] Type Mismatch Error: ----------------------------------------------------
1 |y = "Hello"
  |    ^^^^^^^
  |    Found:    ("Hello" : String)
  |    Required: Int
```

For the time being, we will pretend that variables do not exist. Repeat after me: **val**s are gooood! **var**s are baaaad!

1.2.3 Functions

Here is how you write functions in Scala:

```
scala> def max(x: Int, y: Int): Int =
         if x > y then x else y
def 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.

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

```
scala> max(6, 3)
val 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
val pi: Double = 3.14159

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

def area(Double): Double
```

Note that the scope of the body of a function definition is determined automatically by the indentation level. For instance, the following code does not compile because the last line is no longer interpreted to be part of the body of the function area:

```
scala> def area(r: Double) =
         val pi = 3.14159
         def square(x: Double) = x * x
       pi * square(r)
-- [E006] Not Found Error: ------------------------------------------------
4 |pi * square(r)
  |^^
  |Not found: pi
```

If you have a longer function definition, it can be helpful to mark the end of the function explicitly using an **end** marker, followed by the name of the function:

```
def area(r: Double) =
  val pi = 3.14159
  def square(x: Double) = x * x
  pi * square(r)
end area
```

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 then 1 else n*fac(n-1)
def fac(n: Int): Int

scala> fac(5)
val res4: Int = 120
```

1.2.4 Scopes

You can use curly braces `{ ... }` to create block 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 are two difference to Java, though. First, the scope of a name extends the entire block in which it is defined. Using a name before its definition leads to an error:

```
val a = 3
{
  val b = a // Refers to 'a' defined on the next line.
  val a = 4 // Does not compile.
}
```

However, unlike Java, 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
}
```

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

and here is a tuple holding three items: two `String`s and a `Double` value:

```
scala> val q = ("apple", "pear", 1.0)
val q: (String, String, Double) = (apple, pear, 1.0)
```

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
val res5: Int = 1

scala> p._2
val res6: String = banana
```

Additionally, you can assign each element of the tuple to its own **`val`**:

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

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

```
def f(x: Int, s: String) = x

f(p._1, p._2) // Works.
f(p) // Does not compile.

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

g(p) // Works.
g((1, "banana")) // Works.
g(1, "banana") // Works.
```

1.3 Recursion

Recursion will be our main device 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.

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

How are calls to such functions evaluated? Conceptually, 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 (i.e., the occurrences of a and b in the example) by the actual arguments of the call. In our example, this step yields the expression

  ```
  if 2 < 2 then 2 + sum(2 + 1, 2) 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, this process eventually terminates, producing the result value `0`.

Here is how we compute the value of `sum(1, 4)` using rewriting:

```
sum(1, 4)
-> if 1 < 4 then 1 + sum(1 + 1, 4) else 0
-> if true then 1 + sum(1 + 1, 4) else 0
-> 1 + sum(1 + 1, 4)
-> 1 + sum(2, 4)
-> 1 + (if 2 < 4 then 2 + sum(2 + 1, 4) else 0)
-> 1 + (if true then 2 + sum(2 + 1, 4) else 0)
-> 1 + (2 + sum(2 + 1, 4))
-> 1 + (2 + sum(3, 4))
-> 1 + (2 + (if 3 < 4 then 3 + sum(3 + 1, 4) else 0))
-> 1 + (2 + (if true then 3 + sum(3 + 1, 4) else 0))
-> 1 + (2 + (3 + sum(3 + 1, 4)))
-> 1 + (2 + (3 + sum(4, 4)))
-> 1 + (2 + (3 + (if 4 < 4 then 4 + sum(4 + 1, 4) else 0)))
-> 1 + (2 + (3 + (if false then 4 + sum(4 + 1, 4) else 0)))
-> 1 + (2 + (3 + 0))
```

```
-> 1 + (2 + 3)
-> 1 + 5
-> 6
```

We refer to this sequence of rewriting steps as an *execution trace.*

Termination. Does the rewriting process always terminate and produce a finite execution trace? Consider the following recursive function:

```
def loop(x: Int): Int = loop(x)
```

If we evaluate, e.g., the call `loop(0)`, we obtain an infinite rewriting sequence:

```
loop(0) -> loop(0) -> loop(0) -> ...
```

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 `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 `sum` and the recursion stops. For our non-terminating function `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> sum(1, 1000000)
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 execution trace 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.

During execution of a Scala expression, the arguments of functions that have been called, but have not yet returned, are maintained on the *call stack*. 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:

```
def sumImp(a: Int, b: Int): Int =
  var acc = 0
  var i = a
  while i < b do
    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 results are stored in the variables i and acc, which are reused in each iteration. Unfortunately, this implementation uses mutable variables. Mutable variables make 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 acc to function parameters:

```
def loop(acc: Int, i: Int, b: Int): Int =
  if i < b then loop(i + acc, i + 1, b) else acc

def sumTail(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 acc to the recursive call of loop.

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 the execution trace:

```
sumTail(1, 4)
   -> loop(0, 1, 4)
   -> if 1 < 4 then loop(1 + 0, 1 + 1, 4) else 0
   -> if true then loop(1 + 0, 1 + 1, 4) else 0
   -> loop(1, 2, 4)
   -> if 2 < 4 then loop(2 + 1, 2 + 1, 4) else 1
   -> if true then loop(2 + 1, 2 + 1, 4) else 1
   -> loop(3, 3, 4)
   -> if 3 < 4 then loop(3 + 3, 3 + 1, 4) else 3
   -> if true then loop(3 + 3, 3 + 1, 4) else 3
   -> loop(6, 3, 4)
   -> if 4 < 4 then loop(4 + 6, 4 + 1, 4) else 6
   -> if false then loop(4 + 6, 4 + 1, 4) else 6
   -> 6
```

Observe that the size of the expressions that we obtain throughout the trace 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.

To simplify our implementation, we can move the declaration of the function `loop` inside the body of the function `sumTail`:

```
def sumTail(a: Int, b: Int): Int =
  def loop(acc: Int, i: Int): Int =
    if i < b then loop(i + acc, i + 1) else acc
  loop(0, a)
```

Note that in this version, we have dropped the third parameter `b` of the first version of the function `loop` since it is just passed to the recursive call without change. The occurrence of `b` in the body of the new nested version of `loop` now always refers to the parameter `b` of the outer function `sumTail`.

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. This optimization is referred to as *tail call elimination*. Many modern compilers, 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> sumTail(1, 1000000)
val res0: Int = 704982704
```

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. If you are unsure about how to write a tail-recursive function, it is often helpful 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.

If you are in doubt whether a recursive function that you wrote is tail-recursive, you can add the @tailrec annotation to the declaration of the function:

```
import scala.annotation.tailrec
...
def sumTail(a: Int, b: Int): Int =
  @tailrec def loop(acc: Int, i: Int): Int =
    if i < b then loop(i + acc, i + 1) else acc
  loop(0, a)
```

If the compiler fails to apply tail call elimination to a @tailrec annotated function, then it will issue a warning:

```
-- Error: ----------------------------------------------------
2 |  if a < b then a + sum(a + 1, b) else 0
  |                    ^^^^^^^^^^^^
  |   Cannot rewrite recursive call: it is not in tail position
```

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 than a recursive function that performs the same computation, but that is not tail-recursive. [1]. If you know that the recursion depth of the calls to your function will be small in practice, you may want to write the function without tail-recursion. In general, if you are in doubt, you should always 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

[1] Similarly, recursive functions are easier to understand than computations that use imperative loops.

object:

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

  public Pair(int fst, int snd) {
    first = fst;
    second = snd;
  }

  public int getFirst() {
    return first;
  }

  public int getSecond() {
    return second;
  }
}
```

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.).

Let us ignore for the moment that we can represent pairs directly in Scala using a tuple type. Here is how we can define the corresponding class in 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 **val**s, 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> val p = new Pair(1,2)
val p: Pair = Pair@1458e1cc
scala> p.first
val 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:

```
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 **val**s into **var**s, make them private, and add getter and setter methods. However, we want to avoid using **var** declarations as much as possible. The idiomatic solution in Scala is to make a copy of the entire object and change the appropriate value:

```
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:

```
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:

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

1.4.3 Singleton and Companion Objects

If the construction of an object involves complex initialization code, it is often useful to declare dedicate *factory* methods that perform this initialization. In Java, we would declare such methods as *static* members of the corresponding class:

```
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 that “everything is an object”. Instead of static methods, 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**, which is automatically created from the **object** declaration when the program is started. Since no further instances of the object can be created, object declarations do not have parameter lists.

For every class C in a Scala program, one can declare a singleton object that is also called C. This object is referred to as 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:

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

1.4.4 The apply Method

Methods with the name apply are treated specially by the Scala compiler. For example, if we rename the factory method make in our companion object for the Pair class to apply

```
object Pair:
  def apply(fst: Int, snd: Int) = new Pair(fst, snd)
```

then we can call this method simply by referring to the Pair companion object, followed by the argument list of the call:

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

This is equivalent to the following explicit call to the apply method:

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

The compiler automatically expands Pair(3,4) to Pair.apply(3, 4). That is, objects with an apply method can be used as if they were functions[2]. This feature is particularly useful to enable concise calls to factory methods. In fact, factory methods for the data structures in the Scala standard library are typically implemented using apply methods provided by companion objects.

[2] If you are familiar with C++, then you will notice that this feature is similar to overloading the function call operator () in C++.

1.5 Algebraic Data Types

Algebraic data types (ADTs) 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.

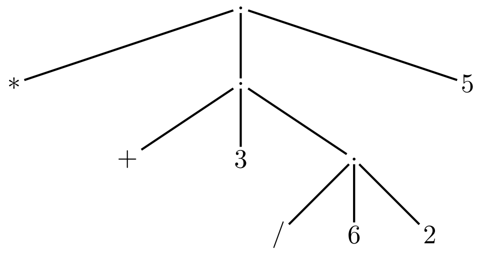
1.5.1 Enumerations

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 here (i.e., arithmetic expressions) is much simpler than a full-blown programming language.

One of the first questions 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 $(3 + 6/2) * 5$ can be visualized as follows:



For the AST node representing an expression $e_1 \, op \, e_2$, we follow the convention of representing the operator *op* as the left-most child of the node, rather than putting it between the children representing the operands e_1 and e_2.

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 second subtree to evaluate the left operand of the operation. Then we do the same for the third subtree, which represents the right operand of the operation. Finally, we combine the results obtained from the two operands according to the operator labeling the first child. We will learn more about ASTs later. 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 *enumerations* or “enums” for short. The following enum defines the ASTs of our arithmetic expressions:

```
enum Expr:
  /* Numbers such as 1, 2, etc. */
  case Num(num: Int)
  /* Expressions composed using binary operators */
  case BinOp(op: Bop, left: Expr, right: Expr)

/* Binary operators */
enum Bop:
  case Add /* + */
  case Sub /* - */
  case Mul /* * */
  case Div /* / */
```

This code declares an enum `Expr` whose instances represent the ASTs of our expression language. The code distinguishes two types of `Expr` objects based on the root node of the represented AST. A `Num(num)` object represents an AST consisting of a single node storing the integer constant `num`. Similarly, an object `BinOp(op, left, right)` represents an AST whose root node combines two subexpressions represented by ASTs `left` and `right` using the *binary operator* `op`. Binary operators are represented by the enum `Bop` which consists of four cases representing the different kinds of arithmetic operations: `Add`, `Sub`, `Mul`, and `Div`.

We refer to the cases of an algebraic data type as its *variants*. For example, `Expr` has variants `Num` and `BinOp`.

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

```
scala> import Expr._, Bop._
scala> val e = BinOp(Add, Num(3), BinOp(Mul, Num(4), Num(5)))
val e: Expr = BinOp(Add, Num(3), BinOp(Mul, Num(4), Num(5)))
```

Note that the enums `Expr` and `Bop` itself have companion objects. In turn, these companion objects have nested companion objects for the variants of the defined ADT within them. For example, the companion object for `Expr` contains companion objects for `Num` and `BinOp`. The nested companion objects `Num` and `BinOp` then have `apply` methods for constructing objects of the corresponding variant. The **`import`** instruction in the above code snippet makes the nested companion objects directly accessible in the code. For instance, in the code `Num(4)`, `Num` refers to the companion object of variant `Num` in `Expr`. Thus, this code expands to `Expr.Num.apply(4)`.

Second, the compiler adds natural implementations of the methods `toString`, `hashCode`, and `equals` for each variant. These will print, hash, and compare a whole tree consisting of the top-level enum 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 overriden `equals` method therefore ensures that enum instances are always compared structurally. For example, we have:

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

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, the compiler implicitly adds a **`val`** prefix to all arguments in the parameter list of an enum object, so that they are maintained as fields:

```
scala> val n = e1.left
val n: Expr = Num(3)
```

Note that the above code only works because we explicitly declared `e1` to be of type `BinOp` earlier. Without this type annotation, the inferred type of `e1` would be `Expr`. However, `Expr` objects are not guaranteed to have a field called `left` since `Expr` also includes `Num` objects.

Finally, the compiler adds a `copy` method to your enum cases for making modified copies. This method is useful if you need to create a copy of an existing enum object `o` that is identical to `o` except for some of `o`'s attributes:

```
scala> e1.copy(op = Sub)
val res4: Expr = 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 enumerations.

Let us first reformulate the three simplification rules in terms of our enum representation of expressions:

```
BinOp(Add, e1, Num(0)) => e1
BinOp(Mul, e1, Num(1)) => e1
BinOp(Mul, e1, Num(0)) => 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`:

```
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.

Each match 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 expression 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:

```
def pretty(e: Expr): String =
  e match
    case BinOp(bop, e1, e2) =>
      val bop_str = bop match
        case Add => "␣+␣"
        case Sub => "␣-␣"
        case Mul => "␣*␣"
        case Div => "␣/␣"
      "(" + pretty(e1) + bop_str + pretty(e2) + ")"
    case Num(n) => n.toString()
```

```
scala> val e = BinOp(Add, BinOp(Mult, Num(3), Num(4)), Num(1))
val e: BinOp = BinOp(Add, BinOp(Mult, Num(3), Num(4)), Num(1))
scala> pretty(e)
val 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 match alternative.
- *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 third 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 enum 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:

```
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 the third match alternative now binds the name `e2` to the value matched by the pattern `Num(0)`. This value is then returned on the righ-hand side of the rule by referring to it using the name `e2`.

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 `simplilfyTop`:

```
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. In general, a variable name such as `e1` may only be used once in a pattern. We can solve this problem by using a different variable name for the second subexpression, say `e2`, and then use a *pattern guard* to enforce that the subexpressions matched by `e1` and `e2` are equal:

```
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 Exhaustiveness Checks

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

```
def simplifyAll(e: Expr): 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(bop, e1, e2) =>
      BinOp(bop, simplifyAll(e), 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:

```
def simplifyAll(e: Expr): 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(bop, e1, e2) =>
      BinOp(bop, simplifyAll(e), simplifyAll(e2))
    case Num(_) => e
```

With complex patterns like this it can be tricky to keep track of all the possible cases. Fortunately, the compiler will automatically check whether the case analysis is exhaustive. For instance, consider the following faulty implementation of simplifyAll where we have omitted the last case for BinOp from the previous implementation:

```
def simplifyAll(e: Expr): 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 Num(_) => e
```

This version does not compile:

```
-- [E029] Pattern Match Exhaustivity Warning: ----------
2 |  e match
  |  ^
  |  match may not be exhaustive.
  |
  |  It would fail on pattern case: Expr.BinOp(_, _, _)
```

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:

```
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*:

```
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:

```
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 makes 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 `NullPointerException`!).

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 value `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 an enum as follows:

```
enum IntOption:
  case Some(value: Int)
  case None
```

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

```
def div(x: Int, y: Int): IntOption =
  if y == 0 then 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 where integer division by 0 is undefined:

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

```
def div(x: Int, y: Int): Option[Int] =
  if y == 0 then 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 later see that this formal approach allows us to prove mathematical theorems about individual programs and, more generally, about a programming language as a whole.

2.1 Notation

Throughout the course notes we will assume a basic understanding of mathematical notation and some concepts and notations from set theory. We briefly recap these in this section.

Properties. We use standard logical notation to express *properties*, i.e. formal mathematical statements that are true or false. For instance, $1 < 2$ is a true property and $2 < 1$ is a false property. For given properties P and Q, we introduce

- the *negation* of P, written $\neg P$, which is true iff[1] P is false,
- the *conjunction* of P and Q, written $P \wedge Q$, which is true iff both P and Q are true,
- the *disjunction* of P and Q, written $P \vee Q$, which is true iff P or Q is true,
- the *implication* of P and Q, written $P \Rightarrow Q$, which is true iff P implies Q (i.e., if P is true then Q is true),
- and the *equivalence* of P and Q, written $P \Leftrightarrow Q$, which is true iff P and Q are logically equivalent (i.e. $P \Rightarrow Q$ and $Q \Rightarrow P$).

[1] "iff" abbreviates "if and only if"

Properties may refer to variables (or unknowns). For instance, the property $0 \leq x \wedge x < 10$ is true for all natural numbers x between 0 and 9. We write $P(x_1, \ldots, x_n)$ to indicate that P refers to the variables x_1 to x_n. For a property $P(x)$, *universal quantification* over x yields the property $\forall x : P(x)$, which is true iff $P(x)$ is true for all possible values of x. Similarly, *existential quantification* over x yields the property $\exists x : P(x)$, which is true iff there exists at least one value for x such that $P(x)$ is true.

Sets. A *set* is an unordered collection of objects, which we refer to as the *elements* or *members* of the set. We write $e \in S$ to indicate that e is an element of the set S and we write $e \notin S$ for $\neg(e \in S)$. We write $\forall x \in S : P(x)$ as a short-hand for the property $\forall x : x \in S \Rightarrow P(x)$ and likewise write $\exists x \in S : P(x)$ for $\exists x : x \in S \wedge P(x)$.

We sometimes denote a set by explicitly enumerating its elements using the notation $\{a, b, c, d, e\}$. Here, $\{a, b, c, d, e\}$ is the set consisting of the elements a, b, c, d, and e. We denote the empty set by $\emptyset$ and we write $\mathbb{N}$ for the set of all natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ and $\mathbb{Z}$ for the set of all integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

At times, we will define a set in terms of a *set comprehension*, i.e., the set of all elements x that satisfy some given property $P(x)$, written $\{\, x \mid P(x) \,\}$. For example, the set comprehension $\{\, x \mid x \in \mathbb{Z} \wedge x < 0 \,\}$ defines the set of all negative integers.[2]

We define the usual operations on sets such as *union* $S \cup T$, *intersection* $S \cap T$, and *set difference* $S \setminus T$:

$$\begin{aligned} S \cup T &\stackrel{\text{def}}{=} \{\, x \mid x \in S \vee x \in T \,\} \\ S \cap T &\stackrel{\text{def}}{=} \{\, x \mid x \in S \wedge x \in T \,\} \\ S \setminus T &\stackrel{\text{def}}{=} \{\, x \mid x \in S \wedge x \notin T \,\} \end{aligned}$$

A set S is a *subset* of another set T, written $S \subseteq T$, if every element of S is also an element of T. For example, we have $\mathbb{N} \subseteq \mathbb{Z}$ and for all sets S we have $\emptyset \subseteq S$ and $S \subseteq S$. Two sets S and T are equal if they have the same elements, i.e. if $S \subseteq T$ and $T \subseteq S$.

Given a set S, we write 2^S for the set of all subsets of S:

$$2^S \stackrel{\text{def}}{=} \{\, x \mid x \subseteq S \,\}$$

We call 2^S the *powerset* of S. For instance, the powerset of $\{0, 1\}$ is $2^{\{0,1\}} = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$.[3]

Given two elements a and b, we construct a new element $\langle a, b \rangle$ called the *(ordered) pair* consisting of a and b. Formally, we can define $\langle a, b \rangle$ as the set

[2] Not every property $P(x)$ defines a set. A notorious example is the property $x \notin x$, which is at the heart of Russel's Paradox.

[3] The notation 2^S is reminiscent of the fact that if S is a finite set consisting of n elements, then the powerset of S has 2^n elements.

$\{\{a\}, \{a, b\}\}$.[4] We denote by $S \times T$ the *Cartesian product* of the sets S and T, which is the set of all pairs consisting of elements from S and T:

$$S \times T \overset{\text{def}}{=} \{\, \langle x, y\rangle \mid x \in S \wedge y \in T \,\}$$

We generalize these definitions to *n-tuples* $\langle a_1, \dots, a_n\rangle$ consisting of n elements a_1,...,a_n and Cartesian products over n sets $S_1 \times \cdots \times S_n$ for arbitrary $n \in \mathbb{N}$. In particular, for an element a, the 1-tuple $\langle a\rangle$ is equal to the set $\{a\}$ and the *empty tuple* $\langle\rangle$ is equal to the empty set.

Relations and Functions. Given two sets S and T, we call an element $R \in 2^{S\times T}$ a *binary relation* over S and T. That is, we have $R \subseteq S \times T$ by definition. If $\langle x, y\rangle \in R$ for some $x \in S$ and $y \in T$, we say that R *relates* x to y. We denote by $R^{-1} \subseteq T \times S$ the *inverse* of R:

$$R^{-1} \overset{\text{def}}{=} \{\, \langle y, x\rangle \mid \langle x, y\rangle \in R \,\}$$

R is called *total* if R relates every element of S to at least one element of T. R is called *functional* if R relates every element of S to at most one element of T. If R is functional, it is also called a *partial function* from S to T. If R is functional and total, it is called a *(total) function* from S to T. We write $R : S \rightharpoonup T$ to indicate that R is a partial function from S to T and $R : S \to T$ to indicate that it is a function from S to T. Every (total) function is also a partial function but the converse does not hold.

Let $f : S \rightharpoonup T$. We say that f is *defined* for $x \in S$ if there exists y such that $\langle x, y\rangle \in f$. In this case, we say that f *maps* x to y. We denote this unique y by $f(x)$. The *domain* of f, written $\mathsf{dom}(f)$, is the set of all $x \in S$ for which f is defined. Note that f is total iff $\mathsf{dom}(f) = S$.

We typically define a (partial) function from S to T using one or more equations that define $f(x)$ for all $x \in \mathsf{dom}(f)$. For example, the following definition defines the *successor* function on natural numbers:

$$\begin{aligned} \mathit{succ} &: \mathbb{N} \to \mathbb{N} \\ \mathit{succ}(x) &= x + 1 \end{aligned}$$

That is, $\mathit{succ} = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \dots\}$.

For a partial function $f : S \rightharpoonup T$ and any elements a and b, we denote by $f[a \mapsto b]$ the partial function from $S \cup \{a\}$ to $T \cup \{b\}$ defined as follows:

$$f[a \mapsto b](x) \overset{\text{def}}{=} \begin{cases} f(x) & x \in \mathsf{dom}(f) \setminus \{a\} \\ b & x = a \\ \text{undefined} & \text{otherwise} \end{cases}$$

That is, $f[a \mapsto b]$ maps a to b and otherwise behaves like f. Note that we allow $a \in S$ and $b \in T$ but we do not require this.

[4] Convince yourself that with this definition we have $\langle a, b\rangle = \langle a', b'\rangle$ if and only if $a = a'$ and $b = b'$.

For a binary relation $R \subseteq S \times S$, we define its *reflexive closure* as the relation $R \cup \{\, \langle x, x \rangle \mid x \in S \,\}$. We further define its *transitive closure*, denoted R^+, as the smallest relation that satisfies the following two conditions:

1. $R \subseteq R^+$, and
2. if $\langle x, y \rangle \in R^+$ and $\langle y, z \rangle \in R^+$ then $\langle x, z \rangle \in R^+$.

We denote by R^* the reflexive and transitive closure of R.

2.2 Structural Recursion and Induction

Recursion is a constructive technique for describing infinite sets (and infinite functions and relations on these sets). Induction is a technique for proving properties about recursively defined sets, functions and relations. 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.

2.2.1 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 certain tuples. Once we have defined N, we will build structurally recursive functions on N that correspond to addition and multiplication, and then prove properties of these functions.

We represent the natural numbers as follows, using only nesting of tuples: we start by 0, which we represent as the empty tuple $\langle\rangle$. Then, given a natural number n, we construct its successor $n + 1$ by wrapping n in another tuple:

$$\begin{array}{ll} 0 & \langle\rangle \\ 1 & \langle\langle\rangle\rangle \\ 2 & \langle\langle\langle\rangle\rangle\rangle \\ \multicolumn{2}{c}{\dots} \end{array}$$

That is, the number 1 is represented by the tuple that contains the empty tuple, the number 2 is the tuple that contains the tuple containing the empty tuple, etc. For the set N, we choose exactly those tuples that represent natural numbers following the above convention[5].

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:

[5] This construction of the natural numbers is similar to the standard set-theoretic construction of natural numbers, except that we use a slightly simpler construction rule for successors here.

1. The empty tuple $\langle\rangle$ is an element of N.

2. If x is an element of N, then the tuple $\langle x\rangle$ 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 more compactly using the following *inference rules*:

$$\begin{array}{c} \textsc{Rule 1} \\ \langle\rangle \in N \end{array} \qquad\qquad \begin{array}{c} \textsc{Rule 2} \\ \dfrac{x \in N}{\langle x\rangle \in N} \end{array}$$

Rule 3 is left implicit.

In general, an inference rule takes the form

$$\frac{P_1 \qquad \ldots \qquad P_n}{C}$$

Such a rule states that if the properties $P_1, \ldots, P_n$ hold, then also C holds, $P_1 \wedge \ldots \wedge P_n \Rightarrow C$. 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* and we omit the line separating the premises from the conclusion in this case. For instance, Rule 1 above is an axiom.

Using 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 four construction steps:

1.	$\langle\rangle$	with rule 1
2.	$\langle\langle\rangle\rangle$	with rule 2
3.	$\langle\langle\langle\rangle\rangle\rangle$	with rule 2
4.	$\langle\langle\langle\langle\rangle\rangle\rangle\rangle$	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. Whenever we give a recursive definition of a set in terms of inference rules, then this additional requirement (which corresponds to Rule 3) is always implicit.

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

$$N = \{\langle\rangle\} \cup \{\, \langle x\rangle \mid x \in N \,\}$$

That is, N is defined as the smallest set (with respect to subset inclusion) that satisfies this recursive equation. We also say that N is the *least fixpoint* of the equation. The left hand side of the equation captures the two construction rules for the elements of N. The fact that N is defined as the smallest set that satisfies the equation (i.e., the least fixpoint rather than any other fixpoint) captures Rule 3.

All of the above definitions of the set N are equivalent, i.e., they define the exact same set of mathematical objects. We will mostly work with definitions of recursive sets that are given in the form of inference rules as these are usually more intuitive than those given as solutions of fixpoint equations.

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, $\langle\rangle$ can only be constructed with the first rule and $\langle\langle\rangle\rangle$ only with the second rule.

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

In general, we will work with structurally recursive definitions that have more than one base case rule and more than one recursive rule.

Algebraic data type definitions in Scala can be viewed as structurally recursive definitions of sets of tuples. For example the following declarations give a possible Scala encoding of our definition of the set N:

```
enum N:
  case Zero // rule 1
  case Succ(x: N) // rule 2
```

We have one **case** declaration per construction rule of N. The implicit Rule 3 is enforced by Scala: **enum** types cannot be extended using subtyping.

2.2.2 Recursive Definitions of Functions

The real power of structural recursion is that we can also use it to define functions that operate on the elements of a recursive set. For example, suppose we want to define a function $D : N \to \mathbb{N}$ that maps the elements of N to the natural numbers they represent. We can do this as follows:

$$\begin{aligned}
D &: N \to \mathbb{N}\\
D(\langle\rangle) &= 0\\
D(\langle x\rangle) &= 1 + D(x)
\end{aligned}$$

This definition has two important properties that make it a *structurally recursive function 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, which ensures that D is a partial function.

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 $\langle x \rangle$ is on the proper subobject x.) This ensures that D is total.

Hence, together, these properties guarantee that the rules for D define a function, i.e., D is well-defined. We can also view the above definition as a blue print for the declaration of a Scala function on our algebraic data type representation of `N`:

```
def D(y: N): Int =
  y match
    case Zero => 0
    case Succ(x) => 1 + D(x)
```

From the fact that the definition of the function D 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 $\oplus : N \times N \to N$ that corresponds to addition on natural numbers:

$$\begin{aligned} \oplus &: N \times N \to N \\ \langle\rangle \oplus y &= y \\ \langle x \rangle \oplus y &= \langle x \oplus y \rangle \end{aligned}$$

We use the symbol $\oplus$ for this function instead of $+$ to distinguish it from the actual addition function $+ : \mathbb{N} \times \mathbb{N}$ on natural numbers. We will see below how these two functions are formally related.

Note that the second rule in the definition of D determines how the value of D for larger arguments is constructed from values of D for smaller arguments. In the case of $\oplus$, the recursion only goes over the first argument of the function.

2.2.3 Structural Induction

Often, we want to prove that a particular property holds for all elements of a recursively defined set. For example, we may want to prove that the function $\oplus$ corresponds to the actual addition function $+$ on natural numbers. In other words, we may want to prove that the property

$$D(x \oplus y) = D(x) + D(y)$$

holds for all elements $x, y \in N$.

In general, if we want to prove that all $x \in N$ satisfy a given property A, we can proceed as follows:

1. Prove that $\langle\rangle$ satisfies A.

2. Prove that for all $x \in N$, if x satisfies A, then $\langle x \rangle$ satisfies A.

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

$$\frac{A(\langle\rangle) \qquad \forall x \in N : A(x) \Rightarrow A(\langle x \rangle)}{\forall x \in N :\ A(x)}$$

Notice that the premises of this inference rule

(1) $A(\langle\rangle)$

(2) $\forall x \in N : A(x) \Rightarrow A(\langle x \rangle)$

are derived directly from the defining rules of N. The first premise is called the *base case* of the induction rule and the second premise is called the *induction step*. The left-hand side of the implication in the induction step is called the *induction hypothesis*. When we prove $A(\langle x \rangle)$ in the induction step for a particular A, we can assume that the induction hypothesis $A(x)$ holds.

For every set that is defined by structural recursion there is a corresponding induction principle that is derived from the definition of the set. Every rule that is an axiom yields a base case and every other rule yields an induction step.

In the next section, we will prove the correctness of the induction principle for N by proving the correctness of a more general induction principle. However, you should already try to develop some intuition for why the induction principle for N is correct. 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 principle hold. We validate that from these premises follows the validity of

$$A(\langle\langle\langle\langle\rangle\rangle\rangle\rangle)$$

That is, we want to validate that A holds for our representation of the number 3. First, from premise (1) follows:

$$A(\langle\rangle)$$

From this fact and premise (2) we conclude:

$$A(\langle\langle\rangle\rangle)$$

Applying premise (2) twice more we obtain the property we wanted to show:

$$A(\langle\langle\langle\langle\rangle\rangle\rangle\rangle) \ .$$

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

Let us use the induction principle to prove the equivalence of the functions $\oplus$ on N and $+$ on $\mathbb{N}$. To this end, we define the property

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

and then prove that for all $x \in N$, $A(x)$ holds. For the proof to succeed, it is important that we let the induction go over x rather than y because we will need to use the recursive definitions of the functions D and $\oplus$ in the proof. Recall that the recursion for $\oplus$ goes over its first argument, which is x in this case.

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

(1) $\forall y \in N : D(\langle\rangle \oplus y) = D(\langle\rangle) + D(y)$

(2) $\forall x \in N : (\forall y \in N : D(x \oplus y) = D(x) + D(y))$
$\Rightarrow (\forall y \in N : D(\langle x\rangle + y) = D(\langle x\rangle) + D(y))$

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. $\forall x \in N : \forall y \in N : D(x \oplus y) = D(x) + D(y)$

Proof. By structural induction over $x \in N$:

Let $x = \langle\rangle$ and $y \in N$. Then

$$\begin{aligned} D(x \oplus y) &= D(\langle\rangle + y) & \\ &= D(y) & \text{Definition of } \oplus \\ &= 0 + D(y) & 0 \text{ neutral element of } + \\ &= D(\langle\rangle) + D(y) & \text{Definition of } D \\ &= D(x) \oplus D(y) & \end{aligned}$$

Let $x = \langle x'\rangle$ and $y \in N$. Then

$$\begin{aligned} D(x \oplus y) &= D(\langle x'\rangle \oplus y) & \\ &= D(\langle x' \oplus y\rangle) & \text{Definition of } \oplus \\ &= 1 + D(x' \oplus y) & \text{Definition of } D \\ &= 1 + (D(x') + D(y)) & \text{Induction hypothesis} \\ &= (1 + (D(x')) + D(y) & \text{Associativity of } + \\ &= D(\langle x'\rangle) + D(y) & \text{Definition of } D \\ &= D(x) + D(y) & \end{aligned}$$

□

2.2.4 Well-founded Induction (optional)

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

Let X be a set and $\succ \subseteq X \times X$ a binary relation over X. A (possibly infinite) sequence $x_1, x_2, x_3, \ldots$ of elements in X is called a *descending chain* of $\succ$ if $x_{i-1} \succ x_i$ for all members x_i of the sequence, $i > 0$. The relation $\succ$ is called *well-founded* if it has no infinite descending chains. 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 starting from any element $x_1 \in X$, we cannot find smaller and smaller elements $x_1 \succ x_2 \succ x_3 \succ \cdots$. At some point a long the chain, we must arrive at an element x_n such that no other element in X is smaller than x_n.

As an example, let us reconsider the set N. The relation

$$\succ_N : N \times N$$
$$x \succ_N y \overset{\text{def}}{\Longleftrightarrow} x = \langle y \rangle$$

is a well-founded relation on N.[6] The reflexive and transitive closure of $\succ_N$ corresponds to the canonical ordering on natural numbers $\geq \subseteq \mathbb{N} \times \mathbb{N}$.

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. □

Let $(X, \succ)$ be a well-founded set and $A(x)$ a property for $x \in X$. We call the following inference rule the *well-founded induction principle*[7] for X, $\succ$, and A:

$$\frac{\forall x \in X : \; (\forall y \in X : x \succ y \Rightarrow A(y)) \Rightarrow A(x)}{\forall x \in X : \; A(x)}$$

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.

[6] This follows from our definition of tuples in terms of sets and the *axiom of foundation* of set theory.

[7] Sometimes, well-founded induction is also called Noetherian induction, named 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. □

The rule for well-founded induction only has a single premise and does not distinguish between base case and induction step. This is possible due to the more general formulation of the induction hypothesis in the premise of this rule. When we instantiate the rule for a particular well founded set $(X, \succ)$, then the base case and induction step typically emerge from further case analysis. For example, the induction principle for N is obtained from the well-founded induction principle by instantiating the latter with N for X and $\succ_N$ for $\succ$. The premise of the instantiated rule is:

$$\forall x \in N : (\forall y \in N : x \succ_N y \Rightarrow A(y)) \Rightarrow A(x)$$

To see that this premise is equivalent to the two premises of the induction principle for N, let us first replace $\succ_N$ by its definition:

$$\forall x \in N : (\forall y \in N : x = \langle y \rangle \Rightarrow A(y)) \Rightarrow A(x)$$

Now, in order to prove this premise for a particular A, we have to distinguish the two possible cases how each $x \in N$ was constructed, corresponding to the structurally recursive definition of N. This gives us two cases that we need to consider:

1. $(\forall y \in N : \langle\rangle = \langle y \rangle \Rightarrow A(y)) \Rightarrow A(\langle\rangle)$
2. $\forall x' \in N : (\forall y \in N : \langle x' \rangle = \langle y \rangle \Rightarrow A(y)) \Rightarrow A(\langle x' \rangle)$

The first case simplifies to $A(\langle\rangle)$ (i.e., the first premise of the induction principle for N) because $\langle\rangle = \langle y \rangle$ does not hold for any $y \in N$ and hence the left side of the outer implication is trivially true. The second case simplifies to

$$\forall x' \in N : (\forall y \in N : x' = y \Rightarrow A(y)) \Rightarrow A(\langle x' \rangle)$$

which can be further simplified to just

$$\forall x' \in N : A(x') \Rightarrow A(\langle x' \rangle) \ .$$

Renaming x' to x yields the second premise of the induction principle for N.

Note that well-founded relations are also closely related to the notion of termination measures that we discussed in Section 1.3.1.

2.3 Lists

Lists are one of the most important data structures in functional programming languages. We will consider lists that are sequences 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*. That is, once a list has been created, it cannot be changed, e.g. by removing or adding an element in the middle of the list. Such data structures are also called *persistent*.

Persistent data structures have the advantage that their representation in memory can be shared across different instances of the data structure. For example, the two lists $1, 4, 3$ and $5, 2, 4, 3$ have the common sublist $4, 3$. If the two lists are stored in memory at the same time, the shared sublist $4, 3$ only needs to be represented once. If used properly, this feature yields space-efficient, high-level implementations of algorithms over persistent lists. In this section, we will define persistent lists using structural recursion and see that this definition corresponds to the `List` data type defined in Scala's standard library.

2.3.1 Defining Lists using Structural Recursion

Mathematically, we can represent lists of integers 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:

$$\langle\rangle \in \mathit{List} \qquad\qquad \frac{\mathit{hd} \in \mathbb{Z} \qquad \mathit{tl} \in \mathit{List}}{\langle \mathit{hd}, \mathit{tl}\rangle \in \mathit{List}}$$

For a non-empty list ℓ of the form $\langle \mathit{hd}, \mathit{tl}\rangle$, we refer to the integer number hd as the *head* of ℓ, and we call the remaining list tl the *tail* of ℓ. 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 *nil* and write $\mathit{hd} :: \mathit{tl}$ for a cons cell $\langle \mathit{hd}, \mathit{tl}\rangle$. We treat $::$ as a right-associative constructor for lists. That is, $x :: y :: \mathit{tl}$ is the list $\langle x, \langle y, \mathit{tl}\rangle\rangle$.

In Scala, we can define lists of integers using an algebraic data type[8]:

```
enum List:
  case Nil
  case Cons(hd: Int, tl: List)
```

Here, `Nil` represents the empty list and a cons cell $\mathit{hd} :: \mathit{tl}$ is represented by `Cons(hd, tl)`. Note again the close resemblance between the mathematical definition and the Scala definition of lists.

As an example, let us construct a Scala list containing the values $1, 4, 2$:

[8] 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 later.

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

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

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

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.
```

2.3.2 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:

$$\begin{aligned} \mathit{length} &: \mathit{List} \to \mathbb{N} \\ \mathit{length}(\mathit{nil}) &= 0 \\ \mathit{length}(\mathit{hd} :: \mathit{tl}) &= 1 + \mathit{length}(\mathit{tl}) \end{aligned}$$

The next function is more interesting, it takes two lists ℓ_1 and ℓ_2 and creates a new list by concatenating ℓ_1 and ℓ_2.

$$\begin{aligned} \mathit{append} &: \mathit{List} \times \mathit{List} \to \mathit{List} \\ \mathit{append}(\mathit{nil}, \ell_2) &= \ell_2 \\ \mathit{append}(\mathit{hd} :: \mathit{tl}, \ell_2) &= \mathit{hd} :: \mathit{append}(\mathit{tl}, \ell_2) \end{aligned}$$

For example, for $\ell_1 = 4 :: 6 :: 1 :: \mathit{nil}$ and $\ell_2 = 5 :: 1 :: \mathit{nil}$ we get

$$\mathit{append}(\ell_1, \ell_2) = 4 :: 6 :: 1 :: 5 :: 1 :: \mathit{nil} \ .$$

Finally, using *append* we can define a function *reverse* that takes a list ℓ and creates a new list that contains the elements of ℓ in reverse order:

$$\begin{aligned} \mathit{reverse} &: \mathit{List} \to \mathit{List} \\ \mathit{reverse}(\mathit{nil}) &= \mathit{nil} \\ \mathit{reverse}(\mathit{hd} :: \mathit{tl}) &= \mathit{append}(\mathit{reverse}(\mathit{tl}), \mathit{hd} :: \mathit{nil}) \end{aligned}$$

For example, we have $\mathit{reverse}(4 :: 2 :: \mathit{nil}) = 2 :: 4 :: \mathit{nil}$.

Note that the definition of *reverse* is still structurally recursive since in the recursive case *reverse* is only applied to the tail *tl* of the input list.

The mathematical definitions of *length*, *append*, and *reverse* directly translate to corresponding Scala functions:

```
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 `reverse` is quadratic in the length of the list `l`. Moreover, the `reverse` function is not tail-recursive and hence requires linear space in the length of `l`. The implementations of `length` and `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 `reverse` so that it runs in linear time and constant space:

```
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* `reverse2` *function. Call this function $reverse_2$. Hint: to define $reverse_2$, first define an auxiliary function $rev : List \times List \to List$.*

Exercise 2.2. *Define tail-recursive Scala versions of the functions* `length` *and* `append`*. Hint: use* `reverse2` *in the definition of* `append`*.*

2.3.3 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. Following the discussion in Section 2.2.3, we derive the following structural induction principle for *List*:

$$\frac{A(nil) \qquad \forall hd \in \mathbb{N}, tl \in List : A(tl) \Rightarrow A(hd :: tl)}{\forall \ell \in List :\; A(\ell)}$$

As an example, the following proposition states that the length of a list obtained by appending two lists ℓ_1 and ℓ_2 is equal to the sum of the lengths of ℓ_1 and ℓ_2.

Proposition 2.4. *For all $\ell_1, \ell_2 \in List$ the following property holds*

$$length(append(\ell_1, \ell_2)) = length(\ell_1) + length(\ell_2) \ .$$

Proof. Let $\ell_1, \ell_2 \in List$. Since *append* is defined by structural recursion on its first argument, the proof proceeds by structural induction on ℓ_1: Base case: assume $\ell_1 = nil$. Then

$$\begin{aligned} length(append(\ell_1, \ell_2)) &= length(append(nil, \ell_2)) \\ &= length(\ell_2) && \text{Def. of } append \\ &= 0 + length(\ell_2) && \text{0 is neutral element} \\ &= length(nil) + length(\ell_2) && \text{Def. of } length \\ &= length(\ell_1) + length(\ell_2) \end{aligned}$$

Induction step: assume $\ell_1 = hd :: tl$. Then

$$\begin{aligned} length(append(\ell_1, \ell_2)) &= length(append(hd :: tl, \ell_2)) \\ &= length(hd :: append(tl, \ell_2)) && \text{Def. of } append \\ &= 1 + length(append(tl, \ell_2)) && \text{Def. of } length \\ &= 1 + (length(tl) + length(\ell_2)) && \text{Induction hypothesis} \\ &= (1 + length(tl)) + length(\ell_2) && \text{Associativity of } + \\ &= length(hd :: tl) + length(\ell_2) && \text{Def. of } length \\ &= length(\ell_1) + length(\ell_2) \end{aligned}$$

□

Exercise 2.3. *For the function $reverse_2$ that you defined in Exercise 2.1, use structural induction to prove that it computes the same function as reverse. That is, for all $\ell \in List$, $reverse(\ell) = reverse_2(\ell)$.*

Chapter 3

Syntax

When we describe a programming language, we distinguish between the *syntax* and the *semantics* of the language. The syntax describes the structure of a program (i.e., how a program is represented), 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 syntax and semantics.

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 (i.e., the actual source code stored in text files). 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. Abstract syntax trees are used to represent programs internally in a compiler or interpreter.

3.1 Concrete Syntax (optional)

The *parsing problem* is concerned with the conversion of program source code represented as sequences of characters (i.e., concrete syntax) into abstract syntax trees. This is a well-understood problem and you can learn more about it in a compiler construction course. In this course, we will mostly side-step the parsing problem and directly work with abstract syntax trees. 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.

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’.

Given an alphabel Σ, we denote by Σ^* the set of all finite words that can be formed using the symbols in Σ[1] The *empty word* is denoted by ϵ.

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:

$$\begin{aligned}
&E \to E\,O\,E\\
&E \to (E)\\
&E \to x \qquad \text{where } x \in \mathbb{Z}\\
&O \to \texttt{+}\\
&O \to -\\
&O \to \texttt{*}\\
&O \to \texttt{/}
\end{aligned}$$

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

$$\begin{aligned}
&\ldots\\
&E \to -2\\
&E \to -1\\
&E \to 0\\
&E \to 1\\
&E \to 2\\
&\ldots
\end{aligned}$$

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. We call this process *derivation*.

[1] The notation Σ^* should not to be confused with the notation R^* introduced earlier where R is a binary relation, which denotes the reflexive transitive closure of R.

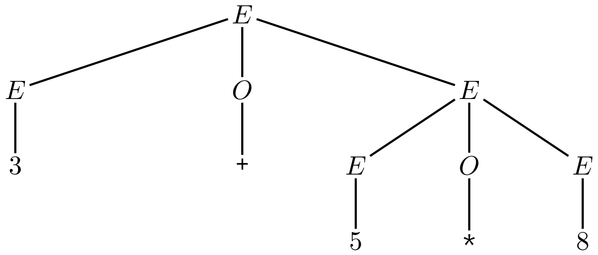
Using the above productions, we can derive words such as

$$\begin{array}{l} 1 \\ 3 + 5 * 8 \\ -14 / (42 + (0 - 1)) \end{array}$$

Here is a derivation of the word $3 + 5 * 8$:

$$\begin{aligned} E &\Rightarrow E\,O\,E \\ &\Rightarrow 3\,O\,E \\ &\Rightarrow 3 + E \\ &\Rightarrow 3 + E\,O\,E \\ &\Rightarrow 3 + 5\,O\,E \\ &\Rightarrow 3 + 5 * E \\ &\Rightarrow 3 + 5 * 8 \end{aligned}$$

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 a context-free language, each derivation step rewrites a single nonterminal symbol in the working word regardless of the context in which this nonterminal occurs. The concrete syntax of most programming languages is context-free. 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

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

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

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 $X \to w$ in P and $u_1, u_2 \in (\Sigma \cup N)^*$ such that $u = u_1 X u_2$ and $v = u_1 w u_2$. That is, v is the result of applying $X \to w$ to u. We denote by $\Rightarrow^*$ the reflexive and transitive closure of the relation $\Rightarrow$ 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.

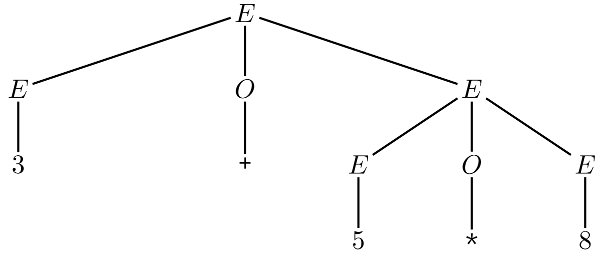
3.1.3 Backus-Naur-Form

Often, context-free grammars are given in so-called *Backus-Naur-Form* (BNF). In this form, we use the symbol ::= instead of $\to$ to separate the two sides of a production. Moreover, in a BNF, productions $X \to w_1, \ldots, X \to w_n$ 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:

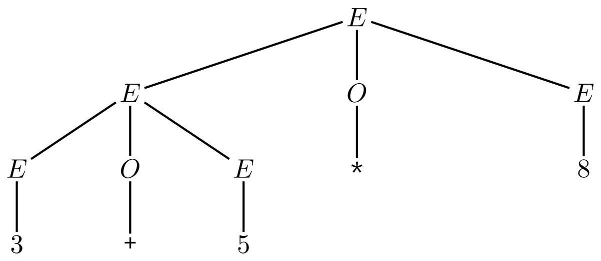
$$\begin{aligned} &x \in \mathbb{Z} \\ &\quad E ::= E\,O\,E \mid (E) \mid x \\ &\quad O ::= \texttt{+} \mid - \mid \texttt{*} \mid \texttt{/} \end{aligned}$$

3.1.4 Eliminating Ambiguity

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



This is not the only possible derivation of “3+5*8“. Another one is given by the following parse tree:



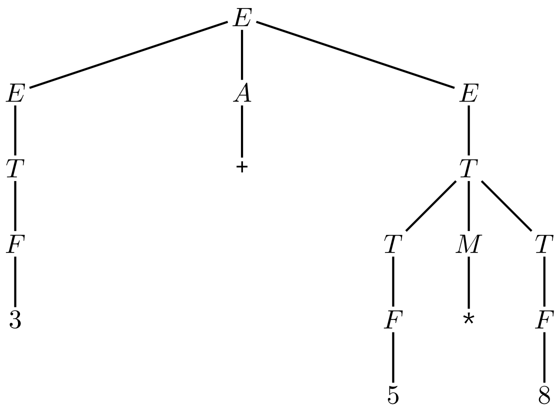
We call a grammar in which a word has more than one derivation *ambiguous*. Ambiguity is a problem because the semantics of programs is given in terms of their abstract syntax trees, which are derived from parse trees. Typically, the semantics of a program depends on the structure of its parse tree. For example, with the canonical semantics of arithmetic expressions, the first parse tree of the expression “3 + 5 * 8“ 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, formalizing the rule of arithmetic notation stating “multiplication before addition”. 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.

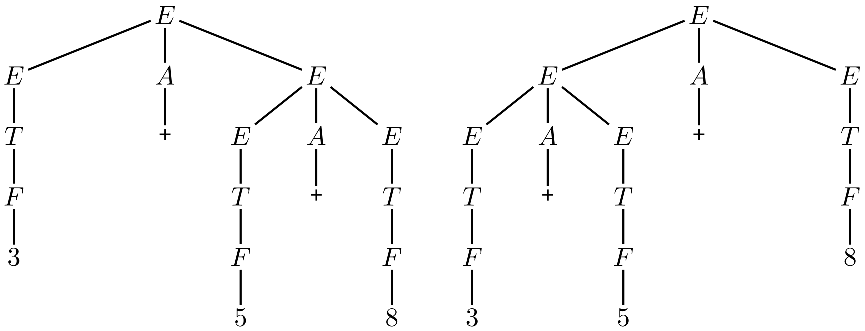
The ambiguity in our arithmetic expression grammar that we have observed for the expression “3 + 5 * 8” stems from the fact that the grammar does not distinguish between the additive operators, ’+’ and ’-’, and the multiplicative operators, ’*’ and ’/’. We would like the multiplicative operators to *bind 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:

$$
\begin{aligned}
E &::= E\,A\,E \mid T \\
A &::= \texttt{+} \mid \texttt{-} \\
T &::= T\,M\,T \mid F \\
M &::= \texttt{*} \mid \texttt{/} \\
F &::= x \mid (E)
\end{aligned}
$$

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



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 arithmetic overflow. 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:

$$
\begin{aligned}
E &::= E\,A\,T \mid T \\
A &::= \texttt{+} \mid \texttt{-} \\
T &::= T\,M\,F \mid F \\
M &::= \texttt{*} \mid \texttt{/} \\
F &::= x \mid (E)
\end{aligned}
$$

3.1.5 Regular Languages

Finally, let us modify our grammar for arithmetic expressions so that it is actually context-free, i.e., so that 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:

$$\begin{aligned} Z &::= \texttt{-}H \mid H \\ H &::= 0 \mid 1T \mid \cdots \mid 9T \\ T &::= \epsilon \mid 0T \mid \cdots \mid 9T \end{aligned}$$

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

Next, we combine this grammar with the grammar:

$$\begin{aligned} E &::= E\,A\,T \mid T \\ A &::= \texttt{+} \mid \texttt{-} \\ T &::= T\,M\,F \mid F \\ M &::= \texttt{*} \mid \texttt{/} \\ F &::= Z \mid (E) \end{aligned}$$

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:

$$\begin{aligned} X &::= \epsilon \\ X &::= a \\ X &::= Y \\ X &::= aY \end{aligned}$$

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*. Another way to describe such languages are *regular expressions*, which you may already be familiar with.

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 automatically generated from regular expressions, whereas the actual parser is automatically generated from a context-free grammar defined over the token alphabet.

3.2 Abstract Syntax

In the previous section, 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. In this section, we study the mathematical objects that describe the abstract syntax of programming languages.

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:

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

Note that in the definition of *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.

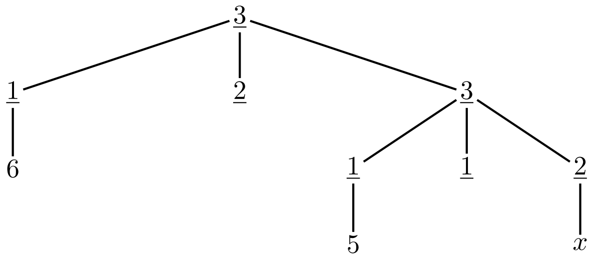
The expression language includes variables x which are drawn from an (infinite) set of variable names *Var*. The sets *Num* and *Var* are parameters of the grammar definition. In the following, you may assume that these two sets are just referring to integer numbers $\mathit{Num} = \mathit{Var} = \mathbb{Z}$.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

Our grammar borrows the notation of the concrete syntax to represent the elements of the sets *Expr* and *Bop*. However, we just use the concrete syntax to "sugarcoat" 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 *variant numbers.* In our running example, we obtain the following tagged grammar of arithmetic expressions (for clarity, the variant numbers are underlined):

$$\begin{array}{rll} e \in \mathit{Expr} ::= & \underline{1} : n \mid \underline{2} : x \mid \underline{3} : e_1\, \mathit{bop}\, e_2 & \text{expressions} \\ \mathit{bop} \in \mathit{Bop} ::= & \underline{1} : \texttt{+} \mid \underline{2} : \texttt{*} & \text{binary operators} \end{array}$$

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 $6\,\texttt{*}(5\,\texttt{+}\,x)$ is notational sugar for the following abstract syntax tree:



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

$$\langle \underline{3}, \langle \underline{1}, 6\rangle, \langle \underline{2}\rangle, \langle \underline{3}, \langle \underline{1}, 5\rangle, \langle \underline{1}\rangle, \langle \underline{2}, x\rangle\rangle\rangle \enspace .$$

Consequently, the grammar rules for the sets *Expr* and *Bop* really stand for inference rules that define the sets *Expr* and *Bop* using structural recursion on tuples:

$$\frac{n \in \mathit{Num}}{\langle \underline{1}, n\rangle \in \mathit{Expr}} \qquad \frac{x \in \mathit{Var}}{\langle \underline{2}, x\rangle \in \mathit{Expr}} \qquad \frac{e_1, e_2 \in \mathit{Expr} \qquad \mathit{bop} \in \mathit{Bop}}{\langle \underline{3}, e_1, \mathit{bop}, e_2\rangle \in \mathit{Expr}}$$

$$\langle \underline{1}\rangle \in \mathit{Bop} \qquad\qquad \langle \underline{2}\rangle \in \mathit{Bop}$$

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 $\langle \underline{1}, 4\rangle$ 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 occuring in e:

$$\begin{aligned}
ov &: \mathit{Expr} \to 2^{\mathit{Var}}\\
ov(n) &= \emptyset\\
ov(x) &= \{x\}\\
ov(e_1\, \mathit{bop}\, e_2) &= ov(e_1) \cup ov(e_2)
\end{aligned}$$

Recall from Section 2.1 that by 2^{Var} we denote the powerset of the set Var.

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

```
enum Expr:
  case Num(n: Double)
  case Var(x: String)
  case BinOp(bop: Bop, e1: Expr, e2: Expr)

enum Bop:
  case Plus, Times
```

Here, the variant constructors `Num`, `Var`, `BinOp` and so forth, play the role of the variant numbers in our mathematical representation of the abstract syntax.

Note that we have introduced a minor discrepancy to the mathematical definition of *Expr* and its Scala representation using the type `Expr`: in the `BinOp` case the binary operator `bop` is the first argument (i.e. the left-most child of the node) rather than the second argument. This change in the representation slightly improves the readability of the code in the following, but it is otherwise nonessential.

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

```
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 generic type `Set` from the Scala standard library, which can represent finite sets of objects. That is, the type `Set[String]` represents all finite sets of string objects.

3.2.2 Environments and Expression Evaluation

Consider the expression $e = x \texttt{*} (y \texttt{+} 6)$. 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 $\mathit{env} : \mathit{Var} \rightharpoonup \mathit{Num}$, that assigns variables to values. Recall that 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

$$\mathit{env} = \{(x, 1), (y, 2)\}$$

We typically use the following arrow notation for the individual variable assignments in an environment

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

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

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

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 formalize this idea by defining a function eval using structural recursion:

$$\begin{aligned} \mathit{eval} &: \mathit{Env} \times \mathit{Expr} \to \mathit{Num} \\ \mathit{eval}(\mathit{env}, n) &= n \\ \mathit{eval}(\mathit{env}, x) &= \mathit{env}(x) \\ \mathit{eval}(\mathit{env}, e_1 \texttt{+} e_2) &= \mathit{eval}(\mathit{env}, e_1) + \mathit{eval}(\mathit{env}, e_2) \\ \mathit{eval}(\mathit{env}, e_1 \texttt{*} e_2) &= \mathit{eval}(\mathit{env}, e_1) \cdot \mathit{eval}(\mathit{env}, e_2) \end{aligned}$$

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 $\langle \underline{1} \rangle \in \mathit{Bop}$. On the right side it stands for the mathematical addition operation on integers. We use different fonts to distinguish these different usages of the symbol.

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

Intuitively, we can think of an environment *env* as the mathematical representation of the program stack, which keeps track of the values of local variables during program execution.

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:

```
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:

```
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)
    case BinOp(Times, e1, e2) =>
      eval(env, e2) * 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:

```
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)
  eval(e)
```

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:

$$3 * x + x$$

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:

$$3 * (x + 4) + (x + 4)$$

Let e and e_s be expressions and $x \in \mathit{Var}$ a variable, then we denote by

$$e[e_s/x]$$

the expression that we obtain by replacing all occurrences of x in e by e_s. In particular, we have:

$$(3 * x + x)[(x + 4)/x] = 3 *(x + 4) +(x + 4)$$

We can define the substitution function formally using structural recursion:

$$\begin{aligned}
\_[\_/\_] &: \mathit{Expr} \times \mathit{Var} \times \mathit{Expr} \to \mathit{Expr} \\
n[e_s/x] &= n \\
y[e_s/x] &= \textbf{if } x = y \textbf{ then } e_s \textbf{ else } y \\
(e_1 \, \mathit{bop} \, e_2)[e_s/x] &= (e_1[e_s/x]) \, \mathit{bop} \, (e_2[e_s/x])
\end{aligned}$$

The Scala version of the substitution function looks as follows:

```
def subst(e: Expr, x: String, es: Expr): Expr =
  e match
    case Num(_) => e
    case Var(y) => if x == y then es else e
    case BinOp(bop, e1, e2) =>
      BinOp(bop, subst(e1, x, es), subst(e2, x, es))
```

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

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

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

Proof. By structural induction on e (exercise). □

3.3 Binding and Scoping

In programming languages, identifiers are used as names for values. They are introduced through variable and constant declarations, procedures, class declarations, etc. 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. We also study an important operation on expressions called *substitution*, which is crucial for many tasks related to programming languages, including program evaluation, optimization, and refactoring.

3.3.1 Expressions with Constant Declarations

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

$$\texttt{const}\ x = e_d \,;\ e_b$$

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 complete abstract syntax of our extended language is now as follows:

$$
\begin{array}{rlr}
n \in \mathit{Num} & & \text{numbers} \\
x \in \mathit{Var} & & \text{variables} \\
e \in \mathit{Expr} ::= & n \mid x \mid e_1 \mathbin{\mathit{bop}} e_2 \mid \mathbf{\mathtt{const}}\ x = e_d\,;\ e_b & \text{expressions} \\
\mathit{bop} \in \mathit{Bop} ::= & \mathtt{+} \mid \mathtt{*} & \text{binary operators}
\end{array}
$$

Note that in our concrete syntax, we omit parenthesis in sequenced **`const`** declarations. For example, the expression

$$\mathbf{\mathtt{const}}\ x = 5;\ \mathbf{\mathtt{const}}\ y = x + 2;\ \mathbf{\mathtt{const}}\ z = y * 5;\ z + z$$

is implicitly parenthesized as follows:

$$\mathbf{\mathtt{const}}\ x = 5;\ (\mathbf{\mathtt{const}}\ y = x + 2;\ (\mathbf{\mathtt{const}}\ z = y * 5;\ z + z))$$

However, for clarity, we will use parenthesis around **`const`** declarations that occur nested inside the defining expressions of other **`const`** declarations:

$$\mathbf{\mathtt{const}}\ y = (\mathbf{\mathtt{const}}\ x = 5;\ x + 2);\ \mathbf{\mathtt{const}}\ z = y * 5;\ z + z$$

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:

$$\mathbf{\mathtt{const}}\ x = 2 * 3;\ \mathbf{\mathtt{const}}\ y = x + 5;\ x * (z + y)$$

If we over-line all defining occurrences of variables in this expression, we get the following:

$$\mathbf{\mathtt{const}}\ \overline{x} = 2 * 3;\ \mathbf{\mathtt{const}}\ \overline{y} = x + 5;\ x * (z + y)$$

In addition to defining and using occurrences, we also distinguish between *bound* and *free* occurrences. Intuitively, when you write a program, the bound occurrences of variables refer to variables that are internal to your program (e.g., global and local variables, function parameters, etc.). The free occurrences refer to external dependencies, i.e., the names of things that are not declared in your program but that you use in your program (e.g., the names of functions that are provided by external libraries).

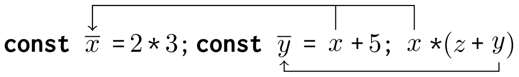
If you consistently rename all bound occurrences of a particular variable using a different (fresh) name, you do not change the behavior of your program. However, if you rename a free variable, you may change the program behavior (e.g., because you change the name of a library function that you are calling, so after renaming you now call a completely different function). In the rest of this section, we study these concepts in detail.

Defining occurrences of variables 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 has no associated defining occurrence is free. For example, consider again the expression from above:

$$\texttt{\textbf{const}}\ \overline{x} = 2 * 3;\ \texttt{\textbf{const}}\ \overline{y} = x + 5;\ x * (z + y)$$

The two using occurrences of x in this expression are bound because they are in the scope of the outer **const** declaration, which binds x. Similarly, the using occurrence of y is bound because it is in the scope of the inner **const** declaration, which binds y. The using occurrence of z is free because it is not in the scope of any **const** declaration that binds z.

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

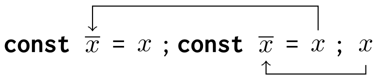

const $\overline{x}$ = 2 * 3; const $\overline{y}$ = x + 5; x *(z + y)

Every arrow points from a bound using occurrence of a variable to the associated defining occurrence. In the example, the using occurrence of z is the only free occurrence. Hence, this occurrence has no arrow to any binding occurrence.

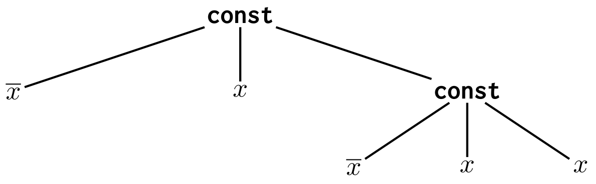
In general, a variable may be bound in more than one place. Variables may also occur both bound and free in the same expression. As an example, consider the following expression:

$$\texttt{\textbf{const}}\ x = x;\ \texttt{\textbf{const}}\ x = x;\ x$$

Here, the using occurrence of x in the defining expression of the outer **const** declaration is free since this occurrence is not in scope of any **const** declaration. The using occurrence of x in the defining expression of the inner **const** declaration is bound by the outer **const** declaration and the using occurrence in the body of the inner **const** declaration is bound by the inner **const** declaration:


const $\overline{x}$ = x ; const $\overline{x}$ = x ; x

If you have difficulties determining the binding structure of an expression, it is helpful to use the AST representation of the expression. For example, here is the AST for the expression **const** $x = x$; **const** $x = x$; x:



For simplicity, we omit the variant numbers in the AST representation. Note that for **const** nodes, the left child represents the variable defined by that **const** declaration, the middle child represents the defining expression, and the right child represents the body. We have already over-lined all the defining occurrences of variables in the AST.

For any using occurrence of a variable x in the AST, you can determine whether it is bound or free, respectively, find its associated defining occurrence as follows: Search for the node in the tree that represents the specific using occurrence of x you want to analyze. Start from that node and walk the tree upwards through the node's ancestors towards the root of the tree. If you visit a **const** node and you came from the right subtree representing the body of that **const** declaration, check whether the left child of that **const** node is labeled by x. If yes, the using occurrence of x that you are analyzing is a bound occurrence. If the current **const** node is the first node you have seen that defines x, you have found the associated defining occurrence of the using occurrence. If the name of the variable defined in the current **const** node is different from x, continue the upwards traversal to the root. If you reach the root without coming through the body of any **const** node that defines x, the using occurrence of x that you are analyzing is a 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 occuring in e, the set of all free variables occurring in e, respectively, the set of all bound variables occuring in e.

$$\begin{aligned}
ov &: \mathit{Expr} \to 2^{\mathit{Var}}\\
ov(n) &= \emptyset\\
ov(x) &= \{x\}\\
ov(e_1\, \mathit{bop}\, e_2) &= ov(e_1) \cup ov(e_2)\\
ov(\textbf{const}\ x = e_d\, ;\, e_b) &= \{x\} \cup ov(e_d) \cup ov(e_b)
\end{aligned}$$

$$\begin{aligned}
fv &: \mathit{Expr} \to 2^{\mathit{Var}}\\
fv(n) &= \emptyset\\
fv(x) &= \{x\}\\
fv(e_1\, \mathit{bop}\, e_2) &= fv(e_1) \cup fv(e_2)\\
fv(\textbf{const}\ x = e_d\, ;\, e_b) &= (fv(e_b) \setminus \{x\}) \cup fv(e_d)
\end{aligned}$$

$$\begin{aligned}
bv &: \mathit{Expr} \to 2^{\mathit{Var}}\\
bv(n) &= \emptyset\\
bv(x) &= \emptyset\\
bv(e_1\, \mathit{bop}\, e_2) &= bv(e_1) \cup bv(e_2)\\
bv(\textbf{const}\ x = e_d\, ;\, e_b) &= \{x\} \cup bv(e_d) \cup bv(e_b)
\end{aligned}$$

An expression e is called *closed* if is has no free variables, i.e., e satisfies $fv(e) = \emptyset$. Think of a closed expression as a program that has no external dependencies to symbols defined elsewhere (e.g. in an external library).

3.3.2 Evaluation with Static Binding

The notion of variable binding is strongly related to evaluation because it is during evaluation when a defining occurrence of a variable is bound to a specific value. We can make this formally precise by extending the *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 $env : Var \rightharpoonup Num$ and we denote by Env the set of all such environments.

Before we give the definition of the *eval* function, let us go through an example. Consider the expression

$$\texttt{const}\ x = x + 2;\ x * y$$

and the environment

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

Evaluation of a constant declaration proceeds as follows. First, we evaluate the defining expression $x + 2$ in the current environment env. In particular, the free occurrence of x in this expression is evaluated to $env(x) = 1$. 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 $x * y$ 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. For example, consider the following expression

$$\texttt{const}\ z = (\texttt{const}\ x = x + 2;\ x * y);\ z + x$$

which contains the expression from our earlier example as a subexpression. When we evaluate the outer **const** declaration in the environment env, we obtain 6 for the value of the defining expression of z. We then discard the intermediate environment that we generated during the evaluation of the defining expression of z, and evaluate the body $z + x$ of the outer **const** declaration using the environment:

$$env'' = env[z \mapsto 6] = \{x \mapsto 1, y \mapsto 2, z \mapsto 6\}$$

The result value that we obtain for the entire expression is thus 7.

We can formalize the new evaluation function *eval* using structural recursion as follows:

$$\begin{aligned}
&\mathit{eval} : \mathit{Env} \times \mathit{Expr} \to \mathit{Num}\\
\mathit{eval}(\mathit{env}, n) &= n\\
\mathit{eval}(\mathit{env}, x) &= \mathit{env}(x)\\
\mathit{eval}(\mathit{env}, e_1 \texttt{+} e_2) &= \mathit{eval}(\mathit{env}, e_1) + \mathit{eval}(\mathit{env}, e_2)\\
\mathit{eval}(\mathit{env}, e_1 \texttt{*} e_2) &= \mathit{eval}(\mathit{env}, e_1) \cdot \mathit{eval}(\mathit{env}, e_2)\\
\mathit{eval}(\mathit{env}, \textbf{\texttt{const}}\ x \texttt{=} e_d \texttt{;}\ e_b) &= \textbf{let}\ \mathit{env}' = \mathit{env}[x \mapsto \mathit{eval}(\mathit{env}, e_d)]\ \textbf{in}\\
&\ \mathit{eval}(\mathit{env}', e_b)
\end{aligned}$$

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

To summarize, the scope of a binding between a variable and its value determines the lifetime of that 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 language with 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.

3.3.3 Substitutions and Bindings

In Section 3.2.3, 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

$$e = \textbf{\texttt{const}}\ y \texttt{=} 2 \texttt{*} 3\texttt{;}\ y \texttt{+} x$$

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

$$e' = \textbf{\texttt{const}}\ y \texttt{=} 2 \texttt{*} 3\texttt{;}\ y \texttt{+} y$$

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 $\mathit{env} \in \mathit{Env}$, $x \in \mathit{Var}$, $e, e_s \in \mathit{Expr}$, if $\mathit{fv}(e) \cup \mathit{fv}(e_s) \subseteq \mathsf{dom}(\mathit{env})$, then

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

For the example above, the substitution property is violated because for the environment

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

we have

$$eval(env, e') = 12$$

whereas

$$\begin{array}{rl} & eval(env[x \mapsto eval(env, y)], e) \\ = & eval(env[x \mapsto 2], e) \\ = & eval(\{x \mapsto 2, y \mapsto 2\}, e) \\ = & 8 \end{array}$$

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:

$$subst : Expr \times Var \times Expr \to Expr$$

$$\begin{aligned} subst(n, x, e_s) &= n \\ subst(y, x, e_s) &= \textbf{if } x = y \textbf{ then } e_s \textbf{ else } y \\ subst(e_1\, bop\, e_2, x, e_s) &= subst(e_1, x, e_s)\, bop\, subst(e_2, x, e_s) \\ subst((\texttt{const } y = e_d\,;\, e_b), x, e_s) &= \textbf{let } e_b' = \textbf{if } x = y \textbf{ then } e_b \textbf{ else } subst(e_b, x, e_s) \textbf{ in} \\ & \texttt{const } y = subst(e_d, x, e_s)\,;\, e_b' \end{aligned}$$

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 $fv(e_s) \cap bv(e) = \emptyset$. 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., $fv(e_s) = \emptyset$.

For example, consider the expression

$$e_r = \texttt{const } z = 2 * 3;\ z + x$$

then we have

$$subst(e_r, x, y) = \texttt{const } z = 2 * 3;\ z + y$$

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

$$e = \texttt{const } y = 2 * 3;\ y + x$$

In particular, we have for all environments *env* with $x \in \mathsf{dom}(env)$:

$$eval(env, e) = eval(env, e_r)$$

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 $\mathit{bv}(e_r) \cap \mathit{fv}(e_s) = \emptyset$
- and then compute $\mathit{subst}(e_r, x, e_s)$.

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

$$\frac{e_b' = \mathit{subst}(e_b, x, y) \qquad y \notin \mathit{ov}(e_b) \qquad e_d \overset{\alpha}{\sim} e_d'}{\mathtt{const}\ x = e_d\,;\ e_b \overset{\alpha}{\sim} \mathtt{const}\ y = e_d'\,;\ e_b'} \qquad \frac{e_1 \overset{\alpha}{\sim} e_1' \qquad e_2 \overset{\alpha}{\sim} e_2'}{e_1\ \mathit{bop}\ e_2 \overset{\alpha}{\sim} e_1'\ \mathit{bop}\ e_2'}$$

$$\frac{e_d \overset{\alpha}{\sim} e_d' \qquad e_b \overset{\alpha}{\sim} e_b'}{\mathtt{const}\ x = e_d\,;\ e_b \overset{\alpha}{\sim} \mathtt{const}\ x = e_d'\,;\ e_b'} \qquad e \overset{\alpha}{\sim} e \qquad \frac{e' \overset{\alpha}{\sim} e}{e \overset{\alpha}{\sim} e'} \qquad \frac{e \overset{\alpha}{\sim} e'' \qquad e'' \overset{\alpha}{\sim} e'}{e \overset{\alpha}{\sim} e'}$$

With the definition of α-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*

$$s : \mathit{Expr} \times \mathit{Var} \times \mathit{Expr} \to \mathit{Expr}$$

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

$$s(e, x, e_s) \overset{\alpha}{\sim} \mathit{subst}(e_r, x, e_s)$$

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 α-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.

Summary. Think of a substitution $e[e_s/x]$ as a way of eliminating an external dependency to x in the expression e by replacing it with another expression e_s. For instance, if e is your program and x is the name of a library function that you are calling in e, you can eliminate the dependency of your program on x by replacing all free occurrences of x in e by the definition of x in the library.

When you compute a substitution $e[e_s/x]$, you want to preserve the result of the evaluation of e with respect to an environment where x is bound to the value obtained by evaluating e_s. This property is called the substitution property. There are two things to keep in mind when computing $e[e_s/x]$:

(1) only free occurrences of x in e are replaced by e_s

(2) if e_s has free variables, these variables should remain free in each context in e where a free using occurrence of x is replaced by e_s. To ensure this, you may have to rename some of the bound variables in e using fresh names so that they do not clash with the free variables of e_s.

Note that when you compute a substitution $e[e_s/x]$, you should *never* rename a free occurrence of a variable in e. For example, we have:

$$(y + x)[y/x] = (y + y)$$

In this example, you should not rename y by some other variable (e.g., to obtain $z+y$ as the result). Here, we have $e = (y+x)$ and y occurs free in e. So y should not be renamed. If you rename a free variable occurrence, you may change the evaluation result and violate the substitution property (e.g., because you change the name of an external library function that you are calling, so after renaming you now call a completely different function). On the other hand, we have

$$(\mathbf{const}\ y = 3; y + x)[y/x] = \mathbf{const}\ z = 3; z + y$$

Here, the bound occurrence of y must be renamed to avoid capturing of the free occurrence of y in the expression that is substituted in for x.

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 *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 expressed in a formal logic. 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, it is defined using structural recursion over the syntax of the language. One therefore also speaks of *structural operational semantics*, or *SOS* for short. 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 $e_1\, \texttt{\&\&}\, e_2$ and disjunctions $e_1\, \texttt{||}\, e_2$;
- comparison operators for equality $e_1\ \texttt{===}\ e_2$ and disequality $e_1\ \texttt{!==}\ e_2$; and
- conditional expressions $e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3$, which evaluate to e_2 if e_1 is true and e_3, otherwise.

The abstract syntax of our new language is as follows:

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

Note that we introduce a new syntactic class *Val* that consists of numbers and Booleans. We call the elements of *Val* *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 *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 to accommodate the fact that values include both numbers and Booleans:

$$\mathit{env} \in \mathit{Env} = \mathit{Var} \rightharpoonup \mathit{Val}$$

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 will view computation as rewriting of syntactic expressions.

The definition of the big-step SOS will look similar to the *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 *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 do not determine the order in which expressions are evaluated. This 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

$$\_ \vdash \_ \Downarrow \_ \subseteq \mathit{Env} \times \mathit{Expr} \times \mathit{Val}$$

A *judgment form* $\mathit{env} \vdash e \Downarrow v$ states that under value environment 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:

$$\begin{array}{c}\textsc{EvalVal}\\ \mathit{env} \vdash v \Downarrow v\end{array}$$

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

$$\textsc{EvalVar}\ \frac{x \in \mathsf{dom}(\mathit{env})}{\mathit{env} \vdash x \Downarrow \mathit{env}(x)}$$

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 + 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, including Scala, provide typing rules that strictly forbid expressions such as `1 + 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 *Bool* to values of type *Num* and vice versa:

$$\begin{aligned}
\mathit{toNum} &: \mathit{Val} \to \mathit{Num}\\
\mathit{toNum}(n) &= n\\
\mathit{toNum}(\texttt{true}) &= 1\\
\mathit{toNum}(\texttt{false}) &= 0\\
\mathit{toBool} &: \mathit{Val} \to \mathit{Bool}\\
\mathit{toBool}(b) &= b\\
\mathit{toBool}(n) &= \textbf{if } n = 0 \textbf{ then } \texttt{false} \textbf{ else } \texttt{true}
\end{aligned}$$

These two functions are consistent with the type coercions between numbers and Booleans performed by JavaScript.

We can now define the semantics of the operator `+` by the following rule:

$$\frac{\textsc{EvalPlus}\quad \mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad v = \mathit{toNum}(v_1) + \mathit{toNum}(v_2)}{\mathit{env} \vdash e_1 \,\texttt{+}\, e_2 \Downarrow v}$$

Note how the coercion function *toNum* is used to convert the intermediate values v_1 and v_2 to numbers so that the result of the addition operation is well-defined. The semantics of the operator `*` is defined accordingly by a 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 premise $\mathit{env} \vdash e_1 \Downarrow v_1$ comes before the premise $\mathit{env} \vdash e_2 \Downarrow v_2$ 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 the evaluation of e_1 and e_2 has 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, which we will discuss in Section 4.2.

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

$$\frac{\textsc{EvalConstDecl}\quad \mathit{env} \vdash e_d \Downarrow v_d \qquad \mathit{env}' = \mathit{env}[x \mapsto v_d] \qquad \mathit{env}' \vdash e_b \Downarrow v_b}{\mathit{env} \vdash \textbf{const}\ x \,\texttt{=}\, e_d; e_b \Downarrow v_b}$$

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

$$\frac{\textsc{EvalEqual}\quad \mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad b = (v_1 = v_2)}{\mathit{env} \vdash e_1 \;\texttt{===}\; e_2 \Downarrow b}$$

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` (after coercion) and one for the case where it evaluates to `false`:

$$\textsc{EvalIfThen}\quad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{toBool}(v_1) = \texttt{true} \qquad \mathit{env} \vdash e_2 \Downarrow v_2}{\mathit{env} \vdash e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \Downarrow v_2}$$

$$\textsc{EvalIfElse}\quad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{toBool}(v_1) = \texttt{false} \qquad \mathit{env} \vdash e_3 \Downarrow v_3}{\mathit{env} \vdash e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 \Downarrow v_3}$$

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, ∧, and disjunction, ∨, on Booleans as follows:

$$\begin{aligned}
\wedge &: \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}\\
\texttt{true} \wedge b &= b\\
\texttt{false} \wedge b &= \texttt{false}\\
\vee &: \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}\\
\texttt{true} \vee b &= \texttt{true}\\
\texttt{false} \vee b &= b
\end{aligned}$$

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

$$\textsc{EvalAnd}\quad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad b_1 = \mathit{toBool}(v_1) \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad b_2 = \mathit{toBool}(v_2)}{\mathit{env} \vdash e_1 \,\texttt{\&\&}\, e_2 \Downarrow b_1 \wedge b_2}$$

The semantics of `||` is given in terms of ∨ 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 capture the actual implementation of these operators in JavaScript and most other programming languages. Boolean operators are typically implemented more efficiently. Observe that the operator ∧ has an interesting property: the term $\texttt{false} \wedge b_2$ is `false` no matter whether b_2 is `true` or `false`. This allows us to optimize the evaluation of e_1 `&&` e_2 as follows. If e_1 evaluates to `false`, then there is no point in continuing evaluation of e_2. We can immediately return the result of evaluating e_1 as the result of the entire

$$\textsc{EvalVal}\qquad \mathit{env} \vdash v \Downarrow v$$

$$\textsc{EvalVar}\qquad \frac{x \in \mathsf{dom}(\mathit{env})}{\mathit{env} \vdash x \Downarrow \mathit{env}(x)}$$

$$\textsc{EvalPlus}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad v = \mathit{toNum}(v_1) + \mathit{toNum}(v_2)}{\mathit{env} \vdash e_1 \texttt{+} e_2 \Downarrow v}$$

$$\textsc{EvalTimes}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad v = \mathit{toNum}(v_1) \cdot \mathit{toNum}(v_2)}{\mathit{env} \vdash e_1 \texttt{*} e_2 \Downarrow v}$$

$$\textsc{EvalConstDecl}\qquad \frac{\mathit{env} \vdash e_d \Downarrow v_d \qquad \mathit{env}' = \mathit{env}[x \mapsto v_d] \qquad \mathit{env}' \vdash e_b \Downarrow v_b}{\mathit{env} \vdash \mathbf{const}\ x \texttt{=} e_d; e_b \Downarrow v_b}$$

$$\textsc{EvalEqual}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad b = (v_1 = v_2)}{\mathit{env} \vdash e_1\ \texttt{===}\ e_2 \Downarrow b}$$

$$\textsc{EvalDisequal}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad b = (v_1 \neq v_2)}{\mathit{env} \vdash e_1\ \texttt{!==}\ e_2 \Downarrow b}$$

$$\textsc{EvalIfThen}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{toBool}(v_1) = \texttt{true} \qquad \mathit{env} \vdash e_2 \Downarrow v_2}{\mathit{env} \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow v_2}$$

$$\textsc{EvalIfElse}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{toBool}(v_1) = \texttt{false} \qquad \mathit{env} \vdash e_3 \Downarrow v_3}{\mathit{env} \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow v_3}$$

$$\textsc{EvalAndFalse}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \texttt{false} = \mathit{toBool}(v_1)}{\mathit{env} \vdash e_1 \texttt{\&\&} e_2 \Downarrow v_1}$$

$$\textsc{EvalAndTrue}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \texttt{true} = \mathit{toBool}(v_1) \qquad \mathit{env} \vdash e_2 \Downarrow v_2}{\mathit{env} \vdash e_1 \texttt{\&\&} e_2 \Downarrow v_2}$$

$$\textsc{EvalOrTrue}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \texttt{true} = \mathit{toBool}(v_1)}{\mathit{env} \vdash e_1\ \texttt{||}\ e_2 \Downarrow v_1}$$

$$\textsc{EvalOrFalse}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \texttt{false} = \mathit{toBool}(v_1) \qquad \mathit{env} \vdash e_2 \Downarrow v_2}{\mathit{env} \vdash e_1\ \texttt{||}\ e_2 \Downarrow v_2}$$

Figure 4.1: Inference rules that define the big-step SOS of our expression language

expression. Only if e_1 evaluates to `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 $e_1 \mathbin{\texttt{\&\&}} e_2$. The short-circuit semantics ensures that e_2 will never be evaluated, unless e_1 is `true`.

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

$$\textsc{EvalAndFalse}\quad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \texttt{false} = \mathit{toBool}(v_1)}{\mathit{env} \vdash e_1 \mathbin{\texttt{\&\&}} e_2 \Downarrow v_1}$$

One interesting detail of this rule is that the result of the evaluation is not `false` but v_1, which gives `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 `true` because $0 \mathbin{\texttt{\&\&}} \texttt{true}$ evaluates to 0:

$$0 \mathbin{\texttt{===}} (0 \mathbin{\texttt{\&\&}} \texttt{true})$$

This might be counter-intuitive but the EvalAndFalse rule faithfully reflects the actual semantics of the `&&` operator in JavaScript.

The case where e_1 evaluates to `true` is covered by the following rule:

$$\textsc{EvalAndTrue}\quad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \texttt{true} = \mathit{toBool}(v_1) \qquad \mathit{env} \vdash e_2 \Downarrow v_2}{\mathit{env} \vdash e_1 \mathbin{\texttt{\&\&}} e_2 \Downarrow v_2}$$

Similar to the rule EvalAndFalse, the result of the evaluation of $e_1 \mathbin{\texttt{\&\&}} e_2$ in this case is the uncoerced result value of the subexpression e_2, which may not be a Boolean.

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 $\mathit{env} \vdash e \Downarrow v$ 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:

$$\textsc{EvalPlus}\ \frac{\textsc{EvalVar}\ \dfrac{x \in \mathsf{dom}(\{x \mapsto 2\})}{\{x \mapsto 2\} \vdash x \Downarrow 2} \qquad \dfrac{\textsc{EvalVal}}{\{x \mapsto 2\} \vdash 3 \Downarrow 3} \qquad 5 = 3 + 2}{\{x \mapsto 2\} \vdash x + 3 \Downarrow 5}$$

We can also build the derivations bottom-up, starting with some concrete judgment form $\mathit{env} \vdash e \Downarrow v$. 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, effectively reconstructing the *eval* function from the inference rules. We explain this through an example.

Suppose we are given the environment

$$\mathit{env} = \{x \mapsto 2\}$$

and we want to evaluate the expression

$$e = x \mathbin{?} 5 \mathbin{:} \texttt{false}$$

Then we can compute the result of the evaluation by filling in the red question marks in the following partial derivation where the ? indicate missing pieces of the derivation that we still need to fill in:

$$\textsc{Eval}?\ \dfrac{?}{\{x \mapsto 2\} \vdash x \mathbin{?} 5 \mathbin{:} \texttt{false} \Downarrow ?}$$

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. Although, we do not yet know which of the two rules we need to use, they 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:

$$\textsc{EvalIf}?\ \dfrac{\textsc{Eval}?\ \dfrac{?}{\{x \mapsto 2\} \vdash x \Downarrow ?} \qquad ? = \mathit{toBool}(?) \qquad \textsc{Eval}?\ \dfrac{?}{\{x \mapsto 2\} \vdash ? \Downarrow ?}}{\{x \mapsto 2\} \vdash x \mathbin{?} 5 \mathbin{:} \texttt{false} \Downarrow ?}$$

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:

$$\textsc{EvalIf}?\ \dfrac{\textsc{EvalVar}\ \dfrac{x \in \mathsf{dom}(\{x \mapsto 2\})}{\{x \mapsto 2\} \vdash x \Downarrow 2} \qquad ? = \mathit{toBool}(?) \qquad \textsc{Eval}?\ \dfrac{?}{\{x \mapsto 2\} \vdash ? \Downarrow ?}}{\{x \mapsto 2\} \vdash x \mathbin{?} 5 \mathbin{:} \texttt{false} \Downarrow ?}$$

Now, we have obtained 2 as the result of evaluating x in *env*. Applying the function *toBool* to 2 yields `true`. Thus we must use the rule EvalIfThen, which allows us to fill more of the remaining holes in the derivation:

$$\textsc{EvalIfThen}\ \dfrac{\dfrac{\textsc{EvalVar} \atop x \in \mathsf{dom}(\{x \mapsto 2\})}{\{x \mapsto 2\} \vdash x \Downarrow 2} \qquad \mathtt{true} = \mathit{toBool}(2) \qquad \dfrac{\textsc{Eval}? \atop ?}{\{x \mapsto 2\} \vdash 5 \Downarrow ?}}{\{x \mapsto 2\} \vdash x\ \mathtt{?}\ 5\ \mathtt{:}\ \mathtt{false} \Downarrow ?}$$

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:

$$\textsc{EvalIfThen}\ \dfrac{\dfrac{\textsc{EvalVar} \atop x \in \mathsf{dom}(\{x \mapsto 2\})}{\{x \mapsto 2\} \vdash x \Downarrow 2} \qquad \mathtt{true} = \mathit{toBool}(2) \qquad {\textsc{EvalVal} \atop \{x \mapsto 2\} \vdash 5 \Downarrow 5}}{\{x \mapsto 2\} \vdash x\ \mathtt{?}\ 5\ \mathtt{:}\ \mathtt{false} \Downarrow 5}$$

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 does faithfully model all the details of a programming language's semantics. In particular, the inductive definition of the big-step SOS given in the previous section only captures computations that can be described by finite derivation trees constructed from the inference rules. That is, it only captures computations that eventually terminate. The behavior of infinite computations is left undefined. This is not an issue for the simple language that we have considered so far because the evaluation of any expression in this language is guaranteed to terminate. However, this will no longer be true once we extend the language with functions.

Another aspect that is not captured by our big-step semantics is the order in which expressions are evaluated. In a programming language where the evaluation of expressions may have side effects, the evaluation order matters. If a language fixes a specific evaluation order, then this should be reflected in the operational semantics. For example, in JavaScript all binary operators are evaluated left to right.

One way to capture these finer details of a language's semantics is to switch to a *small-step* SOS. This type of SOS is more fine-grained and can model execution order and non-terminating computations. It is often the preferred style of semantics when one is interested in proving properties about a programming language as a whole.

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

$$(2+8)*3$$

will be simplified as follows:

$$(\underline{2+8})*3 \rightarrow \underline{10*3} \rightarrow 30$$

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:

- *"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 $10*3$ 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 $(2+8)*3$ and performed the do rule for that subexpression, yielding $10*3$.

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

$$\_ \vdash \_ \rightarrow \_ \subseteq \mathit{Env} \times \mathit{Expr} \times \mathit{Expr}$$

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 that capture the intermediate results of the computation. A judgment form $\mathit{env} \vdash e \rightarrow e'$ 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 $\mathit{env} \vdash e \rightarrow e'$, then e must be a value. That is, whenever the small-step evaluation terminates, we obtain a value.

Our first do rule concerns variables. The rule is almost identical to the EvalVar rule:

$$\textsc{DoVar}\quad \frac{x \in \mathsf{dom}(\mathit{env})}{\mathit{env} \vdash x \rightarrow \mathit{env}(x)}$$

$$\textsc{DoVar}\quad \frac{x \in \mathsf{dom}(\mathit{env})}{\mathit{env} \vdash x \rightarrow \mathit{env}(x)}$$

$$\textsc{DoPlus}\quad \frac{v = \mathit{toNum}(v_1) + \mathit{toNum}(v_2)}{\mathit{env} \vdash v_1 \,\texttt{+}\, v_2 \rightarrow v}$$

$$\textsc{DoTimes}\quad \frac{v = \mathit{toNum}(v_1) \cdot \mathit{toNum}(v_2)}{\mathit{env} \vdash v_1 \,\texttt{*}\, v_2 \rightarrow v}$$

$$\textsc{DoEqual}\quad \frac{b = (v_1 = v_2)}{\mathit{env} \vdash v_1 \,\texttt{===}\, v_2 \rightarrow b}$$

$$\textsc{DoDisequal}\quad \frac{b = (v_1 \neq v_2)}{\mathit{env} \vdash v_1 \,\texttt{!==}\, v_2 \rightarrow b}$$

$$\textsc{DoConstDecl}\quad \mathit{env} \vdash \textbf{const}\ x \,\texttt{=}\, v_d; v_b \rightarrow v_b$$

$$\textsc{DoIfThen}\quad \frac{\mathit{toBool}(v_1) = \texttt{true}}{\mathit{env} \vdash v_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 \rightarrow e_2}$$

$$\textsc{DoIfElse}\quad \frac{\mathit{toBool}(v_1) = \texttt{false}}{\mathit{env} \vdash v_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 \rightarrow e_3}$$

$$\textsc{DoAndFalse}\quad \frac{\texttt{false} = \mathit{toBool}(v_1)}{\mathit{env} \vdash v_1 \,\texttt{\&\&}\, e_2 \rightarrow v_1}$$

$$\textsc{DoAndTrue}\quad \frac{\texttt{true} = \mathit{toBool}(v_1)}{\mathit{env} \vdash v_1 \,\texttt{\&\&}\, e_2 \rightarrow e_2}$$

$$\textsc{DoOrTrue}\quad \frac{\texttt{true} = \mathit{toBool}(v_1)}{\mathit{env} \vdash v_1 \,\texttt{||}\, e_2 \rightarrow v_1}$$

$$\textsc{DoOrFalse}\quad \frac{\texttt{false} = \mathit{toBool}(v_1)}{\mathit{env} \vdash v_1 \,\texttt{||}\, e_2 \rightarrow e_2}$$

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

$$\textsc{SearchBop1}\quad \frac{\mathit{env} \vdash e_1 \rightarrow e_1'}{\mathit{env} \vdash e_1 \mathbin{\mathit{bop}} e_2 \rightarrow e_1' \mathbin{\mathit{bop}} e_2}$$

$$\textsc{SearchBop2}\quad \frac{\mathit{bop} \notin \{\texttt{\&\&}, \texttt{||}\} \qquad \mathit{env} \vdash e_2 \rightarrow e_2'}{\mathit{env} \vdash v_1 \mathbin{\mathit{bop}} e_2 \rightarrow v_1 \mathbin{\mathit{bop}} e_2'}$$

$$\textsc{SearchConstDecl1}\quad \frac{\mathit{env} \vdash e_d \rightarrow e_d'}{\mathit{env} \vdash \textbf{const}\ x \,\texttt{=}\, e_d; e_b \rightarrow \textbf{const}\ x \,\texttt{=}\, e_d'; e_b}$$

$$\textsc{SearchConstDecl2}\quad \frac{\mathit{env}' = \mathit{env}[x \mapsto v_d] \qquad \mathit{env}' \vdash e_b \rightarrow e_b'}{\mathit{env} \vdash \textbf{const}\ x \,\texttt{=}\, v_d; e_b \rightarrow \textbf{const}\ x \,\texttt{=}\, v_d; e_b'}$$

$$\textsc{SearchIf}\quad \frac{\mathit{env} \vdash e_1 \rightarrow e_1'}{\mathit{env} \vdash e_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 \rightarrow e_1' \ \texttt{?}\ e_2 \ \texttt{:}\ e_3}$$

Figure 4.3: Search rules for the small-step SOS of our expression language

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 +:

$$\textsc{DoPlus}\quad \frac{v = \mathit{toNum}(v_1) + \mathit{toNum}(v_2)}{\mathit{env} \vdash v_1 + v_2 \to v}$$

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 $e_1 \notin \mathit{Val}$ or $e_2 \notin \mathit{Val}$ 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:

$$\textsc{DoPlus}\quad \frac{e_1, e_2 \in \mathit{Val} \quad v = \mathit{toNum}(e_1) + \mathit{toNum}(e_2)}{\mathit{env} \vdash e_1 + e_2 \to v}$$

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

$$\textsc{SearchPlus1}\quad \frac{\mathit{env} \vdash e_1 \to e_1'}{\mathit{env} \vdash e_1 + e_2 \to e_1' + e_2}$$

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, $\mathit{env} \vdash e_1 \to e_1'$, 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:

$$\textsc{SearchPlus2}\quad \frac{\mathit{env} \vdash e_2 \to e_2'}{\mathit{env} \vdash v_1 + e_2 \to v_1 + e_2'}$$

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 SearchPlus1 iteratively reduces e_1 to a value v_1, obtaining the expression $v_1 + e_2$. Then the rule SearchPlus2 iteratively reduces e_2 to a value v_2, obtaining the expression $v_1 + v_2$. Finally, the rule 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 `*` is captured by an according do rule called DoTimes and search rules that are almost identical to the rules SearchPlus1 and SearchPlus2. In fact, all binary operators except the short-circuiting ones are evaluated left-to-right. Hence, we can generalize the search rules SearchPlus1 and SearchPlus2 so that they also handle all these other operators:

SearchBop1
$$\frac{env \vdash e_1 \to e_1'}{env \vdash e_1\, bop\, e_2 \to e_1'\, bop\, e_2}$$

SearchBop2
$$\frac{bop \notin \{\texttt{\&\&}, \texttt{||}\} \qquad env \vdash e_2 \to e_2'}{env \vdash v_1\, bop\, e_2 \to v_1\, bop\, e_2'}$$

Note that the rule 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:

DoAndTrue
$$\frac{\texttt{true} = toBool(v_1)}{env \vdash v_1 \,\texttt{\&\&}\, e_2 \to e_2}$$

DoOrFalse
$$\frac{\texttt{false} = toBool(v_1)}{env \vdash v_1 \,\texttt{||}\, e_2 \to e_2}$$

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 **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:

SearchConstDecl1
$$\frac{env \vdash e_d \to e_d'}{env \vdash \textbf{const}\ x = e_d; e_b \to \textbf{const}\ x = e_d'; e_b}$$

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

SearchConstDecl2
$$\frac{env' = env[x \mapsto v_d] \qquad env' \vdash e_b \to e_b'}{env \vdash \textbf{const}\ x = v_d; e_b \to \textbf{const}\ x = v_d; e_b'}$$

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:

DoConstDecl
$$env \vdash \textbf{const}\ x = v_d; v_b \to v_b$$

The rules for conditional expressions e_1 `?` e_2 `:` e_3 are again straightforward.

Chapter 5

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 concrete 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 *unbounded* computations where the number of computation steps to be executed depends on the input values.

You will often hear 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* from now on.

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:

$$x \texttt{ => } e$$

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. Note that the function expression binds the parameter variable x. The scope of this binding is the body e.

In order to be able to use functions, we additionally introduce *function call* expressions, e_1`(`e_2`)`, where e_1 must evaluate to a function expression and e_2 evaluates to the argument that is passed to that function. We will use the term *function application* synonymously to function call.

Intuitively, a call expression (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 *plusTwo*:

$$\begin{aligned} \mathit{plusTwo} &: \mathit{Num} \to \mathit{Num} \\ \mathit{plusTwo}(x) &= x + 2 \end{aligned}$$

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

```
const plusTwo = x => x + 2;
plusTwo(3)
```

Note that we consider the “arrow” operator for constructing anonymous functions to have the lowest precedence of all operators. In particular, `x => x + 2` should be read as `x => (x + 2)` and not `(x => x) + 2`.

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 *lambda calculus*. If you are interested in learning more about this result, you can read Section 5.6. Due to the connection to the lambda calculus, function expressions are also often referred to as *lambda abstractions*.

For the convenience of writing recursive functions succinctly, we will also add named function expressions to our language. These take the form

`function` x_1`(`x_2`)` e

where x_1 refers to the name of the function, x_2 is the formal parameter, and e is the body. The scope of both x_1 and x_2 is restricted to e.

Here is how we write a program in our extended language that computes the factorial of `5` using a recursive definition of the factorial function:

```
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.

In our treatment of recursive functions **`function`** x_1`(`x_2`)` e we deviate slightly from JavaScript syntax. First, for the sake of simplicity, we restrict the scope of x_1 to the function body e. This explains why in the example above, 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 the function name available for subsequent calls in the program (i.e., the body of the **`const`** declaration). Second, we allow the body of a recursive function to be an arbitrary expressions whose value determines the return value of the function, just like for anonymous functions. In actual JavaScript syntax, the body of a recursive function is a statement and the function's result value is determined by a **`return`** statement in the body.

Here is how the factorial example would look like in proper JavaScript syntax:

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

Note that in JavaScript, we do not need to bind the recursive function again to its name using a `const` declaration in order to make it callable by its name outside of its body. Instead, the scope of the name of a recursive function includes both the body of the function and the remainder of the statement block where the function is declared.

We will sometimes write examples in actual JavaScript syntax so that you can easily play with the code using a JavaScript interpreter.

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

$$
\begin{array}{rcll}
x \in \mathit{Var} & & & \text{variables}\\
n \in \mathit{Num} & & & \text{numbers}\\
b \in \mathit{Bool} & ::= & \texttt{true} \mid \texttt{false} & \text{Booleans}\\
v \in \mathit{Val} & ::= & n \mid b \mid \textbf{function}\ p(x)\ e \mid \texttt{typeerror} & \text{values}\\
e \in \mathit{Expr} & ::= & v \mid x \mid e_1\ \mathit{bop}\ e_2 \mid \textbf{const}\ x = e_d;\ e_b & \text{expressions}\\
 & & \mid e_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 \mid e_1(e_2) & \\
\mathit{bop} \in \mathit{Bop} & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{===} \mid \texttt{!==} & \text{binary operators}\\
p & ::= & x \mid \epsilon & \text{function names}
\end{array}
$$

Note that for the uniformity of representation, we write **`function`**`(`x`)` e for x `=>` e in our abstract syntax representation. This corresponds to the case **`function`** $p(x)\ e$ in the grammar when the function name p is ϵ. That is, we write anonymous functions like recursive functions and just omit the function name. Though, we will stick to the more compact concrete syntax x `=>` e when using anonymous functions in examples.

Further 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 **`function`**$(p)\ xe$ 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 *plus* that takes two numbers and adds them:

$$\begin{aligned} \mathit{plus} &: \mathit{Num} \times \mathit{Num} \to \mathit{Num} \\ \mathit{plus}(x, y) &= x + y \end{aligned}$$

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

$$\begin{aligned} \mathit{plus} &: \mathit{Num} \to (\mathit{Num} \to \mathit{Num}) \\ \mathit{plus}(x)(y) &= x + y \end{aligned}$$

Observe that *plus* is now a function that takes a number x and returns another function $\mathit{plus}(x) : \mathit{Num} \to \mathit{Num}$. This returned function takes a number y and returns the number $x + y$. The transformation of functions that take more than one parameter into cascaded functions that each take one parameter is referred to as *currying*[1]. We also call the second version of *plus* a *curried function*. The grammar of our language allows the body e of a function expression **`function`**$(x)\ e$ to be itself a function expression. Thus, we can express the curried version of *plus* in our language as follows:

```
const plus = x => y => x + y;
plus(2)(3)
```

This program will evaluate to `5` as expected. Note that the arrow

One useful feature of curried functions is that we can apply them partially. For example, we can express the function *plusTwo* that adds the value 2 to its argument using the function *plus* as follows:

[1] The term *currying* is derived from the name of the logician Haskell Curry, who popularized this technique.

```
const plus = x => y => x + y;
const plusTwo = plus(2);
plusTwo(3) + 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 $\mathtt{\textbf{function}}\ p(x)\ e$ should not be further evaluated. In fact, from our old rule EvalVal for evaluating values, we immediately obtain

$$\mathit{env} \vdash \mathtt{\textbf{function}}\ p(x)\ e \Downarrow \mathtt{\textbf{function}}\ p(x)\ e$$

For a function expression 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:

EvalCall

$$\frac{\begin{array}{c}\mathit{env} \vdash e_1 \Downarrow v_1 \\ v_1 = \mathtt{\textbf{function}}(x)\ e \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad \mathit{env}' = \mathit{env}[x \mapsto v_2] \qquad \mathit{env}' \vdash e \Downarrow v\end{array}}{\mathit{env} \vdash e_1(e_2) \Downarrow v}$$

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 $\mathtt{\textbf{function}}(x)\ e$ 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 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 in contrast 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:

$$\textsc{EvalCallRec}\quad \frac{\begin{array}{c} env \vdash e_1 \Downarrow v_1 \qquad v_1 = \texttt{function}\; x_1(x_2)\; e \\ env \vdash e_2 \Downarrow v_2 \qquad env' = env[x_1 \mapsto v_1][x_2 \mapsto v_2] \qquad env' \vdash e \Downarrow v \end{array}}{env \vdash e_1(e_2) \Downarrow v}$$

In addition to binding the parameter x_2 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" $\mathsf{typeerror}$:

$$\textsc{EvalTypeErrorCall}\quad \frac{env \vdash e_1 \Downarrow v_1 \qquad v_1 \neq \texttt{function}\; p(x)\; e}{env \vdash e_1(e_2) \Downarrow \mathsf{typeerror}}$$

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:

$$\textsc{EvalPropConst}\quad \frac{env \vdash e_d \Downarrow \mathsf{typeerror}}{env \vdash \texttt{const}\; x = e_d\,;\; e_b \Downarrow \mathsf{typeerror}}$$

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 EvalEqual and 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 $\textsc{EvalTypeErrorEqual}_1$ and $\textsc{EvalTypeErrorEqual}_2$ 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 **const** declarations **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 determine which value each variable refers to. Unfortunately, with the big-step semantics for function calls that we have given above, we no longer correctly implement static binding.

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

```
1 const x = 2;
2 const plusTwo = y => x + y;
3 const f = 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 $\textsc{EvalCall}$. 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` in the environment. 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 $[\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 3]$, yielding 6 as the overall result of the evaluation.

$$\textsc{EvalVal}\qquad \mathit{env} \vdash v \Downarrow v$$

$$\textsc{EvalVar}\qquad \frac{x \in \mathsf{dom}(\mathit{env})}{\mathit{env} \vdash x \Downarrow \mathit{env}(x)}$$

$$\textsc{EvalPlus}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad v = \mathit{toNum}(v_1) + \mathit{toNum}(v_2)}{\mathit{env} \vdash e_1 \,\texttt{+}\, e_2 \Downarrow v}$$

$$\textsc{EvalTimes}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad v = \mathit{toNum}(v_1) \cdot \mathit{toNum}(v_2)}{\mathit{env} \vdash e_1 \,\texttt{*}\, e_2 \Downarrow v}$$

$$\textsc{EvalConstDecl}\qquad \frac{\mathit{env} \vdash e_d \Downarrow v_d \qquad \mathit{env}' = \mathit{env}[x \mapsto v_d] \qquad \mathit{env}' \vdash e_b \Downarrow v_b}{\mathit{env} \vdash \textbf{const}\ x \,\texttt{=}\, e_d\texttt{;}\, e_b \Downarrow v_b}$$

$$\textsc{EvalEqual}\qquad \frac{\begin{array}{c} \mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \\ b = (v_1 = v_2) \qquad v_1 \neq \texttt{function}\ p_1(x_1)\ e_3 \qquad v_2 \neq \texttt{function}\ p_2(x_2)\ e_4 \end{array}}{\mathit{env} \vdash e_1 \ \texttt{===}\ e_2 \Downarrow b}$$

$$\textsc{EvalDisequal}\qquad \frac{\begin{array}{c} \mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \\ b = (v_1 \neq v_2) \qquad v_1 \neq \texttt{function}\ p_1(x_1)\ e_3 \qquad v_2 \neq \texttt{function}\ p_2(x_2)\ e_4 \end{array}}{\mathit{env} \vdash e_1 \ \texttt{!==}\ e_2 \Downarrow b}$$

$$\textsc{EvalIfThen}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{toBool}(v_1) = \mathsf{true} \qquad \mathit{env} \vdash e_2 \Downarrow v_2}{\mathit{env} \vdash e_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 \Downarrow v_2}$$

$$\textsc{EvalIfElse}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathit{toBool}(v_1) = \mathsf{false} \qquad \mathit{env} \vdash e_3 \Downarrow v_3}{\mathit{env} \vdash e_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 \Downarrow v_3}$$

$$\textsc{EvalAndFalse}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathsf{false} = \mathit{toBool}(v_1)}{\mathit{env} \vdash e_1 \,\texttt{\&\&}\, e_2 \Downarrow v_1}$$

$$\textsc{EvalAndTrue}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathsf{true} = \mathit{toBool}(v_1) \qquad \mathit{env} \vdash e_2 \Downarrow v_2}{\mathit{env} \vdash e_1 \,\texttt{\&\&}\, e_2 \Downarrow v_2}$$

$$\textsc{EvalOrTrue}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathsf{true} = \mathit{toBool}(v_1)}{\mathit{env} \vdash e_1 \,\texttt{||}\, e_2 \Downarrow v_1}$$

$$\textsc{EvalOrFalse}\qquad \frac{\mathit{env} \vdash e_1 \Downarrow v_1 \qquad \mathsf{false} = \mathit{toBool}(v_1) \qquad \mathit{env} \vdash e_2 \Downarrow v_2}{\mathit{env} \vdash e_1 \,\texttt{||}\, e_2 \Downarrow v_2}$$

$$\textsc{EvalCall}\qquad \frac{\begin{array}{c} \mathit{env} \vdash e_1 \Downarrow v_1 \\ v_1 = \texttt{function}(x)\ e \qquad \mathit{env} \vdash e_2 \Downarrow v_2 \qquad \mathit{env}' = \mathit{env}[x \mapsto v_2] \qquad \mathit{env}' \vdash e \Downarrow v \end{array}}{\mathit{env} \vdash e_1(e_2) \Downarrow v}$$

$$\textsc{EvalCallRec}\qquad \frac{\begin{array}{c} \mathit{env} \vdash e_1 \Downarrow v_1 \qquad v_1 = \texttt{function}\ x_1(x_2)\ e \\ \mathit{env} \vdash e_2 \Downarrow v_2 \qquad \mathit{env}' = \mathit{env}[x_1 \mapsto v_1][x_2 \mapsto v_2] \qquad \mathit{env}' \vdash e \Downarrow v \end{array}}{\mathit{env} \vdash e_1(e_2) \Downarrow v}$$

$$\textsc{EvalTypeErrorEqual}_1\quad \frac{env \vdash e_1 \Downarrow \textbf{function}\ p(x)\ e \qquad bop \in \{\texttt{===}, \texttt{!==}\}}{env \vdash e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalTypeErrorEqual}_2\quad \frac{env \vdash e_2 \Downarrow \textbf{function}\ p(x)\ e \qquad bop \in \{\texttt{===}, \texttt{!==}\}}{env \vdash e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalTypeErrorCall}\quad \frac{env \vdash e_1 \Downarrow v_1 \qquad v_1 \neq \textbf{function}\ p(x)\ e}{env \vdash e_1(e_2) \Downarrow \mathsf{typeerror}} \qquad \textsc{EvalPropIf}\quad \frac{env \vdash e_1 \Downarrow \mathsf{typeerror}}{env \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalPropBop}_1\quad \frac{env \vdash e_1 \Downarrow \mathsf{typeerror}}{env \vdash e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}} \qquad \textsc{EvalPropBop}_2\quad \frac{env \vdash e_2 \Downarrow \mathsf{typeerror} \quad bop \notin \{\texttt{\&\&}, \texttt{||}\}}{env \vdash e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalPropAnd}\quad \frac{env \vdash e_1 \Downarrow v_1 \qquad v_1 \neq \mathsf{typeerror} \qquad \texttt{true} = \mathit{toBool}(v_1) \qquad env \vdash e_2 \Downarrow \mathsf{typeerror}}{env \vdash e_1\ \texttt{\&\&}\ e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalPropOr}\quad \frac{env \vdash e_1 \Downarrow v_1 \qquad v_1 \neq \mathsf{typeerror} \qquad \texttt{false} = \mathit{toBool}(v_1) \qquad env \vdash e_2 \Downarrow \mathsf{typeerror}}{env \vdash e_1\ \texttt{||}\ e_2 \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalPropCall}_1\quad \frac{env \vdash e_1 \Downarrow \mathsf{typeerror}}{env \vdash e_1(e_2) \Downarrow \mathsf{typeerror}} \qquad \textsc{EvalPropCall}_2\quad \frac{env \vdash e_2 \Downarrow \mathsf{typeerror}}{env \vdash e_1(e_2) \Downarrow \mathsf{typeerror}}$$

$$\textsc{EvalPropConstDecl}\quad \frac{env \vdash e_d \Downarrow \mathsf{typeerror}}{env \vdash \textbf{const}\ x = e_d\,;\ e_b \Downarrow \mathsf{typeerror}}$$

Figure 5.2: Big-step operational semantics: dynamic type error rules

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 to reestablish 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: $fv(e) = \emptyset$. 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 free variables as soon as they appear in subexpressions 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:

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

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

```
const x = 2;
const plusTwo = y => x + y;
const f = x => plusTwo(x);
f(3)
->
const plusTwo = y => 2 + y;
const f = x => plusTwo(x);
f(3)
->
const f = x => (y => 2 + y)(x);
f(3)
-> (x => (y => 2 + y)(x))(3)
-> (y => 2 + y)(3)
-> 2 + 3
-> 5
```

Note that the final value that we compute is `5` instead of `6`, which was 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 $e \Downarrow v$ 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:

$$\textsc{EvalConstDecl}\quad \frac{e_d \Downarrow v_d \quad e_b' = e_b[v_d/x] \quad e_b' \Downarrow v}{\texttt{const}\ x = e_d\,;\ e_b \Downarrow v}$$

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:

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

$$\textsc{EvalCallRec}\quad \frac{\begin{array}{c} e_1 \Downarrow v_1 \\ v_1 = \textbf{function}\, x_1(x_2)\, e \qquad e_2 \Downarrow v_2 \qquad e' = e[v_2/x_2][v_1/x_1] \qquad e' \Downarrow v \end{array}}{e_1(e_2) \Downarrow v}$$

The remaining evaluation rules are straight-forward adaptations of the corresponding rules in the dynamic binding semantics.

Exercise 5.1. *Reconsider the problematic example program:*

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

Verify that this program indeed evaluates to `5` *under the new substitution-based big-step semantics.*

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:

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

$$\begin{array}{l}\textsc{EvalVal}\\ v \Downarrow v\end{array} \qquad \begin{array}{l}\textsc{EvalPlus}\\ \dfrac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = \mathit{toNum}(v_1) + \mathit{toNum}(v_2)}{e_1 \texttt{+} e_2 \Downarrow v}\end{array}$$

$$\begin{array}{l}\textsc{EvalTimes}\\ \dfrac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = \mathit{toNum}(v_1) \cdot \mathit{toNum}(v_2)}{e_1 \texttt{*} e_2 \Downarrow v}\end{array}$$

$$\begin{array}{l}\textsc{EvalConstDecl}\\ \dfrac{e_d \Downarrow v_d \qquad e_b' = e_b[v_d/x] \qquad e_b' \Downarrow v_b}{\textbf{const}\ x \texttt{=} e_d\texttt{;}\, e_b \Downarrow v_b}\end{array}$$

$$\begin{array}{l}\textsc{EvalEqual}\\ \dfrac{\begin{array}{c}e_1 \Downarrow v_1\\ e_2 \Downarrow v_2 \qquad b = (v_1 = v_2) \qquad v_1 \neq \texttt{function}\ p_1\texttt{(}x_1\texttt{)}\ e_3 \qquad v_2 \neq \texttt{function}\ p_2\texttt{(}x_2\texttt{)}\ e_4\end{array}}{e_1 \texttt{ === } e_2 \Downarrow b}\end{array}$$

$$\begin{array}{l}\textsc{EvalDisequal}\\ \dfrac{\begin{array}{c}e_1 \Downarrow v_1\\ e_2 \Downarrow v_2 \qquad b = (v_1 \neq v_2) \qquad v_1 \neq \texttt{function}\ p_1\texttt{(}x_1\texttt{)}\ e_3 \qquad v_2 \neq \texttt{function}\ p_2\texttt{(}x_2\texttt{)}\ e_4\end{array}}{e_1 \texttt{ !== } e_2 \Downarrow b}\end{array}$$

$$\begin{array}{l}\textsc{EvalIfThen}\\ \dfrac{e_1 \Downarrow v_1 \qquad \mathit{toBool}(v_1) = \texttt{true} \qquad e_2 \Downarrow v_2}{e_1 \texttt{ ? } e_2 \texttt{ : } e_3 \Downarrow v_2}\end{array}$$

$$\begin{array}{l}\textsc{EvalIfElse}\\ \dfrac{e_1 \Downarrow v_1 \qquad \mathit{toBool}(v_1) = \texttt{false} \qquad e_3 \Downarrow v_3}{e_1 \texttt{ ? } e_2 \texttt{ : } e_3 \Downarrow v_3}\end{array} \qquad \begin{array}{l}\textsc{EvalAndFalse}\\ \dfrac{e_1 \Downarrow v_1 \qquad \texttt{false} = \mathit{toBool}(v_1)}{e_1 \texttt{ \&\& } e_2 \Downarrow v_1}\end{array}$$

$$\begin{array}{l}\textsc{EvalAndTrue}\\ \dfrac{e_1 \Downarrow v_1 \qquad \texttt{true} = \mathit{toBool}(v_1) \qquad e_2 \Downarrow v_2}{e_1 \texttt{ \&\& } e_2 \Downarrow v_2}\end{array} \qquad \begin{array}{l}\textsc{EvalOrTrue}\\ \dfrac{e_1 \Downarrow v_1 \qquad \texttt{true} = \mathit{toBool}(v_1)}{e_1 \texttt{ || } e_2 \Downarrow v_1}\end{array}$$

$$\begin{array}{l}\textsc{EvalOrFalse}\\ \dfrac{e_1 \Downarrow v_1 \qquad \texttt{false} = \mathit{toBool}(v_1) \qquad e_2 \Downarrow v_2}{e_1 \texttt{ || } e_2 \Downarrow v_2}\end{array}$$

$$\begin{array}{l}\textsc{EvalCall}\\ \dfrac{e_1 \Downarrow v_1 \qquad v_1 = \texttt{function}\texttt{(}x\texttt{)}\ e \qquad e_2 \Downarrow v_2 \qquad e' = e[v_2/x] \qquad e' \Downarrow v}{e_1\texttt{(}e_2\texttt{)} \Downarrow v}\end{array}$$

$$\begin{array}{l}\textsc{EvalCallRec}\\ \dfrac{e_1 \Downarrow v_1 \qquad v_1 = \texttt{function}\ x_1\texttt{(}x_2\texttt{)}\ e \qquad e_2 \Downarrow v_2 \qquad e' = e[v_1/x_1][v_2/x_2] \qquad e' \Downarrow v}{e_1\texttt{(}e_2\texttt{)} \Downarrow v}\end{array}$$

Figure 5.3: Inference rules that define the substitution-based big-step SOS for static binding semantics of our language with functions

.

$$\textsc{EvalTypeErrorEqual}_1\quad \frac{e_1 \Downarrow \textbf{\texttt{function}}\ p(x)\ e \qquad bop \in \{\texttt{===}, \texttt{!==}\}}{e_1\ bop\ e_2 \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalTypeErrorEqual}_2\quad \frac{e_2 \Downarrow \textbf{\texttt{function}}\ p(x)\ e \qquad bop \in \{\texttt{===}, \texttt{!==}\}}{e_1\ bop\ e_2 \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalTypeErrorCall}\quad \frac{e_1 \Downarrow v_1 \qquad v_1 \neq \textbf{\texttt{function}}\ p(x)\ e}{e_1\texttt{(}e_2\texttt{)} \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalPropIf}\quad \frac{e_1 \Downarrow \textsf{typeerror}}{e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalPropBop}_1\quad \frac{e_1 \Downarrow \textsf{typeerror}}{e_1\ bop\ e_2 \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalPropBop}_2\quad \frac{e_2 \Downarrow \textsf{typeerror} \quad bop \notin \{\texttt{\&\&}, \texttt{||}\}}{e_1\ bop\ e_2 \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalPropAnd}\quad \frac{e_1 \Downarrow v_1 \qquad v_1 \neq \textsf{typeerror} \qquad \texttt{true} = \mathit{toBool}(v_1) \qquad e_2 \Downarrow \textsf{typeerror}}{e_1\ \texttt{\&\&}\ e_2 \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalPropOr}\quad \frac{\begin{array}{c} e_1 \Downarrow v_1 \\ v_1 \neq \textsf{typeerror} \qquad \texttt{false} = \mathit{toBool}(v_1) \qquad e_2 \Downarrow \textsf{typeerror}e_1\ \texttt{||}\ e_2 \Downarrow \textsf{typeerror}\end{array}}{}$$

$$\textsc{EvalPropCall}_1\quad \frac{e_1 \Downarrow \textsf{typeerror}}{e_1\texttt{(}e_2\texttt{)} \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalPropCall}_2\quad \frac{e_2 \Downarrow \textsf{typeerror}}{e_1\texttt{(}e_2\texttt{)} \Downarrow \textsf{typeerror}}$$

$$\textsc{EvalPropConstDecl}\quad \frac{e_d \Downarrow \textsf{typeerror}}{\textbf{\texttt{const}}\ x\,\texttt{=}\,e_d\texttt{;}\ e_b \Downarrow \textsf{typeerror}}$$

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

Now, consider the following function sumSqrs that computes the sum of the squares of the numbers in an interval [a,b):

```
function sumSqrs(a) {
  return b => a < b ? a * a + sumSqrs(a+1)(b) else 0;
}
```

The functions sumInts and sumSqrs 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 sumSqrs, it is a * 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:

```
function sum(f) {
  return a => b =>
      a < b ? f(a) + sum(f)(a + 1)(b) else 0
}
```

Now we can define sumInts and sumSqrs more succinctly in terms of sum as follows:

```
const sumInts = sum(a => a);
const sumSqrs = sum(a => 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 C. 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:

```
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:

```
const factorial = n =>
    function loop(i) (
      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 this construct 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:

```
const for =
  n => body => init =>
    function loop (i) (acc =>
      i <= n ? loop(i + 1)(body(i)(acc)) : acc
    )(1)(init);
const factorial =
  const body = i => acc => acc * i;
  n => for(n)(body)(1);
factorial(5)
```

The function **for** is a curried higher-order function. The expression

```
  for(n)(body)(init)
```

computes the value that is stored in the variable `acc` when the following JavaScript program is executed:

```
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 `sum` function from Section 5.4.1. Here is how we can define this function in Scala:

```
def sum(f: Int => Int, a: Int, b: Int) =
  if a < b then f(a) + sum(f, a + 1, b) else 0
```

The *function type* `Int =>Int` of the parameter `f` indicates that `f` is a function that takes a value of type `Int` and maps it again to an `Int`.

We can now define the function `sumSqrs` by first defining a function `square` that squares an integer number, and then applying `sum` to `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`:

```
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:

```
def sum(f: Int => Int): (Int, Int) => Int =
  def sumHelp(a: Int, b: Int): Int =
    if a < b then 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:

```
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:

```
def sumInts = sum(i => i)
def sumSqrs = sum(i => i * i)
```

In our curried version of sum, the function sumHelp is not recursive and is directly returned after being declared. We can thus simplify the definition of sum further by turning sumHelp into an anonymous function:

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

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:

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
  if a < b then f(a) + sum(f)(a, b) else 0
```

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 sumInts and sumSqrs thus look as follows in this case:

```
def sumInts = sum(i => i)_
def sumSqrs = sum(i => i * i)_
```

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 List in the Scala standard library.

Recall the list data structure that we studied in Section 2.3. We can define lists of integers recursively using an algebraic data type as follows:

```
enum List:
  case Nil
  case Cons(hd: Int, tl: List)
```

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

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

```
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)
```

Note that the infix cons operator :: is right-associative, so the parenthesis in the above example can be omitted:

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

As expected, we can use pattern matching to deconstruct lists into their components:

```
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 `append` function from Section 2.3 using Scala's `List` class:

```
def append[A](l1: List[A], l2: List[A]): List[A] =
  l1 match
    case Nil => l2
    case hd :: tl => hd :: append(tl, l2)
```

Note that `append` 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 `append`:

```
scala> append(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:

```
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> 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 on the elements.

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:

```
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:

```
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:

```
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`:

```
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`:

```
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

```
def sum(l: List[Int]): Int = l match
  case Nil => 0
  case hd :: tl => hd + sum(tl)
```

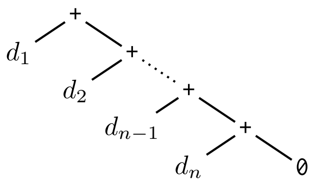
Consider a list `l` of n integer values:

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

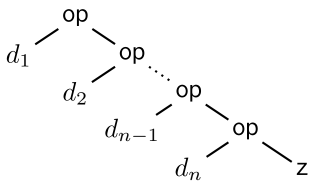
Then unrolling the recursion of `sum` on `l` yields the following computation

d_1 `+ (`d_2 `+ ... (`d_n `+ 0)...)`

That is, in the ith recursive call, we add the current head d_i to the sum of the values in the current tail. Here, 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:



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:



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:

```
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`:

```
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 `append` using `foldRight` as follows:

```
def append[A](l1: List[A], l2: List[A]): List[A] =
  foldRight(l1)(l2)(_ :: _)
```

Also the higher-order function `map` is just a special case of a fold-right:

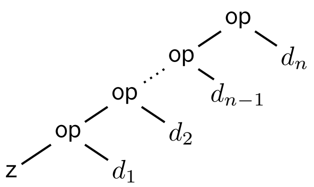
```
def map[A, B](l: List[A])(op: A => B): List[B] =
  foldRight(l)((Nil: List[B]))((h, l) => op(h) :: l)
```

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 left-to-right order:

`op(op(...(op(op(z,` d_1`),` d_2`), ...),` d_{n-1}`),` d_n`)`

The corresponding computation tree then looks as follows:



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`:

```
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:

```
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> 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
(_ + ",␣" + _)
res0: String = 1:a, 2:b, 3:c
```

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.

5.6 Computability (optional)

Computability theory studies the question “What is computable?”. The notion of *computability* relies on formal models of computation. Many formal models of computations have been proposed:

- Functions computable by Turing machines (Turing)
- General recursive functions defined by means of an equation calculus (Gödel-Herbrand-Kleene)
- μ-recursive functions and partial recursive functions (Gödel-Kleene)
- Functions defined from canonical deduction systems (Post)
- Functions given by certain algorithms over finite alphabets (Markov)
- Universal Register Machine-computable functions (Shepherdson-Sturgis)
- ...
- Any function you can write in your favorite programming language.

A fundamental result of computability theory is that all of these (and many other) models of computation are equivalent. That is, they give rise to the same class of computable functions. Any such model of computation is said to be *Turing complete*.[2]

So is our programming language Turing-complete? Alan Turing showed this in 1937. More specifically, he showed this for what is now called the *untyped lambda calculus*. This calculus corresponds to the following sub-language of our programming language considered in this section:

$$\begin{array}{rlr} x \in \mathit{Var} & & \text{variables} \\ e \in \mathit{Expr} & ::= x \mid e_1\texttt{(}e_2\texttt{)} \mid x \texttt{ => } e & \text{expressions} \end{array}$$

That is, expressions in the untyped lambda calculus are restricted to variables, function calls, and anonymous functions. The semantics of the calculus is usually given in terms of a small-step semantics (with static scoping) similar to the rules given in Figure 5.4.

In 1936, Alonzo Church introduced the untyped lambda calculus to prove the nonexistence of a solution to David Hilbert’s *Entscheidungsproblem*[3]. This problem asks whether there exists an algorithm that, given a logical formula as input, decides whether the formula is true or false for all possible assignments of values to the formula’s free variables. The lambda calculus served as a formalization of what it actually means for something to be an algorithm.

[2] An as of today unproved hypothesis attributed to Alonzo Church and Alan Turing is that there exists no meaningful computational model that is more expressive than a Turing-complete computational model.

[3] German for *decision problem*

Alan Turing proved the same negative result to the Entscheidungsproblem independently shortly after Church using Turing machines as an alternative formal computational model. That is, both the lambda calculus and Turing machines were invented to solve an open problem in logic, several years before the first programmable computer even existed.

How can it be that the simple language of the untyped lambda calculus is enough to express any computable function?

- There are no inbuilt types other than functions (e.g., no Booleans, integers, etc.).
- There are no control constructs like conditional expressions.
- There are no recursive definitions or looping constructs.

It turns out that the expressive power of the calculus stems from the fact that we can treat functions as data. It is all just a matter of finding the right encoding of concepts like numbers, Booleans, and recursion. The encodings presented below are inspired by Church's original work and are therefore also referred to as *Church encodings* in his honor.

5.6.1 Church Encodings

When we talk about data types such as integers and Booleans in a programming language, it is important to distinguish the abstract mathematical concept behind these data types (e.g. integer numbers), from the representation of these concepts in the language (e.g. numerals that stand for integer numbers). For example, consider

- 15,
- fifteen,
- XV, and
- `0F`.

These are different numerals that all represent the same number.

To show that concepts such as integers, Booleans, and the standard operations on these data types can be expressed in the lambda calculus, we simply have to agree on a particular representation of these concepts in terms of anonymous functions. The trick is to pick a representation of these data types that lends itself well to implementing their standard operations using the representation itself for performing the involved computation via function applications.

Before we start, we note that although the lambda calculus does not have **`const`** declarations built in, these can be mimicked by defining them as "syntactic sugar" using function applications like this:

$$\texttt{const}\ x\ \texttt{=}\ e_d\texttt{;}\ e_b \overset{\text{def}}{=} \texttt{(}x\ \texttt{=>}\ e_b\texttt{)(}e_d\texttt{)}$$

In the following, we will therefore continue to use **const** declarations for improved readability of code.

Let us start with our encoding of Booleans. How can we represent **true** and **false** in the lambda calculus? One reasonable definition is as follows:

- **true** is a function that takes two values and returns the first, i.e.

  ```
  const true = x => y => x
  ```

- **false** is a function that takes two values and returns the second, i.e.

  ```
  const false = x => y => y
  ```

The intuition behind this definition is that we can think of **true** and **false** as implementing the two branches of a conditional expression. That is, if we now define

```
const ite = b => x => y => b(x)(y)
```

then `ite` is a ternary function that implements a conditional expression, provided its first argument `b` is always either **true** or **false** as defined above.

Exercise 5.2. *Convince yourself that*

```
ite(b)(v1)(v2)
```

evaluates to `v1` *if* `b` *is* **true** *and to* `v2` *if* `b` *is* **false**.

Using these definitions, we can now define the standard logical operations on Booleans:

```
const and = a => b => a(b)(false)

const or = a => b => a(true)(b)

const not = b => x => y => b(y)(x)
// alternative: const not = b => b(false)(true)
```

Using a similar idea, one can encode natural numbers using anonymous functions. The number n is represented by a function which maps a successor function s and a zero element z to n applications of s to z: $s(s(\ldots s(z)\ldots))$:

```
const zero = s => z => z

const one = s => z => s(z)

const two = s => z => s(s(z))

const three = s => z => s(s(s(z)))

...
```

Note that `zero` and **`false`** are actually the same functions modulo α-renaming. As is common, we use the same mathematical object to mean different things in different contexts.

Here are the encodings of some operations on natural numbers:

```
// test for zero
const isZero = n => n(x => false)(true)

// addition by one (successor)
const succ = n => s => z => s(n(s)(z))

// addition
const plus = n => m => n(succ)(m)

// multiplication
const mult = n => m => n(plus(m))(zero)

// exponentiation
const exp = m => n => n(m)
```

For example, the definition of `plus` essentially says: apply `succ` `m`-times to `n`. In particular, if we take the expression `plus(three)(two)`, then this reduces to the computation: `succ(succ(succ(two)))`. Since `succ` encodes addition by one, this expression thus computes the encoding of 5. Here is a complete evaluation of the expression `plus(one)(two)`, which shows that the value obtained from this expression is `three`:

```
plus(one)(two)

= (n => m => n(succ)(m))(one)(two) // Definition of plus

→ (n => one(succ)(n))(two) // DoCall

→ one(succ)(two) // DoCall

= (s => z => s(z))(succ)(two) // Definition of one

→ (z => succ(z))(two) // DoCall

→ succ(two) // DoCall

= (n => s => z => s(n(s)(z)))(two) // Definition of succ

→ s => z => s(two(s(z))) // DoCall

= s => z => s((s1 => z1 => s1(s1(z1)))(s)(z)) // Definition of two
```

```
→ s => z => s((z1 => s(s(z1)))(z1)) // "DoCall"

→ s => z => s(s(s(z))) // "DoCall"

= three // Definition of three
```

The other operators work similarly. For instance, definition of `mult` says: apply `plus(n)` `m`-times to `zero`. In particular, the expression `mult(three)(two)` expands to the expression

```
plus(two)(plus(two)(plus(two)(zero)))
```

which reduces to the encoding of the number 6.

Encoding subtraction is a bit more involved. We proceed analogous to the case for addition and first define a function `pred` that encodes subtraction by one. The idea for `pred` is to define a function that given the number n, uses n to enumerate all pairs of numbers $\langle i, i-1\rangle$ for i between 1 and n. Then projection on the second component of the last pair $\langle n, n-1\rangle$ gives us $n-1$.

Let us start by defining functions for constructing and decomposing pairs:

```
// constructing a pair <x,y>
const pair = x => y => b => b(x)(y)

// projecting a pair p=<x,y> to x
const fst = p => p(true)

// projecting a pair p=<x,y> to y
const snd = p => p(false)
```

Note that we have

```
  fst(pair(x)(y)) → ... → x
```

and similarly

```
  snd(pair(x)(y)) → ... → y
```

Then subtraction by one can be defined as

```
const pred =
  n => snd(n(p => pair(succ(fst(p)))(fst(p)))
            (pair(zero)(zero)))
```

Using `pred` we can now define subtraction

```
const minus = n => m => m(pred)(n)
```

5.6.2 Expressing Recursion

How can we express recursion in the lambda calculus?

As an example, we will implement the factorial function $n!$ in the lambda calculus. Let us start from a standard JavaScript definition of factorial:

```
const fac = function fac(n) (n == 0 ? 1 : n * fac (n - 1))
```

First, we do a literal translation of this definition into the lambda calculus, making use of the operations and constants we have defined so far:

```
const fac = n =>
  ite(isZero(n))(one)(mult(n)(fac(minus(n)(one))))
```

Note that this is not quite right since we still have the call to fac in the body of the function definition, which used to be the recursive call to the function itself. Since we do not have recursion built into the lambda calculus, the definition does not work because fac now occurs free in the body of the function.

So let us do a simple trick and define a new function funFac that takes the function fac that we need for the recursive call as an additional input:

```
const facFun = fac => n =>
  ite(isZero(n))(one)(mult(n)(fac(minus(n)(one))))
```

Note that this definition is no longer recursive: the function facFun simply delegates the work that needs to be done in the recursive call to a function fac that is provided as an additional parameter.

How does this help? We can view facFun as a function that constructs the factorial function iteratively. That is, facFun takes fac as input and if fac is a function that approximates the factorial function (i.e. produces the same result as the factorial function on some inputs), then facFun returns a new function that is a better approximation of the factorial function than fac. To see this, consider the following sequence of definitions:

```
const fac_0 = n => zero
const fac_1 = facFun(fac_0)
const fac_2 = facFun(fac_1)
...
```

Observe that this sequence of functions approximates the factorial function with increasing precision: fac$_0$ always returns the representation of the number 0, so it never produces the correct result. However, fac$_1$ returns 1 if the input n is 0. So it is already a better approximation than fac$_0$. The function fac$_2$ returns the correct result for $n = 0$ and $n = 1$, etc. In general, for all natural numbers i and $n < i$, the expression fac$_i(n)$ reduces to the Church numeral representing $n!$. If we let i go to infinity, the sequence of approximations fac$_i$ converges to the factorial function itself. Formally, the factorial function fac is a function that satisfies the equation:

```
fac = facFun(fac)
```

In other words, we aim to construct fac as the *fixpoint* of facFun. What we thus need is a lambda expression that serves as a *fixpoint operator*, i.e., a function that calculates for us the fixpoint of another function (here facFun). Concretely, we seek a lambda expression fix such that the small-step semantics yields a sequence of reduction steps like this:

```
fix(f) → ... → f(fix(f)) → ... → f(f(fix(f))) → ...
```

If we find a term `fix` with this property, we can use it to construct for us automatically on demand the approximation $\mathtt{fac}_{n+1}$ that is needed to calculate $n!$ for a given natural number n.

There are various ways to define such a fixpoint operator in the lambda calculus. The most well-known and simplest one is the so-called *Y combinator* due to the logician Haskell Curry. To derive the Y combinator, we start with the following *self-replicating* expression:

```
(x => x(x))(x => x(x))
```

Observe that this expression is "stable" under small-step reductions:

```
  (x => x(x))(x => x(x))
→ (x => x(x))(x => x(x))
→ (x => x(x))(x => x(x))
→ ...
```

That is, the above expression describes an infinite self-replicating computation. Now all we have to do is "squeeze" the function `f` whose fixpoint we want to compute into the self-replicating expression. This yields the Y combinator:

```
const fix = f => (x => f(x(x)))(x => f(x(x)))
```

Observe what happens when we apply `fix` to some function `f`:

```
fix(f)
→ (x => f(x(x)))(x => f(x(x)))          // = fix(f)
→ f((x => f(x(x))(x => f(x(x)))         // = f(fix(f))
→ f(f((x => f(x(x)))(x => f(x(x)))))    // = f(f(fix(f))
→ ...
```

The sequence of reduction steps computes the approximation sequence for the fixpoint of `f` as required!

A minor complication that we still need to solve is that the above definition of the Y combinator does not quite work, yet. The problem is that the evaluation of `fix(f)` diverges: the self-application `x(x)` is repeatedly evaluated before the outermost call to `f` is ever evaluated. This is because the `DoCall` rule requires that the argument expression to a function call is fully evaluated, before the call itself is evaluated.

A simple solution to overcome this problem is to slightly rewrite the Y combinator so that the self-application `x(x)` is "protected" by another anonymous function `(y => f(x(x))(y))`:

```
const fix = f => (x => f(y => x(x)(y)))(x => f(y => x(x)(y)))
```

Effectively, this rewrite accounts for the fact that we we only apply `fix` to functions like `facFun` whose fixpoints are themselves functions.

With the new definition of `fix` we now get

```
fix(f)
→ (x => f(y => x(x)(y)))(x => f(y => x(x)(y)))
→ f(y => (x => f(y => x(x)(y)))(x => f(y => x(x)(y))))
=  f(y => fix(f)(y))
→ ... // continues with the evaluation of the body of f!
```

We now define

```
const fac = fix(facFun)
```

If we now evaluate `fac(two)`, then the value for the parameter `y` of the additional anonymous function in `fix` will be supplied by the argument `minus(two)(one)` of the call to `fac` inside of `facFun`. That is, the fixpoint of `fac` is now computed on-demand and only up to the approximation $\texttt{fac}_{n+1}$ needed for computing `fac`(n) for a given n.

Note that the expression `fix(facFun)` is indeed a valid expression in the lambda calculus. In particular, it does not make use of explicit recursion nor does it rely on any inbuilt data types or operations for numbers and Booleans.

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

Each of the following two expressions in our language from Chapter 5 will result in a dynamic type error during evaluation:

$$(3 + 4)(0)$$
$$1 * 3 \mathbin{\texttt{===}} (x \mathbin{\texttt{=>}} 0)$$

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 in this chapter.

6.1.2 A Simple Typed Language

We extend our language from Chapter 5 with types. We choose a 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:

$$
\begin{array}{rll}
n \in & \mathit{Num} & \text{numbers (double)} \\
x \in & \mathit{Var} & \text{variables} \\
b \in & \mathit{Bool} ::= \texttt{true} \mid \texttt{false} & \text{Booleans} \\
\tau \in & \mathit{Typ} ::= \texttt{Bool} \mid \texttt{Num} \mid \tau_1 \Rightarrow \tau_2 & \text{types} \\
v \in & \mathit{Val} ::= n \mid b \mid \texttt{function}\ p(x\texttt{:}\tau)\ t\ e & \text{values} \\
e \in & \mathit{Expr} ::= x \mid v \mid e_1\ \mathit{bop}\ e_2 \mid e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \mid & \text{expressions} \\
 & \qquad \texttt{const}\ x = e_1; e_2 \mid e_1\texttt{(}e_2\texttt{)} & \\
\mathit{bop} \in & \mathit{Bop} ::= \texttt{+} \mid \texttt{*} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{\&\&} \mid \texttt{||} & \text{binary operators} \\
 & p ::= x \mid \epsilon & \text{function names} \\
 & t ::= \texttt{:}\tau \mid \epsilon & \text{return type annotations}
\end{array}
$$

A type τ is either one of the *base types* **Bool** or **Num**, or a *function type* $\tau_1 \Rightarrow \tau_2$, where τ_1 and τ_2 are again types. A function type describes the signature of a function that has a parameter of type τ_1 and returns a value of type τ_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 programmer 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 $e \to e'$ 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 binding 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* $\Gamma : \mathit{Var} \rightharpoonup \mathit{Typ}$ 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 Γ, the expression e has type τ". The inference rules defining the typing relation are given in Figure 6.2.

If a type τ exists such that $\Gamma \vdash e : \tau$ holds, we say that e is *well-typed* under Γ. 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 Γ and e, we say that e has a type error under Γ.

The type inference rules are designed in such a way that if an expression e is well-typed in an environment Γ, then the evaluation of e will not get stuck for any substitution of the free variables in e with values that are consistent

$$\textsc{SearchBop1}\quad \frac{e_1 \to e_1'}{e_1\ \mathit{bop}\ e_2 \to e_1'\ \mathit{bop}\ e_2} \qquad \textsc{SearchBop2}\quad \frac{\mathit{bop} \notin \{\texttt{\&\&}, \texttt{||}\} \qquad e_2 \to e_2'}{v_1\ \mathit{bop}\ e_2 \to v_1\ \mathit{bop}\ e_2'}$$

$$\textsc{SearchIf}\quad \frac{e_1 \to e_1'}{e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \to e_1'\ \texttt{?}\ e_2\ \texttt{:}\ e_3} \qquad \textsc{SearchConst}\quad \frac{e_d \to e_d'}{\texttt{const}\ x\,\texttt{=}\,e_d\texttt{;}\ e_b \to \texttt{const}\ x\,\texttt{=}\,e_d'\texttt{;}\ e_b}$$

$$\textsc{SearchCall1}\quad \frac{e_1 \to e_1'}{e_1\texttt{(}e_2\texttt{)} \to e_1'\texttt{(}e_2\texttt{)}} \qquad \textsc{SearchCall2}\quad \frac{e_2 \to e_2'}{v_1\texttt{(}e_2\texttt{)} \to v_1\texttt{(}e_2'\texttt{)}} \qquad \textsc{DoArith}\quad \frac{n = n_1\ \mathit{bop}\ n_2 \qquad \mathit{bop} \in \{\texttt{+}, \texttt{*}\}}{n_1\ \mathit{bop}\ n_2 \to n}$$

$$\textsc{DoEqual}\quad \frac{b = v_1\ \mathit{bop}\ v_2 \quad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{v_1\ \mathit{bop}\ v_2 \to b} \qquad \textsc{DoAndFalse}\quad \texttt{false}\ \texttt{\&\&}\ e_2 \to \texttt{false} \qquad \textsc{DoAndTrue}\quad \texttt{true}\ \texttt{\&\&}\ e_2 \to e_2$$

$$\textsc{DoOrTrue}\quad \texttt{true}\ \texttt{||}\ e_2 \to \texttt{true} \qquad \textsc{DoOrFalse}\quad \texttt{false}\ \texttt{||}\ e_2 \to e_2 \qquad \textsc{DoIfThen}\quad \texttt{true}\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \to e_2$$

$$\textsc{DoIfElse}\quad \texttt{false}\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \to e_3 \qquad \textsc{DoConst}\quad \texttt{const}\ x\,\texttt{=}\,v_d\texttt{;}\ e_b \to e_b[v_d/x]$$

$$\textsc{DoCall}\quad \frac{v_1 = \texttt{function}\texttt{(}x\texttt{:}\tau\texttt{)}\ t\ e}{v_1\texttt{(}v_2\texttt{)} \to e[v_2/x]} \qquad \textsc{DoCallRec}\quad \frac{v_1 = \texttt{function}\ x_1\texttt{(}x_2\texttt{:}\tau_2\texttt{)}\ \texttt{:}\tau' e}{v_1\texttt{(}v_2\texttt{)} \to e[v_1/x_1][v_2/x_2]}$$

Figure 6.1: Small-step operational semantics for typed JakartaScript

with the typing environment Γ. We will make this intuition formally precise in Section 6.2.

It is instructive to compare the rules of the typing relation with the corresponding rules for the big-step operational semantics with dynamic binding. Note that for the typing relation, the order in which the types for subexpressions are inferred is irrelevant. In fact, it is easy to see that the typing rules are deterministic, i.e., for any Γ and e, there is at most one type τ 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 Γ and an expression e to a type τ. 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 were then 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 Γ, 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:

$$3 + \texttt{true}$$

$$0 \;?\; 1 \;:\; 2$$

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 errors in programs and these errors can be difficult to debug. Also, they can cause non-negligible run-time costs. 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:

$$(0 \mathbin{===} 1) \;?\; 1 \;:\; \texttt{false}$$

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 expres-

$$\textsc{TypeBool}\quad \Gamma \vdash b : \texttt{Bool} \qquad\qquad \textsc{TypeNum}\quad \Gamma \vdash n : \texttt{Num} \qquad\qquad \textsc{TypeVar}\quad \frac{x \in \mathsf{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\textsc{TypeAndOr}\quad \frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \texttt{Bool} \qquad \mathit{bop} \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\, \mathit{bop}\, e_2 : \texttt{Bool}}$$

$$\textsc{TypeArith}\quad \frac{\Gamma \vdash e_1 : \texttt{Num} \qquad \Gamma \vdash e_2 : \texttt{Num} \qquad \mathit{bop} \in \{\texttt{+}, \texttt{*}\}}{\Gamma \vdash e_1\, \mathit{bop}\, e_2 : \texttt{Num}}$$

$$\textsc{TypeEqual}\quad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \neq \tau_1 \Rightarrow \tau_2 \qquad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \texttt{Bool}}$$

$$\textsc{TypeConst}\quad \frac{\Gamma \vdash e_d : \tau_d \qquad \Gamma' = \Gamma[x \mapsto \tau_d] \qquad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash \texttt{const}\ x \,\texttt{=}\, e_d\,\texttt{;}\ e_b : \tau_b}$$

$$\textsc{TypeIf}\quad \frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 : \tau} \qquad\qquad \textsc{TypeCall}\quad \frac{\Gamma \vdash e_1 : \tau_2 \Rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \tau}$$

$$\textsc{TypeFun}\quad \frac{\Gamma' = \Gamma[x \mapsto \tau] \qquad \Gamma' \vdash e : \tau'}{\Gamma \vdash \texttt{function(}x\texttt{:}\tau\texttt{)}\ e : \tau \Rightarrow \tau'} \qquad\qquad \textsc{TypeFunAnn}\quad \frac{\Gamma' = \Gamma[x \mapsto \tau] \qquad \Gamma' \vdash e : \tau'}{\Gamma \vdash (\texttt{function(}x\texttt{:}\tau\texttt{)}\ \texttt{:}\tau'\ e\ ) : \tau \Rightarrow \tau'}$$

$$\textsc{TypeFunRec}\quad \frac{\Gamma' = \Gamma[x_1 \mapsto \tau_1][x_2 \mapsto \tau_2] \qquad \Gamma' \vdash e : \tau' \qquad \tau_1 = \tau_2 \Rightarrow \tau'}{\Gamma \vdash (\texttt{function}\ x_1\texttt{(}x_2\texttt{:}\tau_2\texttt{)}\ \texttt{:}\tau'\ e\ ) : \tau_1}$$

Figure 6.2: Type inference rules

sion to be well-typed, we would have to introduce a *supertype* of **Num** and **Bool** that describes both *Num* and *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., **Bool** and **Num**) 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 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:

```
const id = 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 τ such that the following expression is well-typed in our new language:

```
const id = (x:τ) => x;
id(false) ? id(2) : id(1)
```

If we choose $\tau =$ **Bool**, then the calls `id(2)` and `id(1)` will yield static type errors. If we choose $\tau =$ **Num**, then the call `id(false)` will not be well-typed. Finally, if we choose $\tau = \tau_1 \Rightarrow \tau_2$ for any types τ_1 and τ_2, then none of the calls to `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 $e \in \mathit{Expr}$, if e is closed and well-typed, then either $e \in \mathit{Val}$ or $e \to e'$ for some $e' \in \mathit{Expr}$.

2. **Preservation**: for all $e, e' \in \mathit{Expr}$ such that $e \to e'$, 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 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 $\texttt{Bool}$, then $v \in \mathit{Bool}$.*
2. *If v is of type $\texttt{Num}$, then $v \in \mathit{Num}$.*
3. *If v is of type $\tau \Rightarrow \tau'$, then either*
   - $v = \texttt{function}(x\,{:}\,\tau)\; t\, e$ *where* $t = \epsilon$ *or* $t = \,{:}\tau'$*, or*
   - $v = \texttt{function}\; x_1(x_2\,{:}\,\tau)\; {:}\tau' e$

Proof (Exercise). The lemma follows easily from the rules of the typing relation that involve values. □

Theorem 6.2 (Progress)**.** *Let $e \in \mathit{Expr}$ be a closed expression. If e is well-typed, then either $e \in \mathit{Val}$ or $e \to e'$ for some $e' \in \mathit{Expr}$.*

Proof (Exercise). Since e is well-typed and closed we have $\emptyset \vdash e : \tau$ for some type τ. 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 TypeNum, TypeBool, TypeFunction, TypeFunctionAnn, or TypeFunctionRec, then e must be a value and there is nothing to be proved. The rule TypeVar cannot be the last rule used in the derivation since it would imply that $e = x$ for some variable $x \in \mathit{Var}$, contradicting the assumption that e is closed. From the remaining cases, we only show the case for the rule TypeCall. The other cases are similar and left as an exercise.

Thus, assume that TypeCall is the last rule that was used in the derivation of $\emptyset \vdash e : \tau$. From the premises of TypeCall, it follows that e must be of the form $e = e_1\texttt{(}e_2\texttt{)}$ such that $\emptyset \vdash e_1 : (\tau' \Rightarrow \tau)$ and $\emptyset \vdash e_2 : \tau'$ for some e_1, e_2, x, and τ'. We distinguish three subcases:

Case 1 $\underline{e_1 \text{ is not a value}}$: 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 $e_1 \to e_1'$. Then by rule SearchCall$_1$, we can conclude $e \to e_1'\texttt{(}e_2\texttt{)}$.

Case 2 $\underline{e_1 \text{ is a value but } e_2 \text{ is not}}$: similar to the previous case, we can conclude by induction hypothesis and rule SearchCall$_2$ that $e \to e_1\texttt{(}e_2'\texttt{)}$ for some e_2'.

Case 3 $\underline{e_1 \text{ and } e_2 \text{ are both values}}$: it follows from the Canonical Forms Lemma that e_1 must be of the forms

$$e_1 = \textbf{function}\texttt{(}x\texttt{:}\tau'\texttt{)}\, t\, e' \text{ or } e_1 = \textbf{function}\; x_1\texttt{(}x_2\texttt{:}\tau'\texttt{)}\;\texttt{:}\tau\, e'.$$

In the first case, we conclude from rule DoCall that $e \to e'[e_2/x]$. In the second case, rule DoCallRec implies that $e \to e'[e_1/x_1][e_2/x_2]$.

□

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* $\Gamma[x \mapsto \tau_1][y \mapsto \tau_2] \vdash e : \tau$ *and* $x \neq y$*, then* $\Gamma[y \mapsto \tau_2][x \mapsto \tau_1] \vdash e : \tau$*.*

Proof (Exercise). The intuition for why this lemma is correct is that if we extend an environment $\Gamma' = \Gamma[x \mapsto \tau_1]$ with another binding for a variable y, $\Gamma'' = \Gamma'[y \mapsto \tau_2]$, then the second extension does not interfere with the first extension unless x and y are equal.

Thus, define $\Gamma_1 = \Gamma[x \mapsto \tau_1][y \mapsto \tau_2]$ and $\Gamma_2 = \Gamma[y \mapsto \tau_2][y \mapsto \tau_1]$. All you need to prove is that $\Gamma_1 = \Gamma_2$. To do so, prove that for all variables z, if $z \in \mathsf{dom}(\Gamma_1)$ then $z \in \mathsf{dom}(\Gamma_2)$ 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 $\_[\_ \mapsto \_]$ 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 Γ, then it is also well-typed under any environment $\Gamma' = \Gamma[x \mapsto \tau']$, 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 τ' 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 τ' will never actually be used in the typing derivation for e.

Lemma 6.4 (Weakening)**.** *If* $\Gamma \vdash e : \tau$ *and* $x \notin \mathit{fv}(e)$*, 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. □

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 $\Gamma \vdash e_1[e_2/x] : \tau_1$ directly follows from the assumption $\Gamma \vdash e_2 : \tau_2$. 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

- $e_1 = \textbf{function}(y{:}\tau)\; e$,
- $\tau_1 = \tau \Rightarrow \tau'$, 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 $y \notin \mathit{fv}(e_2)$. 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 $y \notin \mathit{fv}(e_2)$ 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 $\Gamma[y \mapsto \tau] \vdash e[e_2/x] : \tau'$. Thus, by rule TypeFunction we conclude

$$\Gamma \vdash \textbf{function}(y{:}\tau)\; (e[e_2/x]) : \tau \Rightarrow \tau'.$$

By the definition of substitution and since $y \neq x$, we have

$$\textbf{function}(y{:}\tau)\; (e[e_2/x]) = (\textbf{function}(y{:}\tau)\; e)[e_2/x] = e_1[e_2/x]$$

Hence, we conclude $\Gamma \vdash e_1[e_2/x] : \tau_1$. □

Theorem 6.6 (Preservation)**.** *Let $e, e' \in Expr$ such that $e \to e'$. 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 τ. We prove a slightly stronger property than required, namely that if $e \to e'$, then e' is closed and $\emptyset \vdash e' : \tau$ (i.e., the exact type τ 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 $e_1 \to e_1'$, 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 $e \to e'$. 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 JakartaScript 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 JakartaScript:

$$\texttt{function}\ \mathit{fac}(x)\ (x\ \texttt{===}\ 0\ \texttt{?}\ 1\ \texttt{:}\ x \ast \mathit{fac}(x - 1))$$

For this function to be well-typed in our simple type system, we would have to explicitly annotate the parameter x of function *fac* as well as the function's return type. However, if we take a closer look at the function's body:

$$x\ \texttt{===}\ 0\ \texttt{?}\ 1\ \texttt{:}\ x \ast \mathit{fac}(x - 1)$$

we observe that there is really only one possible type annotation that can work. Specifically, if we look at the test $x\ \texttt{===}\ 0$ in the conditional, then we can infer that x must be of type **Num** since it is compared with the value 0. Similarly, from the fact that the result of the recursive call $\mathit{fac}(x - 1)$ is used in a multiplication operation, we can infer that the return value of *fac* must also be of type **Num**. 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 **Num** and **Bool**, or a function type built from base types). For example, consider the following curried function

$$\mathit{apply} = x \Rightarrow f \Rightarrow f(x)$$

From the subexpression $f(x)$ of *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 τ_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 τ_1 and what the specific return type τ_2 of f is. In fact, for the evaluation of a call to the function *apply* the specific types of the parameter and return value of f don't matter. We say that *apply* is parameterized in the types τ_1 and τ_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:

$$\begin{array}{lr} \alpha \in \mathit{TVar} & \text{type variables} \\ \tau \in \mathit{Typ} ::= \mathbf{Bool} \mid \mathbf{Num} \mid \alpha \mid \tau_1 \Rightarrow \tau_2 & \text{types} \end{array}$$

If a type τ does not contain any type variables, it is called *monomorphic*, otherwise it is called *polymorphic*. We can think of polymorphic types as types that parameterize over monomorphic types. For example, the type $\alpha \Rightarrow \mathbf{Bool}$ stands for the monomorphic types $\mathbf{Bool} \Rightarrow \mathbf{Bool}$, $\mathbf{Num} \Rightarrow \mathbf{Bool}$, $(\mathbf{Bool} \Rightarrow \mathbf{Bool}) \Rightarrow \mathbf{Bool}$, 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:

$$x \mathtt{===} 0 \mathtt{\ ?\ } 1 \mathtt{\ :\ } x \mathtt{*} \mathit{fac}\mathtt{(}x \mathtt{-} 1\mathtt{)}$$

Let us call this expression e. We generate the type variables and typing constraints for the subexpressions of e in one go:

x	α_x	-
0	α_0	$\alpha_0 \doteq \mathtt{Num}$
$x \mathtt{===} 0$	α_{eq}	$\alpha_x \doteq \alpha_0$, $\alpha_{eq} \doteq \mathtt{Bool}$,
1	α_1	$\alpha_1 \doteq \mathtt{Num}$
fac	α_{fac}	-
$x \mathtt{-} 1$	$\alpha_{\mathtt{-}}$	$\alpha_x \doteq \mathtt{Num}$, $\alpha_1 \doteq \mathtt{Num}$, $\alpha_{\mathtt{-}} \doteq \mathtt{Num}$
$\mathit{fac}\mathtt{(}x \mathtt{-} 1\mathtt{)}$	α_{call}	$\alpha_{\mathit{fac}} \doteq (\alpha_{\mathtt{-}} \Rightarrow \alpha_{call})$
$x \mathtt{*} \mathit{fac}\mathtt{(}x \mathtt{-} 1\mathtt{)}$	$\alpha_{\mathtt{*}}$	$\alpha_x \doteq \mathtt{Num}$, $\alpha_{call} \doteq \mathtt{Num}$, $\alpha_{\mathtt{*}} \doteq \mathtt{Num}$
$x \mathtt{===} 0 \mathtt{\ ?\ } 1 \mathtt{\ :\ } x \mathtt{*} \mathit{fac}\mathtt{(}x \mathtt{-} 1\mathtt{)}$	α_{ite}	$\alpha_{eq} \doteq \mathtt{Bool}$, $\alpha_{ite} \doteq \alpha_1$, $\alpha_{ite} \doteq \alpha_{\mathtt{*}}$

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 $\doteq$ 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 $x \mathtt{-} 1$ of e has the associated type variable $\alpha_{\mathtt{-}}$. Since we know that the arguments of the binary operator `-` must be of type `Num`, we obtain two typing constraints $\alpha_x \doteq \mathtt{Num}$ and $\alpha_1 \doteq \mathtt{Num}$, where α_x and α_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 `Num`. Hence, we obtain the additional constraint $\alpha_{\mathtt{-}} \doteq \mathtt{Num}$. Another particularly interesting case is the call expression $\mathit{fac}\mathtt{(}x \mathtt{-} 1\mathtt{)}$.

From this expression we can infer that *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 α_{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. $\alpha_0 \doteq \mathtt{Num}$
2. $\alpha_x \doteq \alpha_0$
3. $\alpha_{eq} \doteq \mathtt{Bool}$
4. $\alpha_1 \doteq \mathtt{Num}$
5. $\alpha_{-} \doteq \mathtt{Num}$
6. $\alpha_{fac} \doteq (\alpha_{-} \Rightarrow \alpha_{call})$
7. $\alpha_{call} \doteq \mathtt{Num}$
8. $\alpha_{\star} \doteq \mathtt{Num}$
9. $\alpha_{eq} \doteq \mathtt{Bool}$
10. $\alpha_{ite} \doteq \alpha_1$
11. $\alpha_{ite} \doteq \alpha_{-}$

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 $\sigma : \mathit{TVar} \to \mathit{Typ}$ from type variables to types, such that if we substitute the type variables occuring in each constraint according to σ, then the two sides of each constraint become equal. That is, σ 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 τ according to σ. We refer to the problem of finding such a solution σ for a given set of typing constraints C as the *unification problem*. A solution σ of a unification problem instance C is called a *unifier* of C.

We can compute a unifier σ 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 σ_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 σ_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. $\alpha_0 \doteq \mathtt{Num}$

Observe that the left-hand side of this constraint is a type variable α_0. Thus, to solve this specific constraint, all we need to do is map α_0 to the type on the right-hand side, which is $\mathtt{Num}$. We therefore define $\sigma_1 = \sigma_0[\alpha_0 \mapsto \mathtt{Num}]$. Observe, that σ_1 is indeed a unifier of constraint 1, since

$$\alpha_0\sigma_1 = \mathtt{Num} = \mathtt{Num}\,\sigma_1$$

More generally, the unification algorithm will maintain the invariant that after processing the ith constraint, the current candidate solution σ_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 σ_i to the unprocessed constraints after each extension. That is, in our example, we substitute α_0 by $\texttt{Num}$ in constraints 2 to 11, which gives us the new set of constraints:

2. $\alpha_x \doteq \texttt{Num}$
3. $\alpha_{eq} \doteq \texttt{Bool}$
4. $\alpha_1 \doteq \texttt{Num}$
5. $\alpha_{-} \doteq \texttt{Num}$
6. $\alpha_{fac} \doteq (\alpha_{-} \Rightarrow \alpha_{call})$
7. $\alpha_{call} \doteq \texttt{Num}$
8. $\alpha_{\star} \doteq \texttt{Num}$
9. $\alpha_{eq} \doteq \texttt{Bool}$
10. $\alpha_{ite} \doteq \alpha_1$
11. $\alpha_{ite} \doteq \alpha_{-}$

We continue processing the constraints in the given order. The constraints 2 to 6 are similar to constraint 1. We extend the candidate unifier σ_0 for each of these cases as described above to obtain the following remaining constraints and current candidate unifier σ_6:

7. $\alpha_{call} \doteq \texttt{Num}$
8. $\alpha_{\star} \doteq \texttt{Num}$
9. $\texttt{Bool} \doteq \texttt{Bool}$
10. $\alpha_{ite} \doteq \texttt{Num}$
11. $\alpha_{ite} \doteq \texttt{Num}$

$$\begin{aligned}\sigma_6 = \{\ & \alpha_0 \mapsto \texttt{Num},\\ & \alpha_x \mapsto \texttt{Num},\\ & \alpha_{eq} \mapsto \texttt{Bool},\\ & \alpha_1 \mapsto \texttt{Num},\\ & \alpha_{-} \mapsto \texttt{Num},\\ & \alpha_{fac} \mapsto (\texttt{Num} \Rightarrow \alpha_{call})\}\end{aligned}$$

The case for constraint 7 is similar to the previous cases, except that the type variable α_{call} now also appears in the type to which α_{fac} is mapped by the current candidate unifier σ_6. In order to maintain our invariant that the candidate unifier unifies all constraints processed so far, we also have to apply the mapping $\alpha_{call} \mapsto \texttt{Num}$ to σ_6 before we extend σ_6 with this new mapping.

After we do this, we obtain the new mapping:

$$\begin{aligned}\sigma_7 = \{\ & \alpha_0 \mapsto \texttt{Num},\\ & \alpha_x \mapsto \texttt{Num},\\ & \alpha_{eq} \mapsto \texttt{Bool},\\ & \alpha_1 \mapsto \texttt{Num},\\ & \alpha_{-} \mapsto \texttt{Num},\\ & \alpha_{fac} \mapsto (\texttt{Num} \Rightarrow \texttt{Num}),\\ & \alpha_{call} \mapsto \texttt{Num}\}\end{aligned}$$

The unprocessed constraints 8 to 11 remain unchanged since α_{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 $\sigma_8 = \sigma_7[\alpha_\star \mapsto \texttt{Num}]$.

Constraint 9 is again interesting. It is now of the form $\texttt{Bool} \doteq \texttt{Bool}$. The two sides of this constraint are already unified, so there is no need to extend σ_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

$$\begin{aligned}\sigma_{11} = \{\ & \alpha_0 \mapsto \texttt{Num},\ \alpha_x \mapsto \texttt{Num},\ \alpha_{eq} \mapsto \texttt{Bool},\ \alpha_1 \mapsto \texttt{Num},\\ & \alpha_{-} \mapsto \texttt{Num},\ \alpha_{fac} \mapsto (\texttt{Num} \Rightarrow \texttt{Num}),\\ & \alpha_{call} \mapsto \texttt{Num},\ \alpha_\star \mapsto \texttt{Num},\ \alpha_{ite} \mapsto \texttt{Num}\}\end{aligned}$$

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:

$$apply = x \texttt{ => } f \texttt{ => } f\texttt{(}x\texttt{)}$$

From the expression *apply* we collect the following subexpressions with associated type variables and typing constraints:

f	α_f	-
x	α_x	-
$f\texttt{(}x\texttt{)}$	α_{call}	$\alpha_f \doteq (\alpha_x \Rightarrow \alpha_{call})$
$f \texttt{ => } f\texttt{(}x\texttt{)}$	α_{fun}	$\alpha_{fun} \doteq (\alpha_f \Rightarrow \alpha_{call})$
$x \texttt{ => } f \texttt{ => } f\texttt{(}x\texttt{)}$	α_{apply}	$\alpha_{apply} \doteq (\alpha_x \Rightarrow \alpha_{fun})$

That is, we need to solve the following unification problem to show that *apply* is well-typed:

1. $\alpha_f \doteq (\alpha_x \Rightarrow \alpha_{call})$
2. $\alpha_{fun} \doteq (\alpha_f \Rightarrow \alpha_{call})$
3. $\alpha_{apply} \doteq (\alpha_x \Rightarrow \alpha_{fun})$

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:

$$\begin{aligned}\sigma_2 = \{\ & \alpha_f \mapsto \alpha_x \Rightarrow \alpha_{call}\\ & \alpha_{fun} \mapsto \alpha_x \Rightarrow \alpha_{call} \Rightarrow \alpha_{call}\}\end{aligned}$$

The remaining constraint 3 now looks as follows:

3. $\alpha_{apply} \doteq (\alpha_x \Rightarrow (\alpha_x \Rightarrow \alpha_{call}) \Rightarrow \alpha_{call})$

Processing this remaining constraint yields the actual unifier of the original set of constraints:

$$\begin{aligned}\sigma_3 = \{\ & \alpha_f \mapsto \alpha_x \Rightarrow \alpha_{call}\\ & \alpha_{fun} \mapsto (\alpha_x \Rightarrow \alpha_{call}) \Rightarrow \alpha_{call}\\ & \alpha_{apply} \mapsto (\alpha_x \Rightarrow (\alpha_x \Rightarrow \alpha_{call}) \Rightarrow \alpha_{call})\}\end{aligned}$$

We obtain the inferred type of *apply* by looking up the type to which the associated type variable α_{apply} is mapped by the unifier σ_3. Note that this type is polymorphic as it still contains the type variables α_x and α_{call}. This tells us that we can safely call *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 *apply* are safe:

$$\begin{aligned}&\mathit{apply}\texttt{(}3\texttt{)(}x\ \texttt{=>}\ x+2\texttt{)}\\ &\mathit{apply}\texttt{(true)(}x\ \texttt{=>}\ x\ \texttt{|| false)}\end{aligned}$$

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. $\alpha_{+} \doteq \texttt{Num}$
2. $\alpha_x \doteq \texttt{Num}$
3. $\alpha_1 \doteq \texttt{Num}$
4. $\alpha_3 \doteq \texttt{Num}$

5. $\alpha_x \doteq \texttt{Bool}$

6. $\alpha_{ite} \doteq \alpha_{+}$

7. $\alpha_{ite} \doteq \alpha_3$

After processing the first 4 constraints we obtain the following candidate unifier

$$\sigma_4 = \{\alpha_{+} \mapsto \texttt{Num}, \alpha_x \mapsto \texttt{Num}, \alpha_1 \mapsto \texttt{Num}, \alpha_3 \mapsto \texttt{Num}\}$$

and the remaining set of constraints now looks as follows:

5. $\texttt{Num} \doteq \texttt{Bool}$

6. $\alpha_{ite} \doteq \texttt{Num}$

7. $\alpha_{ite} \doteq \texttt{Num}$

Continuing with constraint 5, we detect a problem. The constraint is now of the form $\texttt{Num} \doteq \texttt{Bool}$. Since **Num** and **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 **Bool**. Then we use x again in the "then" branch as an argument to +, which means that x must also have type **Num** – 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:

$$\alpha_x \doteq \alpha_x \Rightarrow \alpha_{call}$$

On first sight, it appears that this constraint has a simple solution given by the following mapping:

$$\sigma = \{\alpha_x \mapsto \alpha_x \Rightarrow \alpha_{call}\}$$

However, observe that σ is not actually a unifier of the constraint since applying σ to the two sides of the constraint does not make the two sides equal:

$$\begin{aligned} \alpha_x\sigma &= \alpha_x \Rightarrow \alpha_{call} \\ &\neq (\alpha_x \Rightarrow \alpha_{call}) \Rightarrow \alpha_{call} \\ &= (\alpha_x \Rightarrow \alpha_{call})\sigma \end{aligned}$$

In fact, there is no mapping of the type variables α_x and α_{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 $\alpha \doteq \tau$, we first check whether α occurs in τ before we extend the current candidate unifier with the mapping $\alpha \mapsto \tau$. If α occurs in τ, 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 **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

$$
\begin{array}{rlr}
n \in \mathit{Num} & & \text{Nums (double)} \\
x \in \mathit{Var} & & \text{variables} \\
b \in \mathit{Bool} ::= & \mathtt{true} \mid \mathtt{false} & \text{Booleans} \\
v \in \mathit{Val} ::= & n \mid b \mid \mathtt{function}\, p(x)\, e & \text{values} \\
e \in \mathit{Expr} ::= & x \mid v \mid e_1\, \mathit{bop}\, e_2 \mid e_1 \ \mathtt{?}\ e_2 \ \mathtt{:}\ e_3 \mid e_1(e_2) & \text{expressions} \\
\mathit{bop} \in \mathit{Bop} ::= & \mathtt{+} \mid \mathtt{*} \mid \mathtt{\&\&} \mid \mathtt{||} & \text{binary operators} \\
p ::= & x \mid \epsilon & \text{function names}
\end{array}
$$

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 Γ and the expression e are the input of the relation. The output computed by the relation is the type variable α and the set of typing constraints C. Informally, this judgment states that under typing environment Γ, the expression e is well-typed, provided that the typing constraints C have a solution σ. In particular, given a solution σ of C, the type of e for this solution is $\sigma(\alpha)$.

The typing environment Γ is simply a mapping from (expression) variables to type variables, i.e., $\Gamma : \mathit{Var} \rightharpoonup \mathit{TVar}$. 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 σ and returns an extension of σ to an actual unifier of C, or $\bot$ 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 $\tau_1 \doteq \tau_2$. The function *unify* applies the function *unifyOne* to the individual constraints in C, one at a time. The defining cases for *unify* and *unifyOne* should be read top-down. The first matching case applies, similar to a match expression in a Scala program.

In the case for $\alpha \doteq \tau$ of *unifyOne*, the actual extension of the candidate unifier σ happens. The test $\alpha \in \mathit{tv}(\tau)$ 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., $\mathit{tv}(\alpha \Rightarrow \beta) = \{\alpha, \beta\}$. If the occurrence check fails, *unifyOne* returns $\bot$ 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:

$$(\lambda\beta.\, \sigma(\beta)[\tau/\alpha])[\alpha \mapsto \tau]$$

The notation $\lambda x.\, e$ 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 $\alpha \mapsto \tau$ to all current mappings $\beta \mapsto \sigma(\beta)$ in σ using type variable substitution:

$$\begin{aligned} \sigma' &: \mathit{TVar} \rightharpoonup \mathit{Typ} \\ \sigma'(\beta) &= \sigma(\beta)[\tau/\alpha] \end{aligned}$$

HMBool

$$\frac{\alpha \text{ fresh}}{\Gamma \vdash b : \alpha \mid \{\alpha \doteq \texttt{Bool}\}}$$

HMNum

$$\frac{\alpha \text{ fresh}}{\Gamma \vdash n : \alpha \mid \{\alpha \doteq \texttt{Num}\}}$$

HMVar

$$\Gamma \vdash x : \Gamma(x) \mid \emptyset$$

HMAndOr

$$\frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \qquad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \qquad \mathit{bop} \in \{\texttt{\&\&}, \texttt{||}\} \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\, \mathit{bop}\, e_2 : \alpha \mid C_1 \cup C_2 \cup \{\alpha \doteq \texttt{Bool}, \alpha_1 \doteq \texttt{Bool}, \alpha_2 \doteq \texttt{Bool}\}}$$

HMArith

$$\frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \qquad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \qquad \mathit{bop} \in \{\texttt{+}, \texttt{*}\} \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\, \mathit{bop}\, e_2 : \alpha \mid C_1 \cup C_2 \cup \{\alpha \doteq \texttt{Num}, \alpha_1 \doteq \texttt{Num}, \alpha_2 \doteq \texttt{Num}\}}$$

HMIf

$$\frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \qquad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \qquad \Gamma \vdash e_3 : \alpha_3 \mid C_3 \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3 : \alpha \mid C_1 \cup C_2 \cup C_3 \cup \{\alpha_1 \doteq \texttt{Bool}, \alpha \doteq \alpha_2, \alpha \doteq \alpha_3\}}$$

HMCall

$$\frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \qquad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \alpha \mid C_1 \cup C_2 \cup \{\alpha_1 \doteq (\alpha_2 \Rightarrow \alpha)\}}$$

HMFun

$$\frac{\Gamma' = \Gamma[x \mapsto \alpha_1] \qquad \Gamma' \vdash e : \alpha_2 \mid C \qquad \alpha, \alpha_1 \text{ fresh}}{\Gamma \vdash \texttt{function(}x\texttt{)}\; e : \alpha \mid C \cup \{\alpha \doteq (\alpha_1 \Rightarrow \alpha_2)\}}$$

HMFunRec

$$\frac{\Gamma' = \Gamma[x \mapsto \alpha][y \mapsto \alpha_1] \qquad \Gamma' \vdash e : \alpha_2 \mid C \qquad \alpha, \alpha_1 \text{ fresh}}{\Gamma \vdash \texttt{function}\; x\texttt{(}y\texttt{)}\; e : \alpha \mid C \cup \{\alpha \doteq (\alpha_1 \Rightarrow \alpha_2)\}}$$

Figure 6.3: Typing constraint generation rules for the Hindley-Milner type system

```
$unify(C) = unify(C, \emptyset)$
$unify(\emptyset, \sigma) = \sigma$
$unify(C, \bot) = \bot$
$unify(\{\tau_1 \doteq \tau_2\} \cup C, \sigma) =$
    $\textbf{let } \sigma' = unifyOne(\tau_1 \doteq \tau_2, \sigma) \textbf{ in}$
    $\textbf{if } \sigma' = \bot \textbf{ then } \bot \textbf{ else } unify(C\sigma', \sigma')$
$unifyOne(\alpha \doteq \alpha, \sigma) = \sigma$
$unifyOne(\alpha \doteq \tau, \sigma) =$
    $\textbf{if } \alpha \in tv(\tau) \textbf{ then } \bot \textbf{ else } (\lambda\beta.\, \sigma(\beta)[\tau/\alpha])[\alpha \mapsto \tau]$
$unifyOne(\tau \doteq \alpha, \sigma) = unifyOne(\alpha = \tau, \sigma)$
$unifyOne((\tau_1 \Rightarrow \tau_2) \doteq (\tau_1' \Rightarrow \tau_2'), \sigma) =$
    $unify(\{\tau_1 \doteq \tau_1', \tau_2 \doteq \tau_2'\}, \sigma)$
$unifyOne(\mathtt{Num} \doteq \mathtt{Num}, \sigma) = \sigma$
$unifyOne(\mathtt{Bool} \doteq \mathtt{Bool}, \sigma) = \sigma$
$unifyOne(\tau_1 \doteq \tau_2, \sigma) = \bot$
```

Figure 6.4: A simple unification algorithm

Note that this definition of σ' is equivalent to the following definition which defines σ' in terms of an anonymous function:

$$\sigma' = \lambda\beta.\, \sigma(\beta)[\tau/\alpha]$$

Then σ'' is obtained from σ' by extending it with the new mapping $\alpha \mapsto \tau$:

$$\begin{aligned} \sigma'' &: TVar \rightharpoonup Typ \\ \sigma'' &= \sigma'[\alpha \mapsto \tau]. \end{aligned}$$

The case $(\tau_1 \Rightarrow \tau_2) \doteq (\tau_1' \Rightarrow \tau_2')$ 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 $C' = \{\tau_1 \doteq \tau_1', \tau_2 \doteq \tau_2'\}$. 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 $\tau_1 \doteq \tau_2$ where τ_1 and τ_2 cannot be unified, such as $\mathtt{Bool} = \mathtt{Num}$, 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 σ for a constraint $\alpha \doteq \tau$, we have to iterate over both the existing mappings in σ 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 = x => x;
id(true) ? id(1) : 0
```

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 *x* => *x*, which generates the following single typing constraint:

$$\alpha_{id} \doteq \alpha_x \Rightarrow \alpha_x$$

Here, α_{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 α_{id} for each usage of *id* in the body. Instead, we use a fresh copy of the type variable and generate a fresh copy of the constraints obtained from the defining expression for each usage of *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 *id* in the body of the declaration is consistent with the constraints imposed by *id*'s definition. However, different usages of *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. $\alpha_{id} \doteq \alpha_x \Rightarrow \alpha_x$

2. $\alpha_{id,1} \doteq \alpha_{\mathtt{true}} \Rightarrow \alpha_{call,1}$

3. $\alpha_{id,1} \doteq \alpha_{x,1} \Rightarrow \alpha_{x,1}$

4. $\alpha_{\mathtt{true}} \doteq \mathbf{Bool}$

5. $\alpha_{call,1} = \mathbf{Bool}$

6. $\alpha_{id,2} \doteq \alpha_1 \Rightarrow \alpha_{call,2}$

7. $\alpha_{id,2} \doteq \alpha_{x,2} \Rightarrow \alpha_{x,2}$

8. $\alpha_1 \doteq \texttt{Num}$

9. $\alpha_0 \doteq \texttt{Num}$

10. $\alpha_{call,1} \doteq \texttt{Bool}$

11. $\alpha_{ite} \doteq \alpha_{call,2}$

12. $\alpha_{ite} \doteq \alpha_0$

Note that the constraints 3 and 7 are the fresh copies of constraint 1 for the two usages of *id* in the body of the **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

```
const id = x => x;
id(true) ? id(1) : 0
```

with the expression

```
(id => id(true) ? id(1) : 0)(x => x)
```

From a semantic point of view the two expressions are equivalent. We have simply replaced the **const** declaration of *id* by function abstraction over *id* in the body of the declaration, followed by a call that immediately binds *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 *id* in the function body are bound to the same type variable. That is, the generated constraints require that *id* is at the same time of type

$$\texttt{Bool} \Rightarrow \texttt{Bool}$$

and of type

$$\texttt{Num} \Rightarrow \texttt{Num}$$

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 **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 **const** declarations in defining expressions. In fact, the type inference problem for the Hindley-Milner type system with **const** declarations is known to be 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 **`let`** declarations, which are like **`const`** declarations except that the keyword **`const`** is replaced by **`let`**:

$$\texttt{\textbf{let}}\ x = e_d\,;\ e_b$$

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`:

```
let x = 1;
x = x + 1;
x
```

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.

```
let x = 1;
x = x + 1
```

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

```
let x = 2;
let y = 2;
y = x = 3;
x + y
```

This program thus evaluates to 6.

At first, it may seem as if we could eliminate **let** 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:

```
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 **let** 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

```
let x = 2;
const f = 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 3. On the other hand, consider the program

```
const x = 2;
const f = 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 2[1].

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:

$$\begin{array}{rll}
n \in \mathit{Num} & & \text{numbers (double)}\\
b \in \mathit{Bool} ::= & \mathtt{true} \mid \mathtt{false} & \text{Booleans}\\
a \in \mathit{Addr} = & \mathbb{N} & \text{addresses}\\
x \in \mathit{Var} & & \text{variables}\\
\tau \in \mathit{Typ} ::= & \mathtt{Bool} \mid \mathtt{Num} \mid x{:}\tau_1 \Rightarrow \tau_2 & \text{types}\\
v \in \mathit{Val} ::= & n \mid b \mid a \mid \mathtt{function}\, p(x : \tau)\; t\; e & \text{values}\\
e \in \mathit{Expr} ::= & x \mid v \mid e_1\, \mathit{bop}\, e_2 \mid \mathit{uop}\, e_1 \mid e_1 \;\mathtt{?}\; e_2 \;\mathtt{:}\; e_3 \mid & \text{expressions}\\
 & \mathit{mut}\; x \,\mathtt{=}\, e_d \,\mathtt{;}\; e_b \mid e_1\mathtt{(}e_2\mathtt{)} & \\
\mathit{bop} \in \mathit{Bop} ::= & \mathtt{+} \mid \mathtt{*} \mid \mathtt{\&\&} \mid \mathtt{||} \mid \mathtt{===} \mid \mathtt{!==} \mid \mathtt{=} & \text{binary operators}\\
\mathit{uop} \in \mathit{Uop} ::= & \mathtt{*} & \text{unary operators}\\
p ::= & x \mid \epsilon & \text{function names}\\
t ::= & \mathtt{:}\tau \mid \epsilon & \text{return type annotations}\\
\mathit{mut} \in \mathit{Mut} ::= & \mathtt{const} \mid \mathtt{let} & \text{mutabilities}
\end{array}$$

Note that we introduce variable declarations by generalizing **const** declarations in our previous language to a declaration that is parameterized by a *mutability*, $mut \in Mut$. A mutability *mut* is either **let** or **const**. We introduce *addresses*, $a \in Addr$, as a new kind of values. An address denotes a location in memory. We define $Addr = \mathbb{N}$. However, the specific definition of *Addr* is immaterial. We only rely on the fact that *Addr* is an infinite set (i.e., we never run out of addresses when we need fresh ones for memory allocation). The

[1] Even if we assumed dynamic binding semantics, this program would still evaluate to 2.

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 `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 state M, expression e evaluates to value v and the new memory state M'". The presence of a memory state 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, $M : \mathit{Addr} \rightharpoonup \mathit{Val}$. We can then think of a **`let`** variable x as a **`const`** variable that stores an address to a memory location in M. Whenever we use a **`let`** 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.

$$\textsc{EvalAssignVar}\ \frac{\langle M, e\rangle \Downarrow \langle M', v\rangle \qquad a \in \mathsf{dom}(M')}{\langle M, \texttt{*}\ a\ \texttt{=}\ e\rangle \Downarrow \langle M'[a \mapsto v], v\rangle} \qquad \textsc{EvalDerefVar}\ \frac{a \in \mathsf{dom}(M)}{\langle M, \texttt{*}\ a\rangle \Downarrow \langle M, M(a)\rangle}$$

$$\textsc{EvalLetDecl}\ \frac{\begin{array}{c}\langle M, e_d\rangle \Downarrow \langle M_d, v_d\rangle \\ a \notin \mathsf{dom}(M_d) \qquad M' = M_d[a \mapsto v_d] \qquad \langle M', e_b[\texttt{*}\,a/x]\rangle \Downarrow \langle M'', v_b\rangle\end{array}}{\langle M, \texttt{let}\ x = v_d\,\texttt{;}\ e_b\rangle \Downarrow \langle M'', v_b\rangle}$$

Figure 7.1: Big-step operational semantics of imperative primitives

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 EvalLetDecl, EvalDerefVar, and EvalAssignVar. The rule EvalLetDecl handles **let** 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 $a \notin \mathsf{dom}(M_d)$. 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 $\texttt{*}\ a$. 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 $\texttt{*}\ a$ in the EvalLetDecl 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 mathematical model, but a production run-time system must somehow enable

$$\textsc{EvalVal}\qquad \langle M, v\rangle \Downarrow \langle M, v\rangle$$

$$\textsc{EvalPlus}\qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \qquad v = v_1 + v_2}{\langle M, e_1 \,\texttt{+}\, 2\rangle \Downarrow \langle M'', v\rangle}$$

$$\textsc{EvalTimes}\qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \qquad v = v_1 \cdot v_2}{\langle M, e_1 \,\texttt{*}\, e_2\rangle \Downarrow \langle M'', v\rangle}$$

$$\textsc{EvalConstDecl}\qquad \frac{\langle M, e_d\rangle \Downarrow \langle M_d, v_d\rangle \qquad e_b' = e_b[v_d/x] \qquad \langle M_d, e_b'\rangle \Downarrow \langle M', v_b\rangle}{\langle M, \mathbf{const}\ x \,\texttt{=}\, e_d; e_b\rangle \Downarrow \langle M', v_b\rangle}$$

$$\textsc{EvalEqual}\qquad \frac{\begin{array}{c}\langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \\ b = (v_1 = v_2) \qquad v_1 \neq \texttt{function}\, p_1(x_1)\, e_3 \qquad v_2 \neq \texttt{function}\, p_2(x_2)\, e_4\end{array}}{\langle M, e_1 \;\texttt{===}\; e_2\rangle \Downarrow \langle M'', b\rangle}$$

$$\textsc{EvalDisequal}\qquad \frac{\begin{array}{c}\langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \\ b = (v_1 \neq v_2) \qquad v_1 \neq \texttt{function}\, p_1(x_1)\, e_3 \qquad v_2 \neq \texttt{function}\, p_2(x_2)\, e_4\end{array}}{\langle M, e_1 \;\texttt{!==}\; e_2\rangle \Downarrow \langle M'', b\rangle}$$

$$\textsc{EvalIfThen}\qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', \texttt{true}\rangle \qquad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle}{\langle M, e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3\rangle \Downarrow \langle M'', v_2\rangle}$$

$$\textsc{EvalIfElse}\qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', \texttt{false}\rangle \qquad \langle M', e_3\rangle \Downarrow \langle M'', v_3\rangle}{\langle M, e_1 \;\texttt{?}\; e_2 \;\texttt{:}\; e_3\rangle \Downarrow \langle M'', v_3\rangle}$$

$$\textsc{EvalAndFalse}\qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', \texttt{false}\rangle}{\langle M, e_1 \,\texttt{\&\&}\, e_2\rangle \Downarrow \langle M', \texttt{false}\rangle}$$

$$\textsc{EvalAndTrue}\qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', \texttt{true}\rangle \qquad \langle M', e_2\rangle \Downarrow \langle M'' v_2\rangle}{\langle M, e_1 \,\texttt{\&\&}\, e_2\rangle \Downarrow \langle M'', v_2\rangle}$$

$$\textsc{EvalOrTrue}\qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', \texttt{true}\rangle}{\langle M, e_1 \,\texttt{||}\, e_2\rangle \Downarrow \langle M', \texttt{true}\rangle}$$

$$\textsc{EvalOrFalse}\qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', \texttt{false}\rangle \qquad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle}{\langle M, e_1 \,\texttt{||}\, e_2\rangle \Downarrow \langle M'', v_2\rangle}$$

$$\textsc{EvalCall}\qquad \frac{\begin{array}{c}\langle M, e_1\rangle \Downarrow \langle M', \texttt{function(}x\texttt{)}\, e\rangle \\ \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \qquad e' = e[v_2/x] \qquad \langle M'', e'\rangle \Downarrow \langle M''', v\rangle\end{array}}{\langle M, e_1\texttt{(}e_2\texttt{)}\rangle \Downarrow \langle M''', v\rangle}$$

$$\textsc{EvalCallRec}\qquad \frac{\begin{array}{c}\langle M, e_1\rangle \Downarrow \langle M', \texttt{function}\, x_1\texttt{(}x_2\texttt{)}\, e\rangle \\ \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \qquad e' = e[v_1/x_1][v_2/x_2] \qquad \langle M'', e'\rangle \Downarrow \langle M''', v\rangle\end{array}}{\langle M, e_1\texttt{(}e_2\texttt{)}\rangle \Downarrow \langle M''', v\rangle}$$

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 state and the omission of implicit type conversions.

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 most interesting new case is the assignment expressions e_1 = e_2. Specifically, the following expression should be considered well-typed:

$$\texttt{let}\ x = 3;\ x = 5$$

whereas the next expression should not be well-typed:

$$\texttt{const}\ x = 3;\ x = 5$$

That is, we should only allow assignments to variables that have actually been declared by a **let** 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 Γ as follows:

$$\Gamma : \mathit{Var} \rightharpoonup \mathit{Mut} \times \mathit{Typ}$$

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 Γ 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 **let** 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 **let** 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 τ. That is, we do not allow a variable x to be reassigned to a value whose type is different from the type of x's initialization expression. This is in contrast to JavaScript, which allows such

$$\begin{array}{c}\textsc{TypeBool}\\ \Gamma \vdash b : \texttt{Bool}\end{array} \qquad \begin{array}{c}\textsc{TypeNum}\\ \Gamma \vdash n : \texttt{Num}\end{array}$$

$$\textsc{TypeAndOr}\quad \frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \texttt{Bool} \qquad bop \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\, bop\, e_2 : \texttt{Bool}}$$

$$\textsc{TypeArith}\quad \frac{\Gamma \vdash e_1 : \texttt{Num} \qquad \Gamma \vdash e_2 : \texttt{Num} \qquad bop \in \{\texttt{+}, \texttt{*}\}}{\Gamma \vdash e_1\, bop\, e_2 : \texttt{Num}}$$

$$\textsc{TypeEqual}\quad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \neq x\!:\!\tau_1 \Rightarrow \tau_2 \qquad bop \in \{\texttt{===}, \texttt{!==}\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{Bool}}$$

$$\textsc{TypeIf}\quad \frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ \texttt{?}\ e_2\ :\ e_3 : \tau} \qquad \textsc{TypeCall}\quad \frac{\Gamma \vdash e_1 : \tau' \Rightarrow \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \tau}$$

Figure 7.3: Type checking rules for non-imperative primitives (no changes compared to Figure 6.2)

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 $a \in \mathit{Addr}$ and dereference operations $\texttt{*}\ e$ 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.

$$\frac{\Gamma \vdash e_d : \tau_d \qquad \Gamma' = \Gamma[x \mapsto \langle \mathit{mut}, \tau_d \rangle] \qquad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash \mathit{mut}\ x = e_d\,;\, e_b : \tau_b}\ \textsc{TypeDecl}$$

$$\frac{x \in \mathsf{dom}(\Gamma) \qquad \Gamma(x) = \langle \mathit{mut}, \tau \rangle}{\Gamma \vdash x : \tau}\ \textsc{TypeVar} \qquad \frac{\Gamma(x) = (\mathbf{let}, \tau) \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x = e : \tau}\ \textsc{TypeAssignVar}$$

$$\frac{\Gamma' = \Gamma[x \mapsto \langle \mathbf{const}, \tau \rangle] \qquad \Gamma' \vdash e : \tau'}{\Gamma \vdash \mathbf{function}(x\!:\!\tau)\ e : x\!:\!\tau \Rightarrow \tau'}\ \textsc{TypeFun} \qquad \frac{\Gamma' = \Gamma[x \mapsto \langle \mathbf{const}, \tau \rangle] \qquad \Gamma' \vdash e : \tau'}{\Gamma \vdash \mathbf{function}(x\!:\!\tau)\ \!:\!\tau'\ e\ \ : \tau \Rightarrow \tau'}\ \textsc{TypeFunAnn}$$

$$\frac{\Gamma' = \Gamma[x_1 \mapsto \langle \mathbf{const}, \tau_1 \rangle][x_2 \mapsto \langle \mathbf{const}, \tau_2 \rangle] \qquad \Gamma' \vdash e : \tau' \qquad \tau_1 = \tau_2 \Rightarrow \tau'}{\Gamma \vdash \mathbf{function}\ x_1(x_2\!:\!\tau_2)\ \!:\!\tau'\ \ : \tau_1}\ \textsc{TypeFunRec}$$

Figure 7.4: Type checking rules for imperative primitives

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 **const**. As an example, consider the following program, which defines a function f with a call-by-value parameter x:

```
const f = (const x: Num) => 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 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 = (const x: Num) => x + x;
let 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 JakartaScript, we indicate call-by-name parameters by prefixing the parameter with the keyword **`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:

```
const f = (name x: Num) => x + x;
const y = 3;
const r = f(y + 1);
console.log(y);
console.log(r);
```

This program will print `3` and `8` just like in the version with the call-by-value parameter. Again, we can 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:

```
const f = (name x: Num) => x + x;
let y = 3;
const r = f(y = y + 1);
console.log(y);
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 of 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:

```
const f = (const b: Bool) =>
   (name x: Num) => b ? x + x : 0;
let y = 3;
const r = f(false)(y = y + 1);
console.log(y);
console.log(r);
```

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:

```
const f = x => (x() + x());
let y = 3;
const r = f(() => 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 Num 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 evaluation of the function body, and parameters that are essentially treated like mutable variables and can be reassigned new values. We refer to the latter type of parameters as *call-by-variable* parameters. We indicate such parameters by prefixing the parameter name with the keyword **let**. Essentially, the difference between a **const** parameter and a **let** parameter is that inside the function body, a **let** parameter is treated as if it was declared by a **let** declaration instead of a **const** declaration. The following example highlights this difference:

```
const f = (let x: Num) => (x = x + 1, x);
let y = 3;
const r = f(y);
console.log(y);
console.log(r);
```

This program will print `3` and `4`. The parameter variable `x` is reassigned inside of the body of `f`. However, the effect of this assignment is not observable outside of `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.

Call by Reference

Finally, we can consider a combination of call by name and call by variable. We refer to this mode as *call by reference*. This mode is indicated by the keyword **ref**. The following program highlights the difference between call by variable and call by reference:

```
const f = (ref x: Num) => (x = x + 1, x);
let y = 3;
const r = f(y);
console.log(y);
console.log(r);
```

This program will print `4` and `4`. Unlike in the previous program where we passed the argument to `f` by variable, the assignment to `x` is now treated as an assignment to the mutable variable `y`, which is passed to `x` in the call to `f`. Note that we can only pass assignable expressions as arguments to call-by-reference parameters. We refer to such expressions as *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 `f` attempts to reassign `y` which has been declared as an immutable **const** variable:

```
const f = (ref x: Num) => (x = x + 1, x);
const y = 3;
const r = f(y); // type error because y is not assignable
console.log(y);
console.log(r);
```

7.2.2 A Simple Language with Parameter Passing Modes

The abstract syntax of our extended language is as follows:

$$\begin{array}{rll}
n \in \mathit{Num} & & \text{Nums (double)} \\
b \in \mathit{Bool} ::= & \texttt{true} \mid \texttt{false} & \text{Booleans} \\
a \in \mathit{Addr} = & \mathbb{N} & \text{addresses} \\
x \in \mathit{Var} & & \text{variables} \\
\tau \in \mathit{Typ} ::= & \texttt{Bool} \mid \texttt{Num} \mid (\mathit{mode}\ \tau_1) \Rightarrow \tau_2 & \text{types} \\
v \in \mathit{Val} ::= & n \mid b \mid a \mid \texttt{function}\ p(\mathit{mode}\ x : \tau)\ t\ e & \text{values} \\
e \in \mathit{Expr} ::= & x \mid v \mid e_1\ \mathit{bop}\ e_2 \mid \mathit{uop}\ e_1 \mid e_1 \ ?\ e_2 \ :\ e_3 \mid & \text{expressions} \\
 & \mathit{mut}\ x = e_d\,;\, e_b \mid e_1(e_2) & \\
\mathit{bop} \in \mathit{Bop} ::= & \texttt{+} \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{=} & \text{binary operators} \\
\mathit{uop} \in \mathit{Uop} ::= & \texttt{*} & \text{unary operators} \\
p ::= & x \mid \epsilon & \text{function names} \\
t ::= & :\tau \mid \epsilon & \text{type annotations} \\
\mathit{mut} \in \mathit{Mut} ::= & \texttt{const} \mid \texttt{let} & \text{mutabilities} \\
\mathit{mode} \in \mathit{PMode} ::= & \texttt{const} \mid \texttt{let} \mid \texttt{name} \mid \texttt{ref} & \text{passing modes}
\end{array}$$

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. 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 $\texttt{*}\ a$. In this case, the argument $\texttt{*}\ a$ 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 TypeFun rules, the given parameter passing mode is mapped to the appropriate mutability using the function *mut*. 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.

$$\frac{\begin{array}{c}\textsc{EvalCallVal}\\ \langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad v_1 = (\texttt{const}\ x\,{:}\,\tau_2)\ \texttt{=>}\ e \\ \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \qquad e' = e[v_2/x] \qquad \langle M'', e'\rangle \Downarrow \langle M''', v\rangle\end{array}}{\langle M, e_1\texttt{(}e_2\texttt{)}\rangle \Downarrow \langle M''', v\rangle}$$

$$\frac{\begin{array}{c}\textsc{EvalCallValRec}\\ \langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad v_1 = \texttt{function}\ x_1\texttt{(const}\ x_2\,{:}\,\tau_2\texttt{)}\ \texttt{:}\tau\ e \\ \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \qquad e' = e[v_1/x_1][v_2/x_2] \qquad \langle M'', e'\rangle \Downarrow \langle M''', v\rangle\end{array}}{\langle M, e_1\texttt{(}e_2\texttt{)}\rangle \Downarrow \langle M''', v\rangle}$$

$$\frac{\begin{array}{c}\textsc{EvalCallVar}\\ \langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad v_1 = (\texttt{let}\ x\,{:}\,\tau_2)\ \texttt{=>}\ e \qquad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \\ a \notin \mathsf{dom}(M'') \qquad e' = e[\star\ a/x] \qquad \langle M''[a \mapsto v_2], e'\rangle \Downarrow \langle M''', v\rangle\end{array}}{\langle M, e_1\texttt{(}e_2\texttt{)}\rangle \Downarrow \langle M''', v\rangle}$$

$$\frac{\begin{array}{c}\textsc{EvalCallVarRec}\\ \langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad v_1 = \texttt{function}\ x_1\texttt{(let}\ x_2\,{:}\,\tau_2\texttt{)}\ \texttt{:}\tau\ e \qquad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \\ a \notin \mathsf{dom}(M'') \qquad e' = e[v_1/x_1][\star\ a/x_2] \qquad \langle M''[a \mapsto v_2], e'\rangle \Downarrow \langle M''', v\rangle\end{array}}{\langle M, e_1\texttt{(}e_2\texttt{)}\rangle \Downarrow \langle M''', v\rangle}$$

$$\frac{\begin{array}{c}\textsc{EvalCallName}\\ \langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad v_1 = (\texttt{name}\ x\,{:}\,\tau_2)\ \texttt{=>}\ e \qquad e' = e[e_2/x] \qquad \langle M', e'\rangle \Downarrow \langle M'', v\rangle\end{array}}{\langle M, e_1\texttt{(}e_2\texttt{)}\rangle \Downarrow \langle M'', v\rangle}$$

$$\frac{\begin{array}{c}\textsc{EvalCallNameRec}\\ \langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \\ v_1 = \texttt{function}\ x_1\texttt{(name}\ x_2\,{:}\,\tau_2\texttt{)}\ \texttt{:}\tau\ e \qquad e' = e[v_1/x_1][e_2/x_2] \qquad \langle M', e'\rangle \Downarrow \langle M'', v\rangle\end{array}}{\langle M, e_1\texttt{(}e_2\texttt{)}\rangle \Downarrow \langle M'', v\rangle}$$

$$\frac{\begin{array}{c}\textsc{EvalCallRef}\\ \langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \qquad v_1 = (\texttt{ref}\ x\,{:}\,\tau_2)\ \texttt{=>}\ e \qquad e' = e[\star\ a/x] \qquad \langle M', e'\rangle \Downarrow \langle M'', v\rangle\end{array}}{\langle M, e_1\texttt{(}\ \star\ a\texttt{)}\rangle \Downarrow \langle M'', v\rangle}$$

$$\frac{\begin{array}{c}\textsc{EvalCallRefRec}\\ \langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \\ v_1 = \texttt{function}\ x_1\texttt{(ref}\ x_2\,{:}\,\tau_2\texttt{)}\ \texttt{:}\tau\ e \qquad e' = e[v_1/x_1][\star\ a/x_2] \qquad \langle M', e'\rangle \Downarrow \langle M'', v\rangle\end{array}}{\langle M, e_1\texttt{(}\ \star\ a\texttt{)}\rangle \Downarrow \langle M'', v\rangle}$$

Figure 7.5: New inference rules that define the big-step semantics of function call expressions for the different parameter passing modes

$$\begin{array}{l}\textsc{TypeCall}\\ \dfrac{\Gamma \vdash e_1 : \texttt{(}\mathit{mode}\ \tau_2\texttt{)} \Rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2 \qquad \mathit{mode} \notin \{\textbf{ref}\}}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \tau}\end{array}$$

$$\begin{array}{l}\textsc{TypeCallRef}\\ \dfrac{\Gamma \vdash e : \texttt{(}\textbf{ref}\ \tau_2\texttt{)} \Rightarrow \tau \qquad x \in \mathsf{dom}(\Gamma) \qquad \Gamma(x) = (\textbf{let}, \tau_2)}{\Gamma \vdash e\texttt{(}x\texttt{)} : \tau}\end{array}$$

$$\begin{array}{l}\textsc{TypeFun}\\ \dfrac{\Gamma' = \Gamma[x \mapsto (\mathit{mut}(\mathit{mode}), \tau_2)] \qquad \Gamma' \vdash e : \tau}{\Gamma \vdash \texttt{(}\mathit{mode}\ x\texttt{:}\tau_2\texttt{)}\ \texttt{=>}\ e\ : \texttt{(}\mathit{mode}\ \tau_2\texttt{)} \Rightarrow \tau}\end{array}$$

$$\begin{array}{l}\textsc{TypeFunAnn}\\ \dfrac{\Gamma' = \Gamma[x \mapsto (\mathit{mut}(\mathit{mode}), \tau_2)] \qquad \Gamma' \vdash e : \tau}{\Gamma \vdash \textbf{function}\texttt{(}\mathit{mode}\ x\texttt{:}\tau_2\texttt{)}\ \texttt{:}\tau\ e\ : \texttt{(}\mathit{mode}\ \tau_2\texttt{)} \Rightarrow \tau}\end{array}$$

$$\begin{array}{l}\textsc{TypeFunRec}\\ \dfrac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto (\textbf{const}, \tau_1)][x_2 \mapsto (\mathit{mut}(\mathit{mode}), \tau_2)]\\ \Gamma' \vdash e : \tau \qquad \tau_1 = \texttt{(}\mathit{mode}\ \tau_2\texttt{)} \Rightarrow \tau\end{array}}{\Gamma \vdash \textbf{function}\ x_1\texttt{(}\mathit{mode}\ x_2\texttt{:}\tau_2\texttt{)}\ \texttt{:}\tau\ e\ : \tau_1}\end{array}$$

Figure 7.6: New type checking rules for function and call expressions

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 JakartaScript fragment as well as in Scala.

JakartaScript while loop. Loops are one of the most important control constructs in imperative languages. We do not have loops built into our current JakartaScript fragment. However, using call-by-name parameter passing, we can write recursive functions that can be used almost as if they were loops:

```
const while = function while(name cond: Bool):
  (name Undefined) => Undefined
  {
    return (name body: Undefined) =>
      cond ? (body, while(cond)(body)) : undefined
  };

let x = 0;
while(x < 10)({
  console.log(x);
  x = x + 1;
  undefined
})
```

In the above JakartaScript program, we define a curried function `while` that takes a value of type `Bool`, 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:

```
let x = 0;
while(x < 10)({
  console.log(x);
  x = x + 1;
})
```

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 **`Num`**. 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:

```
  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:

```
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:

```
class repeat(body: => Unit):
  def until(cond: => Boolean): Unit =
    body
    if (!cond) until(cond)

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 JAKARTASCRIPT. 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:

```
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:

```
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 input 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 evaluation relation.

7.3.1 A Stateful JakartaScript Interpreter

We start our exposition with a straightforward Scala implementation of the big-step operational semantics that we formalized in Section 7.1.2. 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[Addr, 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 evaluation relation $M, e \Downarrow M', v$ from Section 7.1 by a function `eval` with the following signature:

```
  def eval(m: Mem, e: Expr): (Mem, Val)
```

That is, `eval` 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. We refer to such a computation as being *stateful* since it depends on an input state and produces an output state in addition to the actual result value.

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 EvalVal, EvalUMinus, and EvalPlus rule. The code closely follows the corresponding inference

```
/** Mutabilities */
enum Mut:
  case MConst, MLet

/** Binary operators */
enum Bop:
  case Times, Plus, Assign

/** Unary operators */
enum Uop:
  case Deref

/** Expressions */
type Params = List[(String, (PMode, Typ))]
enum Expr:
  ...
  /** Variable declarations
  case Decl(mut: Mut, x: String, ed: Expr, eb: Expr)

  /** Binary operator expressions */
  case BinOp(bop: Bop, e1: Expr, e2: Expr)

  /** Unary operators expressions */
  case UnOp(uop: Uop, e: Expr)

  /** Addresses */
  case Addr (addr: Int)

/** Values */
type Val = Addr | Num | ...

/** 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(mp, 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

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> val l = List(2,5)
l: List[Int] = List(2,5)

scala> for (x <- l) yield x + 1
```

```
res0: List[Int] = List(3,6)
```

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

```
  for (x <- l) 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 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:

```
scala> for (x <- l; y <- l) yield (x, y)
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:

```
scala> l map (x => l map (y => (x, y)))
res2: List[List[(Int, 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:

```
scala> res2.flatten
res3: List[(Int, 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:

```
scala> l flatMap (x => l map (y => (x, y)))
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

```
for (xn <- en; ...; x2 <- e2; x1 <- e1) yield e0
```

to the expression

```
en flatMap (xn =>...e2 flatMap (x2 => e1 map (x1 => e0))...)
```

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> for (x <- Some(0)) yield x + 1
res5: Option[Int] = Some(1)

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

A Monadic Interpreter

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 eval 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.

```
def eval(e: Expr): Mem => (Mem, Val) =
  def eToNum(e: Expr): Mem => (Mem, Int) =
    m =>
      val (mp, v) = eval(e)(m)
      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

Curried Interpreter. The transformation that we apply to the interpreter is as follows: we modify the signature of the eval function from:

```
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 stateful 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.

A call sm.map(f) to the map method of a state monad sm of type State[S,R] transforms sm into a State[S,P] by composing a stateless computation f: R=>P with the stateful computation run: S=>(S,R) encapsulated in sm. The stateful computation that results from this composition is stored in the run field of the resulting state monad and is visualized in the upper half of Figure 7.11.

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).

```
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)
      val p = f(r)
      (sp, p)
    }

  def flatMap[P](f: R => State[S,P]): State[S,P] =
    new State[S,P]{ s =>
      val (sp, r) = run(s)
      val (spp, p) = f(r)(sp) // same as f(r).apply(s)
      (spp, p)
    }

object State:
  def apply[S, R](f: S => (S, R)): State[S,R] =
    new State(f)
  def init[S]: State[S,S] =
    State( s => (s, s) )
  def insert[S,R](r: R): State[S,R] =
    init map ( s => (s, r) )
  def read[S,R](f: S => R): State[S,R] =
    init map ( s => (s, f(s)) )
  def write[S](f: S => S): State[S,Unit] =
    init map ( s => (f(s), ()) )
:
```

Figure 7.10: Implementation of the state monad

That is, the implementation of EvalUMinus corresponds to a call to `map` on the result of `eToNum(e1)` encapsulated in a state monad:

```
eToNum(e1).map(n1 => Num(- n1))
```

The resulting state monad is depicted in Figure 7.12.

Similar to `map`, the `flatMap` method transforms the given `State[S,R]` into a `State[S,P]` by composing its `run` function with another stateful computation produced by the function `f`. The stateful computation resulting from `f` is also encapsulated in a state monad of type `State[S,P]`. The composition of the `run` functions of these state monads that is performed by `flatMap` is visualized in the lower half of Figure 7.11.

If we again carefully analyze the code in Figure 7.9, we observe that the case implementing the rule for EvalPlus corresponds to a call to `flatMap` on the result of `etoNum(e1)` encapsulated in a `State[Mem,Val]`. Here, `eToNum(e1)` takes the role of `run` in `flatMap` and `f` is the function

```
n1 =>
  new State[Mem, Val]{mp =>
    val (mpp, n2) = eToNum(e2)(mp)
    (mpp, Num(n1 + n2))
  }
```

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 7.13. 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 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 7.14.

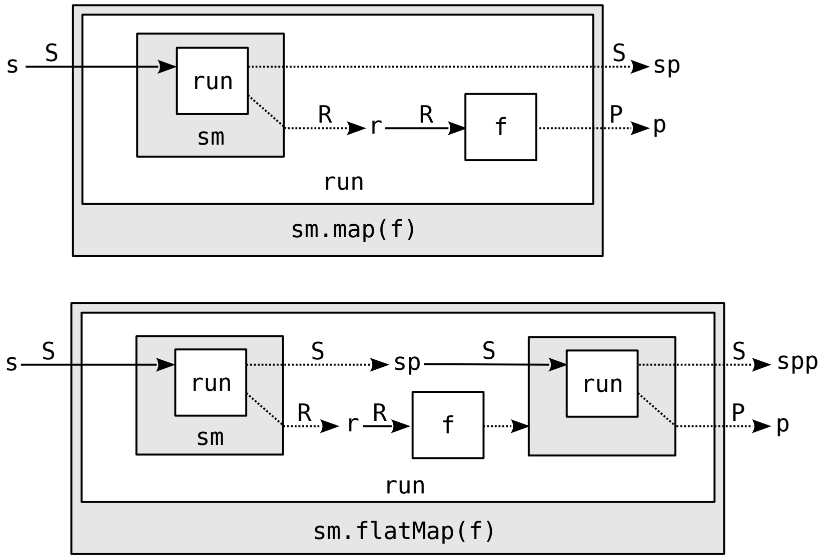


Figure 7.11: Visualization of the state monads constructed by calls to `map` respectively `flatMap` on a state monad `sm` of type `State[S,R]`. In both cases, the resulting state monad is of type `State[S,P]`. White boxes indicate functions and gray boxes indicate state monads with their `run` functions inside. The arrows indicate inputs and outputs of the functions and are labeled by the types of the corresponding values. Outputs are dotted arrows and inputs solid arrows.

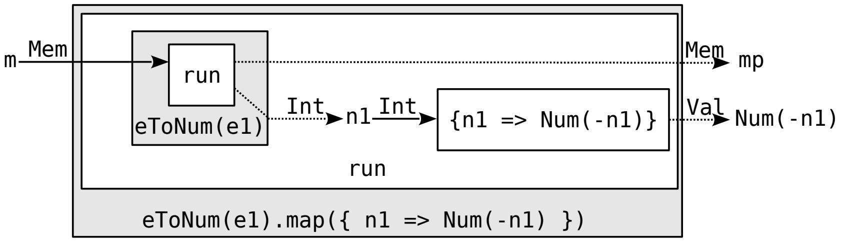


Figure 7.12: Visualization of the state monad `eToNum(e1).map(n1 =>Num(-n1))`.

```
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 7.13: 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
        n1 <- 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 7.14: Variant of the interpreter shown in Figure 7.13 with **for** expressions

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. We will also extend the type system with basic support for objects. We first describe the semantics of the new language constructs informally.

8.1.1 A Simple Typed Language with Objects

An object literal is a comma-separated sequence of field names with mutabilities (**`const`** or **`let`**) and initialization expressions surrounded by braces:

$$\{mut_1\ f_1 : e_1, \ldots, mut_k\ f_k : e_k\}$$

When an object literal is evaluated, we create an *object value.* 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 at a fresh address a in memory and the object literal itself evaluates to the address a. For example, the object literal `{ let f: 2 + 3, let g: 1 < 2 }` will create the object value `{ f: 5, g: true }` and store it at a fresh address a, which will then be the result value of the object literal. Note that we omit the mutability information of the fields in the created object. This information will only be needed during type checking.

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. For example if e evaluates to the address of an object `{ f: 5, g: true }`, then e`.f` evaluates to `5`.

A field of an object is mutable if it was prefixed with the mutability **`let`** in the object literal from which the object was created. On the other hand, if the field was 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 **`let`** mutability. For compatibility with JavaScript we make the field mutability modifiers optional and the mutability of a field defaults to **`let`** if the modifier is omitted. We add **`const`** fields to our language because they give us more flexibility once we extend the type system to deal with objects.

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:

```
const x = { let 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:

$$
\begin{array}{rcll}
n \in & \mathit{Num} & & \text{numbers (double)} \\
b \in & \mathit{Bool} & ::= \mathbf{true} \mid \mathbf{false} & \text{Booleans} \\
a \in & \mathit{Addr} & = \mathbb{N} & \text{addresses} \\
x \in & \mathit{Var} & & \text{variables} \\
f \in & \mathit{Fld} & & \text{field names} \\
\tau \in & \mathit{Typ} & ::= \mathtt{Bool} \mid \mathtt{Num} \mid x : \tau_1 \Rightarrow \tau_2 \mid & \\
& & \{\mathit{mut}_1\ f_1 : \tau_1, \ldots, \mathit{mut}_k\ f_k : \tau_k\} & \text{types} \\
v \in & \mathit{Val} & ::= n \mid b \mid a \mid \mathtt{function}\, p(x : \tau)\, t\ e & \text{values} \\
e \in & \mathit{Expr} & ::= x \mid v \mid e_1\, \mathit{bop}\, e_2 \mid \mathit{uop}\, e_1 \mid e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \mid & \text{expressions} \\
& & \mathit{mut}\ x = e_d\,;\, e_b \mid e_1(e_2) \mid & \\
& & e\,.\,f \mid \{\mathit{mut}_1\ f_1 : e_1, \ldots, \mathit{mut}_k\ f_k : e_k\} & \\
\mathit{bop} \in & \mathit{Bop} & ::= \texttt{+} \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{=} & \text{binary operators} \\
\mathit{uop} \in & \mathit{Uop} & ::= \texttt{*} & \text{unary operators} \\
& p & ::= x \mid \epsilon & \text{function names} \\
\mathit{mut} \in & \mathit{Mut} & ::= \mathbf{const} \mid \mathbf{let} & \text{mutabilities}
\end{array}
$$

8.1.2 Operational Semantics

We formalize objects o as partial functions from fields to values:

$$o \in \mathit{Obj} = \mathit{Fld} \rightharpoonup \mathit{Val} \ .$$

For an object o and field $f \in \mathsf{dom}(o)$ we write $o.f$ for the value $o(f)$.

Observe that the new language syntax only contains object literals, but not objects. Objects only enter the picture at run-time, i.e., when we evaluate an expression. However, they only do so indirectly. In particular, the value that an object literal evaluates to is an addresses rather than an object. The associated object is instead stored in the memory state M and accessed indirectly via its address.

This means that a memory state M is now a partial function from addresses to values and objects. We formalize this by introducing a semantic domain of memory contents:

$$\mathit{Con} ::= v \mid o$$

and redefine memory states as follows:

$$M \in \mathit{Mem} = \mathit{Addr} \rightharpoonup \mathit{Con} \ .$$

We do not track any information about the mutability of fields in objects. This is because the type system that we will discuss in Section 8.3 will statically ensure that immutable fields cannot be reassigned. Hence, we can safely erase

$$\textsc{EvalObj}\quad \frac{\begin{array}{c} M_0 = M \qquad \forall i \in [1, n] : \langle M_{i-1}, e_i \rangle \Downarrow \langle M_i, v_i \rangle \\ a \notin \mathsf{dom}(M_n) \qquad M' = M_n[a \mapsto \{f_1 \mapsto v_1\,, \ldots, f_n \mapsto v_n\}] \end{array}}{\langle M, \{mut_1\ f_1 : e_1\,, \ldots, mut_n\ f_n : e_n\} \rangle \Downarrow \langle M', a \rangle}$$

$$\textsc{EvalDerefFld}\quad \frac{\langle M, e \rangle \Downarrow \langle M', a \rangle \qquad M'(a) = o \qquad f \in \mathsf{dom}(o)}{\langle M, e\,.\,f \rangle \Downarrow \langle M', o.f \rangle}$$

$$\textsc{EvalAssignFld}\quad \frac{\begin{array}{c} \langle M, e_1 \rangle \Downarrow \langle M', a \rangle \\ \langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle \qquad M''(a) = o \qquad f \in \mathsf{dom}(o) \qquad M''' = M''[a \mapsto o[f \mapsto v_2]] \end{array}}{\langle M, e_1\,.\,f \texttt{ = } e_2 \rangle \Downarrow \langle M''', v_2 \rangle}$$

Figure 8.1: New evaluation rules for the language primitives related to objects

the mutability information once we have determined that the program is well-typed.

Figure 8.1 shows the big-step evaluation rules for the new language constructs. 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 EvalAssignFld formalizes the assignment to a field by updating the corresponding field of the object 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 memory states now store both values and objects.

8.1.3 Typing Objects

We also extend our type system to account for the new language constructs. Recall that the goal of the type system is to ensure that a well-typed program does not get stuck during evaluation. There are several situations where the evaluation of the new constructs can get stuck. For instance, consider the following simple program:

```
const x = {f: 1};
x.g
```

The evaluation of this program would get stuck at the point when the expression `x.g` is evaluated because the rule EvalDerefFld demands that the field `g` exists in the object that is pointed to by the address that `x` evaluates to. However, this object only has a field `f` but not a field `g`.

To account for this and similar situations where the program evaluation may get stuck, we extend our language of type expressions with *object types*:

$$\tau \in \mathit{Typ} ::= \ldots \mid \{mut_1\ f_1 : \tau_1\,, \ldots, mut_n\ f_n : \tau_n\}$$

$$\textsc{TypeObj}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash \{mut_1\ f_1 {:} e_1, \dots, mut_k\ f_k {:} e_k\} : \{mut_1\ f_1 {:} \tau_1, \dots, mut_n\ f_n {:} \tau_k\}}$$

$$\textsc{TypeDerefFld}\quad \frac{\Gamma \vdash e : \{mut_1\ f_1 {:} \tau_1, \dots, mut_k\ f_k {:} \tau_k\} \qquad f = f_i \qquad \tau = \tau_i \qquad i \in [1, k]}{\Gamma \vdash e.f : \tau}$$

$$\textsc{TypeAssignFld}\quad \frac{\Gamma \vdash e_1 : \{\dots \mathtt{\textbf{let}}\ f {:} \tau \dots\} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1.f \texttt{ = } e_2 : \tau}$$

Figure 8.2: Typing rules for objects

The syntax of object types is like that of object literals, except that the field initialization expressions are replaced by types. Intuitively, an object type describes the set of all addresses a that the program may generate during evaluation, such that a points to an object in memory that has fields $f_1, \dots, f_k$ storing values of types $\tau_1, \dots, \tau_k$, and this object was initially created from an object literal prescribing the field mutabilities $mut_1, \dots, mut_k$.

The rules that extend our typing relation $\Gamma \vdash e : \tau$ to account for objects are given in Figure 8.2. The rule TypeObj infers the object type of an object literal $\{\overline{mut\ f : e}\}$ by recursively inferring the types of the initialization expressions e of the fields f. The rules for field dereference, TypeDerefFld, and assignment to a field, TypeAssignFld, are straightforward. Note that 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 **`let`**. This requirement is analogous to the rule TypeAssignVar for typing assignments to variables.

8.2 Encoding Essential OO Language Features

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 ignore types in this section and use the concrete syntax of JavaScript so that you can easily experiment with 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 get that reads the current counter value, and a method inc that increments the counter. We can declare such an counter in our language using an object that has two fields holding functions that implement the get and inc methods:

```
const rep = { x: 0 };
const counter = {
    get: () => rep.x,
    inc: () => rep.x = rep.x + 1
  };
```

You may wonder why we declared the field x that holds the counter value in a separate object, rep, instead of adding it directly to the counter object. There are two reasons for this. First, if x was a field of counter, then any client who has a reference to 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 counter object in another object rep that is read and written by the get and inc methods. A client of the counter object who does not have a direct reference to the rep object will not be able to modify the state of counter directly (i.e., without calling the inc method).

The second reason why we cannot add the field x to counter directly is that the methods inc and 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 **this** (or self). The **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 inc three times and then returns the new counter value:

```
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 `rep` to the counter representation. Hence, clients can still bypass 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. For most languages, the class mechanism tends to be rather complicated because it is 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:

```
const counterClass = function(rep) {
    return {
      get: () => rep.x,
      inc: () => rep.x = rep.x + 1
    }
  };
```

Using the function `counterClass` we can now define a *constructor* that creates new counter objects on demand:

```
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`:

```
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* mechanism 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:

```
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 *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:

```
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 subclasses 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:

```
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 subtyping to our object-oriented version of JakartaScript.

Extending the Representation and Method Overriding. When we derive a subclass from a superclass we are not limited to adding 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: () => rep.x = rep.y,
      backup: () => 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: () => (_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:

```
const setCounterClass = function(_this) {
    _this.x = 0;
    _this.get = () => _this.x;
    _this.set = (i) => _this.x = i;
    _this.inc = () => _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:

```
const newSetCounter = function() {
    const dummy = {
      x: 0,
      get: () => 0,
      set: i => undefined,
      inc: () => undefined };
    return setCounterClass(dummy);
  };
```

8.3 Subtyping

In this section, we will extend the type system of JakartaScript to provide typing rules that formalize our intuition of the substitution principle in object-oriented programs.

Substitution Principle Revisited. Consider the following simple program[1]:

```
const fun = (x: {f: Num}) => 2 * x.f;
fun({f: 1})
```

In this program it should be safe to replace the argument expression `{f:1}` of the call to function `fun` by the expression `{f:1,g:2}`. The new program can still be safely evaluated. In fact, it will evaluate to exactly the same value. However, the call `fun({f:1,g:2})` is not well-typed according to our current typing rules. The problem is the rule 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:1,g:2})`, since the argument has type `{f: Num, g: Num}` whereas the parameter of function `fun` has type `{f: Num}`. In the remainder of this section, we will extend our typing rules so that the program with the modified call to `fun` and similar programs are well-typed.

8.3.1 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 τ' can be safely substituted by values of type τ. Informally, this means that all the things that can be done with values of type τ' can also be done with values of type τ. If $\tau <: \tau'$ holds, we say that τ is a *subtype* of τ' or, conversely, that τ' is a *supertype* of τ.

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 τ and τ is a subtype of τ', then e also has type τ'. 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

[1] Recall that fields without explicit mutability modifier have **`let`** mutability by default.

$$\begin{array}{l}\textsc{TypeSub}\\ \dfrac{\Gamma \vdash e : \tau' \qquad \tau' <: \tau}{\Gamma \vdash e : \tau}\end{array}$$

Figure 8.3: The subtyping rule

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.

For technical purposes, it will be useful to have a *largest* and a *smallest* type in our type system with respect to the subtype relation. We therefore extend our language of type expressions with two additional types:

$$\tau ::= \ldots \mid \texttt{\textbf{Any}} \mid \texttt{\textbf{Nothing}}\ .$$

Intuitively, the type **`Any`** represents the set of all values whereas **`Nothing`** represents the empty set of values.

Scala also has a smallest and a largest type, which are also called `Nothing` and `Any`, respectively. Java does not have a smallest type or largest type. However, Java has an object type that is a supertype of all other object types (but not primitive types like `int`), namely the type `Object`.

Our subtype relation is then defined by the rules in Figure 8.4. In the following, we discuss the individual rules in detail.

General properties of subtyping. We start with some basic properties of the subtype relation. First, we want to capture the property every type should be a subtype of itself, i.e., the subtype relation is reflexive. This is captured by the rule SubRefl. Similarly, the subtype relation should be transitive, i.e., if τ_1 is a subtype of τ_2 and τ_2 a subtype of τ_3, then τ_1 should also be a subtype of τ_3. This is captured by the rule SubTrans.

Next, we discuss some properties of the types **`Any`** and **`Nothing`**. Since **`Any`** stands for the set of all values, it is a supertype of any other type. This property is captured by the rule SubAny. Conversely, **`Nothing`** stands for the empty set of values, so it is a subtype of any other type (rule SubNothing).

From the perspective of writing programs, the types **`Any`** and **`Nothing`** are not very useful on their own. For example, if we determine that the type of an expression is **`Any`**, then the only way we can interact with the resulting value is to pass it around. For example, we can safely pass such a value to a function that expects a value of type **`Any`**. However, that function won't be able to do anything with that value other than returning it or passing it to other functions that expect a value of type **`Any`**.

Similarly, we will never be able to infer that an expression has type **`Nothing`** because no value belongs to that type. Technically, if our type system was able

SubRefl

$$\tau <: \tau$$

SubTrans

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

SubAny

$$\tau <: \texttt{\textbf{Any}}$$

SubNothing

$$\texttt{\textbf{Nothing}} <: \tau$$

SubObjWidth

$$\frac{}{\{\overline{\mathit{mut}\ f : \tau}, \mathit{mut}'\ g : \tau'\} <: \{\overline{\mathit{mut}\ f : \tau}\}}$$

SubObjDepth

$$\frac{\tau <: \tau'}{\{\ldots \texttt{\textbf{const}}\ f : \tau \ldots\} <: \{\ldots \texttt{\textbf{const}}\ f : \tau' \ldots\}}$$

SubObjMut

$$\frac{}{\{\ldots \texttt{\textbf{let}}\ f : \tau \ldots\} <: \{\ldots \texttt{\textbf{const}}\ f : \tau \ldots\}}$$

SubObjPerm

$$\{\ldots \mathit{mut}\ f : \tau, \mathit{mut}'\ g : \tau' \ldots\} <: \{\ldots \mathit{mut}'\ g : \tau', \mathit{mut}\ f : \tau \ldots\}$$

SubFun

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{(\tau_1 \Rightarrow \tau_2) <: (\tau_1' \Rightarrow \tau_2')}$$

Figure 8.4: The inference rules that define the subtyping relation

to determine whether an expression `e` never produces a value (because it always diverges), then it could infer type **`Nothing`** for `e`. However, our typing rules do not distinguish between terminating and non-terminating expressions, so adding the type **`Nothing`** to the language of typing expressions seems obsolete.

Nevertheless, the types **`Any`** and **`Nothing`** are still useful. As we will see in the next section, having these types ensures that any two types have a least common supertype and a greatest common subtype[2]. This property simplifies some aspects of our type inference algorithm.

Including the type **`Any`** can also be useful from a practical perspective: 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 a cast is then checked at evaluation time. The combination of having a greatest type 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 type `Object` was the only available mechanism for implementing generic data structures[3].

The type **`Nothing`** can also be useful in combination with other programming language features. For example, suppose we extended our language with a mechanism for throwing and catching exceptions. Then an expression that throws an exception can be assigned type **`Nothing`** because its evaluation will never produce a value. (After throwing the exception, evaluation will continue with an exception handler or abort the evaluation with an error if no exception handler exists.) The rule `SubNothing` in conjunction with rule `TypeSub` will then guarantee that we can throw exceptions anywhere in a program, regardless of the constraints that the surrounding program context imposes on the type of the value produced by the throwing expression.

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 it. This is captured by the rule 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 obtain a subtype of another type.

In our structural subtyping relation, we consider additional forms of object subtyping. Specifically, consider the following program:

```
const fun = (x: {const f: {let g: Num}}) => x.f.g;
f({const f: {let g: 0, let h: 2}})
```

[2] In mathematical terms, the subtype relation forms a *lattice* if we consider object types to be equal up to reordering of fields.

[3] Generics were not included in the Java language before version 5.0.

The function `fun` expects an argument that is an object with a **`const`** field `f`, which itself stores an object with a field `g` of type **`Num`**. When we call `fun`, we provide an object whose field `f` is an object with two fields `f` and `g` of type `Num`. Thus, the call can be safely evaluated. We can capture this in our type system by making the type `{const f: {let g: Num, let h: Num}}` a subtype of `{const f: {let g: Num}}`. Note that this subtype relationship is not yet covered by the rule SubObjWidth.

More generally, we can obtain a subtype of an object type by replacing the type τ of one of its **`const`** fields by a subtype τ' of τ. This is captured by the rule SubObjDepth. We refer to this form of subtyping as *depth subtyping*. Note that we only allow depth subtyping for **`const`** fields but not for **`let`** fields. We explain the rationale for this restriction in more detail below.

If we want to use depth-subtyping for a **`let`** field, we first need to demote it to a **`const`** field. This idea is captured by the rule SubObjMut. That is, this rule captures the fact that a **`let`** field has more capabilities than a **`const`** field because it can be both read and written and not 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 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 transitivity rule SubTrans. For example, the following derivation shows that the type $\{\mathtt{const}\, g{:}\mathtt{Bool}, \mathtt{let}\, f{:}\mathtt{Bool}\}$ is a subtype of $\{\mathtt{let}\, f{:}\mathtt{Bool}\}$:

$$\textsc{SubTrans}\ \dfrac{\begin{array}{c}\textsc{SubObjPerm}\ \{\mathtt{const}\, g{:}\mathtt{Bool},\ \mathtt{let}\, f{:}\mathtt{Bool}\} <: \{\mathtt{let}\, f{:}\mathtt{Bool},\ \mathtt{const}\, g{:}\mathtt{Bool}\} \\ \textsc{SubObjWidth}\ \{\mathtt{let}\, f{:}\mathtt{Bool},\ \mathtt{const}\, g{:}\mathtt{Bool}\} <: \{\mathtt{let}\, f{:}\mathtt{Bool}\}\end{array}}{\{\mathtt{const}\, g{:}\mathtt{Bool},\ \mathtt{let}\, f{:}\mathtt{Bool}\} <: \{\mathtt{let}\, f{:}\mathtt{Bool}\}}$$

Subtyping of functions. 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 be safely replaced by another function expression of a different 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 τ_1 and τ_1', respectively, the result types τ_2 and τ_2' should be related so that this subtype relationship is safe. To train our intuition, let us consider the following example program:

```
1 const fun1 = (x: { let f: Num }) =>
2   { g: x.f + x.f };
3 const fun2 = (x: { let f: Num }) =>
4   { g: x.f + x.f, h: true };
5 const x = fun1({f: 2});
6 x.g
```

The type of `fun1`'s return value is `{let g: Num}` and the type of `fun2`'s return value is `{let g: Num, let h: Bool}`. Clearly, it is safe to replace the call to `fun1` on line 7 by a call to `fun2` because all the capabilities of the object returned by `fun1` (reading and updating field `g`) are also provided by the object returned by `fun2`. This observation generalizes: it is safe to substitute a function fun_1 by a function fun_2 if fun_2 provides stronger guarantees about its return type than fun_1. That is, if $\tau_2 <: \tau_2'$, then

$$(\tau_1 \Rightarrow \tau_2) <: (\tau_1 \Rightarrow \tau_2') \ .$$

We say that function types are *covariant* in their result types to indicate that the subtype relation keeps the same direction when we recurse into the result types of function types.

What about the parameters types of function types? It is tempting to require $\tau_1 <: \tau_1'$. However, this would be unsafe as the following program demonstrates:

```
1 const fun1 = (x: {let f: Num}) => x.f + x.f;
2 const fun2 = (x: {let f: Num, let g: Num}) => x.f + x.g;
3 fun1({f: 2})
4 fun2({f: 2, g: 2})
```

The parameter type of `fun2` is a subtype of `fun1`'s parameter type. Observe that it is not safe to replace the call to `fun1` on line 8 with a call to `fun2` with the same argument. We pass an object to `fun1` that only provides a field `f` but function `fun2` accesses fields `f` and `g` of its argument. However, we can safely replace the call to `fun2` on line 9 with a call to `fun1`. The object passed to `fun2` has fields `f` and `g` but `fun1` only accesses field `f`. 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.

8.3.2 Subtyping and Assignments

Remember that the main purpose of our 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. The subtyping rules are carefully designed to preserve type safety. In particular, if we allowed depth subtyping of **let** fields, we would introduce a loop hole in our type system. We explain this in the following.

With depth-subtyping of **let** fields we would allow the rule TypeSub to be used too liberally. The typing rules would then no longer 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 τ by an expression of a subtype τ'. 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 τ, 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.

The reason for restricting the depth-subtyping rule to **const** fields is related to the issue that **let** fields can serve as sinks of values. The following program demonstrates this issue:

```
const y = { f: {g: true} };
const fun = (x: {let f: {}}) => x.f = {};
fun(y);
y.f.g
```

The type of `y` is `{let f: {let g: Bool}}`. Evidently, the function `fun` is well-typed since its parameter `x` has type `{let f: {}}` and we are assigning `x.f` to `{}`. If we allowed depth-subtyping of **let** fields, then the call `fun(y)` would also be well-typed since the type of `y` would then be a subtype of `{let 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 get stuck when we try to evaluate the final dereference operation on field `g`. For this reason we disallow depth-subtyping of **let** fields. With the restricted depth-subtyping rule, the above program is rejected by the type checker because the call `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 **`let`** to **`const`** using the rule SubObjMut. For example, consider the following program:

```
const y = {let f: {const g: true}};
const fun = (x: {const f: {}}) => x.f;
fun(y);
y.f.g
```

This program can be safely evaluated. It is also well-typed because the following subtype relationship holds

```
{let f: {const g: Bool}} <: {const f: {}}
```

This subtype relationship enables us to type the call to `fun`. This relationship can be derived using one application of the rule SubObjMut and one application of the rule SubObjDepth. These two applications can then be combined using the rule SubTrans.

On the other hand, consider the following program, which is a combination of the two programs above:

```
const y = {let f: {const g: true}};
const fun = (x: {const f: {}}) => x.f = {};
fun(y);
y.f.g
```

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. Specifically, the body of the function `fun` is no longer well-typed because `fun` now tries to assign a new value to the **`const`** field of object `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 `S` is a subtype of a type `T`, then `S[]` (the type of an array of elements of type `S`) is a subtype of `T[]`. This treatment of arrays is unsafe because arrays can serve both as sources and sinks of values, as demonstrated by the following program:

```
String[] a = new String[1];
Object[] b = a; // a and b are now aliased
b[0] = new Object(); // a[0] now stores an Object
a[0].indexOf('c'); // indexOf not a method of Object
```

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 neither follows that `S[]` is a subtype of `T[]` nor does it follow that `T[]` is a subtype of `S[]`. The types `S[]` and `T[]` are unrelated. With this rule, the program above is rejected by the type checker because the assignment of `a` to `b` on line 2 is not well-typed. Additional run-time checks for array store operation are then no longer needed. For example, Scala's type checker follows this rule and considers array types invariant in their element types.

8.4 Type Inference with Subtyping

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:

$$\textsc{TypeSub}\quad \frac{\Gamma \vdash e : \tau' \qquad \tau' <: \tau}{\Gamma \vdash e : \tau}$$

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 τ' 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 τ of τ' in the rule TypeSub. 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:

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ \textsc{SubTrans}$$

In order to apply this rule during type inference (reading the rule bottom-up), we would have to guess the intermediate type τ_2 to make the transitive step between τ_1 and τ_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.

8.4.1 Algorithmic 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:

$$\textsc{SubTrans}\ \frac{\begin{array}{c}\textsc{SubObjPerm}\ \{\mathbf{\mathtt{const}}\, g\!:\!\mathbf{\mathtt{Bool}}, \mathbf{\mathtt{let}}\, f\!:\!\mathbf{\mathtt{Bool}}\} <: \{\mathbf{\mathtt{let}}\, f\!:\!\mathbf{\mathtt{Bool}}, \mathbf{\mathtt{const}}\, g\!:\!\mathbf{\mathtt{Bool}}\} \\ \textsc{SubObjWidth}\ \{\mathbf{\mathtt{let}}\, f\!:\!\mathbf{\mathtt{Bool}}, \mathbf{\mathtt{let}}\, g\!:\!\mathbf{\mathtt{Bool}}\} <: \{\mathbf{\mathtt{let}}\, f\!:\!\mathbf{\mathtt{Bool}}\}\end{array}}{\{\mathbf{\mathtt{const}}\, g\!:\!\mathbf{\mathtt{Bool}}, \mathbf{\mathtt{let}}\, f\!:\!\mathbf{\mathtt{Bool}}\} <: \{\mathbf{\mathtt{let}}\, f\!:\!\mathbf{\mathtt{Bool}}\}}$$

We can see that we use the transitivity rule to combine several applications of the object subtyping rules–here, the rules SubObjPerm and SubObjWidth. 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 SubObjPerm, SubObjWidth, SubObjDepth, and SubObjMut into a single rule for object subtyping. The resulting object subtyping rule, called SubObj, 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 SubRefl can be restricted to the cases where τ is one of the basic types **Num** and **Bool**.

8.4.2 Algorithmic Typing with Subtyping

Our solution for making our typing relation syntax-directed is similar: we eliminate the subtyping rule TypeSub 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.

SubRefl
$$\tau <: \tau$$

SubAny
$$\tau <: \texttt{Any}$$

SubNothing
$$\texttt{Nothing} <: \tau$$

SubFun
$$\frac{\tau_2 <: \tau_2' \qquad \tau_1' <: \tau_1}{(\tau_1 \Rightarrow \tau_2) <: (\tau_1' \Rightarrow \tau_2')}$$

SubObj
$$\frac{\begin{array}{c}\{g_1, \ldots, g_m\} \subseteq \{f_1, \ldots, f_n\} \\ \text{for all } i, j \text{, if } f_i = g_j \text{, then } \mathit{mut}_j' = \texttt{const} \text{ and } \tau_i <: \tau_j' \\ \text{or } \mathit{mut}_i = \mathit{mut}_j' \text{ and } \tau_i = \tau_j'\end{array}}{\{\mathit{mut}_1\, f_1 : \tau_1, \ldots, \mathit{mut}_n\, f_n : \tau_n\} <: \{\mathit{mut}_1'\, g_1 : \tau_1', \ldots, \mathit{mut}_m'\, g_m : \tau_m'\}}$$

Figure 8.5: Algorithmic subtyping rules for JakartaScript

TypeCall
$$\frac{\Gamma \vdash e_1 : \tau_2 \Rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2' \qquad \tau_2' <: \tau_2}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \tau}$$

TypeFunAnn
$$\frac{\Gamma' = \Gamma[x \mapsto (\texttt{const}, \tau_2)] \qquad \Gamma' \vdash e : \tau' \quad \tau' <: \tau}{\Gamma \vdash \texttt{function(}x\texttt{:}\tau_2\texttt{)}\ \texttt{:}\tau\ e\ : \tau_2 \Rightarrow \tau}$$

TypeFunRec
$$\frac{\Gamma' = \Gamma[x_1 \mapsto \tau_1][x_2 \mapsto (\texttt{const}, \tau_2)] \qquad \tau_1 = (\tau_2 \Rightarrow \tau) \qquad \Gamma' \vdash e : \tau' \qquad \tau' <: \tau}{\Gamma \vdash \texttt{function}\ x_1\texttt{(}x_2\texttt{:}\tau_2\texttt{)}\ \texttt{:}\tau\ e\ : \tau_1}$$

TypeAssignVar
$$\frac{\Gamma(x) = (\texttt{let}, \tau') \qquad \Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash x\ \texttt{=}\ e : \tau}$$

TypeAssignFld
$$\frac{\Gamma \vdash e_1 : \{\ldots \texttt{let}\ f\texttt{:}\tau' \ldots\} \qquad \Gamma \vdash e_2 : \tau \qquad \tau <: \tau'}{\Gamma \vdash e_1\texttt{.}f\ \texttt{=}\ e_2 : \tau}$$

TypeIf
$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_3 \qquad \tau_2 \sqcup \tau_3 = \tau}{\Gamma \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 : \tau}$$

TypeEqual
$$\frac{\tau_1 \sqcup \tau_2 \neq \texttt{Any} \qquad \begin{array}{c}\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \\ \tau_1, \tau_2 \text{ have no function types}\end{array} \qquad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \texttt{Bool}}$$

Figure 8.6: Algorithmic typing rules for JakartaScript 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:

```
const counterClass =
  function(_this: {let x: Num,
                   let get: () => Num,
                   let inc: () => Undefined})
  : {const get: () => Num, const inc: () => Undefined}
  {
    _this.x = 0;
    _this.get = () => _this.x;
    _this.inc = () => _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. Similarly, a client can no longer write to the fields `get` and `inc` after these fields have been properly intialized.

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. In particular, the following program is still rejected by the new typing relation:

```
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 `{let 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:

```
const o = true ? {f: 0, g: true} : {f: 2, h: 0};
2 * o.f
```

The inferred type of the "then" branch `{f:0, g: true}` is

```
{let f: Num, let g: Bool}
```

and the inferred type of the "else" branch `{f: 2, h: 0}` is

```
{let f: Num, let h: Num}
```

While the two types do not agree, they have a common supertype:

```
{let f: Num} .
```

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.f` only accesses the field `f` of `o`, which it expects to be of type `Num`. 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.f` 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 want to preserve as much information as possible that is common to the two merged 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 τ_1 and τ_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 τ_1 and τ_2 until we find the first common supertype. For example, we have

$$\begin{aligned} & \{\texttt{let f:Num}, \texttt{let g}:\texttt{Bool}\} \sqcup \{\texttt{let f:Num}, \texttt{let h}:\texttt{Num}\} \\ = & \{\texttt{let f:Num}\} \end{aligned}$$

Hence, the type that the new TypeIf rule infers for the expression

```
true ? {f: 0, g: true} : {f: 2, h: 0}
```

is $\{\mathtt{let}\ \mathtt{f}{:}\mathtt{Num}\}$. You may ask yourself why the inferred type is not

$$\{\mathtt{let}\ \mathtt{f}{:}\mathtt{Num}, \mathtt{let}\ \mathtt{g}{:}\mathtt{Bool}\}$$

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 τ_1 and τ_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 τ_1 and τ_2 of the two expressions must have some common supertype that is different from **Any**. We express this condition by stating that $\tau_1 \sqcup \tau_2 \neq \mathtt{Any}$. 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 τ_1 and τ_2 is a type τ that satisfies the following two properties: (1) τ is a supertype of both τ_1 and τ_2 and (2) any other supertype τ' of τ_1 and τ_2 must also be a supertype of τ.

In our type system, joins are not unique. That is, given τ_1 and τ_2, we can have two distinct types τ and τ' such that both satisfy the conditions (1) and (2) above. For example, consider the types:

$$\tau_1 = \{f{:}\mathtt{Num}, g{:}\mathtt{Bool}; h{:}\mathtt{Bool}\}$$
$$\tau_2 = \{h{:}\mathtt{Bool}, g'{:}\mathtt{Num}, f{:}\mathtt{Num}\}$$

$$\textsc{JoinNothing}_1 \quad \texttt{Nothing} \sqcup \tau = \tau \qquad\qquad \textsc{JoinNothing}_2 \quad \tau \sqcup \texttt{Nothing} = \tau \qquad\qquad \textsc{JoinBasic}_= \quad \frac{\tau \in \{\texttt{Bool}, \texttt{Num}, \texttt{Any}\}}{\tau \sqcup \tau = \tau}$$

$$\textsc{JoinObjEmp} \quad \{\} \sqcup \{mut_g\ g \mathbin{:} \tau_g\} = \{\} \qquad\qquad \textsc{JoinObjNo} \quad \frac{f \notin \{\overline{g'}\} \qquad \{\overline{mut_g\ g \mathbin{:} \tau_g}\} \sqcup \{\overline{mut_{g'}\ g' \mathbin{:} \tau_{g'}}\} = \tau}{\{mut_f\ f \mathbin{:} \tau_f, \overline{mut_g\ g \mathbin{:} \tau_g}\} \sqcup \{\overline{mut_{g'}\ g' \mathbin{:} \tau_{g'}}\} = \tau}$$

$$\textsc{JoinObjMut}_{\neq} \quad \frac{\begin{array}{c}(\tau_f \neq \tau'_f \text{ or } mut_f \neq mut'_f) \qquad \{\overline{mut_g\ g \mathbin{:} \tau_g}\} \sqcup \{\overline{mut_{g'}\ g' \mathbin{:} \tau_{g'}}\} = \{\overline{mut_h\ h \mathbin{:} \tau_h}\} \\ \tau_f \sqcup \tau'_f = \tau''_f \qquad \tau = \{\texttt{const}\ f \mathbin{:} \tau''_f, \overline{mut_h\ h \mathbin{:} \tau_h}\}\end{array}}{\{mut_f\ f \mathbin{:} \tau_f, \overline{mut_g\ g \mathbin{:} \tau_g}\} \sqcup \{mut'_f\ f \mathbin{:} \tau'_f, \overline{mut_{g'}\ g' \mathbin{:} \tau_{g'}}\} = \tau}$$

$$\textsc{JoinObjMut}_= \quad \frac{\{\overline{mut_g\ g \mathbin{:} \tau_g}\} \sqcup \{\overline{\overline{mut_{g'}\ g' \mathbin{:} \tau_{g'}}}\} = \{\overline{mut_h\ h \mathbin{:} \tau_h}\} \qquad \tau = \{mut_f\ f \mathbin{:} \tau_f, \overline{mut_h\ h \mathbin{:} \tau_h}\}}{\{mut_f\ \mathbin{:} \tau_f, \overline{mut_g\ f \mathbin{:} \tau_g}\} \sqcup \{mut_f\ f \mathbin{:} \tau_f, \overline{mut_{g'}\ f' \mathbin{:} \tau_{g'}}\} = \tau}$$

$$\textsc{JoinFun} \quad \frac{\tau_1 \sqcap \tau_2 = \tau \qquad \tau'_1 \sqcup \tau'_2 = \tau'}{(\tau_1 \Rightarrow \tau'_1) \sqcup (\tau_2 \Rightarrow \tau'_2) = (\tau \Rightarrow \tau')}$$

Figure 8.7: Rules for computing joins.

Then both of the following types are joins of τ_1 and τ_2:

$$\tau = \{f \mathbin{:} \texttt{Num}, h \mathbin{:} \texttt{Bool}\}$$
$$\tau' = \{h \mathbin{:} \texttt{Bool}, f \mathbin{:} \texttt{Num}\}$$

Note that we have $\tau <: \tau'$ and $\tau' <: \tau$. In fact, τ and τ' are equal up to permutation of their fields. This is true in general: all the joins of two types τ_1 and τ_2 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 $\tau_1 \sqcup \tau_2$ for *the* join of τ_1 and τ_2.[4]

The inference rules in Figure 8.7 describe how the join function $\sqcup$ can be computed using recursion on the structure of type expressions. Note that the rules only provide a partial definition of $\tau_1 \sqcup \tau_2$. In all the missing cases where no rule applies, such as taking the join of a basic type and an object type, we define $\tau_1 \sqcup \tau_2 = \texttt{Any}$.

We next discuss the individual rules in detail.

Joins with `Nothing`. Since `Nothing` is the smallest type, we have $\texttt{Nothing} \sqcup \tau = \tau \sqcup \texttt{Nothing} = \tau$ for all types τ. These cases are captured by the rules

[4] More formally, we consider the quotient of *Typ* under the equivalence relation that is induced by the subtype relation. Then $\tau_1 \sqcup \tau_2$ is the least upper bound of the equivalence classes represented by τ_1 and τ_2 in the partial order that is induced by $<:$ on this quotient. The least upper bounds of a partial order are always unique if they exist.

JoinNothing$_1$ and JoinNothing$_2$.

Joins of Basic Types. If one of the two types τ_1 and τ_2 is a basic type and the other type is different from **`Nothing`**, then computing their join $\tau_1 \sqcup \tau_2$ 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 is captured by the rule JoinBasic$_=$.

Joins of Object Types. Next, we discuss the case where τ_1 and τ_2 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 τ by taking all fields f that are common to both τ_1 and τ_2, and *merging* the types of f appropriately. The join computation thus works by recursion over the fields of τ_1.

The base case is captured by the rule JoinObjEmp. The remaining rules capture the cases where τ_1 has at least one field f. The recursion then goes over the left-most field f of τ_1 using further case analysis. Let us first consider the case where f only occurs in τ_1 but not τ_2. This case is captured by the rule JoinObjNo. In this case, f will not be included in the resulting join τ and we simply obtain τ by recursively computing the join for the remaining fields of τ_1 with τ_2. If f occurs in both τ_1 and τ_2, we have to make another case distinction. In the first subcase, we consider the situation where f is a **`let`** field in both τ_1 and τ_2, and the types of f are identical. Then we can simply add f as a **`let`** field of that type to the resulting join τ. This case is captured by the rule JoinObjLet$_=$. In the remaining cases, f must either have a different type or a different mutability in τ_1 and τ_2. These two cases are captured by the rule JoinObjMut$_{\neq}$. To obtain any common object supertype of τ_1 and τ_2 that still contains field f, we must use depth-subtyping for the field f, change the mutability of f from **`let`** to **`const`**, or both. Either way, the join τ of τ_1 and τ_2 will have a **`const`** field f whose type is the join of the types of f in τ_1 and τ_2.

Joins of Function Types. Finally, we consider the case of computing the join of two function types $\tau_1 \Rightarrow \tau_1'$ and $\tau_2 \Rightarrow \tau_2'$. The join of two function types is again a function type of the form $\tau \Rightarrow \tau'$ as captured by the rule JoinFun.

How do we compute the types τ and τ' of the resulting function type? Since function types are covariant in their result types, it is easy to see that we must have $\tau' = \tau_1' \sqcup \tau_2'$. The following program exemplifies this intuition:

```
const fun1 = () => {f: 0, g: true};
const fun2 = () => {f: 1, h: 1};
const fun = true ? fun1 : fun2;
fun().f
```

The inferred type for `fun` must be such that all subsequent usages of `fun` are safe, regardless of whether `fun` is evaluated to `fun1` or `fun2`. The expression `fun().f` accesses the field `f` of `fun`'s return value. This is safe since the objects returned by `fun1` and `fun2` both have a field `f` of type `Num`. However, if we replaced `fun().f` by `fun().h`, then the program would only be safe if the conditional expression defining `fun` always took the "else" branch. The inferred return type for `fun` should capture the maximal information that is common to the return types of `fun1` and `fun2`, 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:

```
const fun1 = (o: {let f: Num, let g: Num}) => o.f + o.g;
const fun2 = (o: {let f: Num, let h: Num}) => o.f + o.h;
const fun = true ? fun1 : fun2;
fun({f: 1, g: 2, h: 3})
```

The inferred type of `fun1` is

$$\{\mathbf{let}\ \mathtt{f}{:}\mathbf{Num}, \mathbf{let}\ \mathtt{g}{:}\mathbf{Num}\} \Rightarrow \mathbf{Num}$$

and the inferred type of `fun2` is

$$\{\mathbf{let}\ \mathtt{f}{:}\mathbf{Num}, \mathbf{let}\ \mathtt{h}{:}\mathbf{Num}\} \Rightarrow \mathbf{Num}$$

The call to `fun` must be safe regardless of whether `fun` is bound to `fun1` or `fun2`. Note that `fun1` accesses fields `f` and `g` of the passed object whereas `fun2` accesses fields `f` and `h`. Hence, the object that is passed to `fun` must at least have the fields `f`, `g`, and `h`. Computing the join of the parameter types of `fun1` and `fun2` yields

$$\{\mathbf{let}\ \mathtt{f}{:}\mathbf{Num}\}$$

Thus, if we defined the parameter type of the join of the types of `fun1` and `fun2` to be $\{$**let** `f`:**Num**$\}$, then the call `fun({f: 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 `{let f: Num, let g: Num }` and `{let f: Num, let h: Num }` is the type

```
{let f: Num, let g: Num, let h: Num}
```

Thus the inferred type for `fun` is:

```
{let f: Num, let g: Num, let h: Num} => Num
```

We refer to the greatest common subtype of two types τ_1 and τ_2 as their *meet*, which we denote by $\tau_1 \sqcap \tau_2$.

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[5]. Thus, a meet τ of two types τ_1 and τ_2 is a type that satisfies the following two conditions: (1) τ is a common subtype of τ_1 and τ_2; and (2) any other common subtype τ' of τ_1 and τ_2 is a also a subtype of τ. Similar to joins, meets are unique up to reordering of fields in object type expressions. So we denote by $\tau_1 \sqcap \tau_2$ 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, MeetFun, is the dual of the rule JoinFun. That is, we obtain MeetFun from JoinFun by replacing meets by joins and vice versa.

Again, similar to the rules for computing joins, the rules only partially define the meet of two types. For all the missing cases, we define $\tau_1 \sqcap \tau_2 =$ **`Nothing`**.

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 $\tau = \tau_1 \sqcap \tau_2$ of two object types τ_1 and τ_2, we proceed as follows. Again, the computation of the meet recurses over the fields of τ_1. The individual cases are captured by the MeetObj rules. First, if a field f of τ_1 does not occur in τ_2', then it is simply added to the resulting meet τ, and vice versa for the fields that occur in τ_2 but not in τ_1. These cases are covered by the rules MeetObjNO and MeetObjEmp.

The remaining rules consider the cases where a field f of τ_1 also occurs in τ_2. In this case, we first check whether f has mutability **`let`** in both types. If yes, then the types τ_f and τ_f' of f in τ_1 and τ_2 must be equal to obtain any common subtype (rule MeetObjLet). If they are not equal, then $\tau_1 \sqcap \tau_2 =$ **`Nothing`** (this case is omitted).

If the mutability of f is **`const`** in both τ_1 and τ_2, then we can still use depth-subtyping to obtain a common subtype of τ_f and τ_f' (rule MeetObjConst). In this case, the mutability of f in τ remains **`const`** and its type is computed recursively by taking the meet of the types τ_f and τ_f'.

The remaining two rules MeetObjConstLet and MeetObjLetConst handle the cases where the mutabilities of f differ in τ_1 and τ_2. In these cases, we can still use depth-subtyping to obtain a common subtype of τ_1 and τ_2'. However, since depth-subtyping is restricted to **`const`** fields, the meet is nontrivial only if the type of the **`let`** version of f is a subtype of the type of the **`const`** version of

[5] That is, meets are the greatest lower bounds of the partial order that is induced by the subtype relation.

$$\textsc{MeetAny}_1 \quad \mathtt{Any} \sqcap \tau = \tau$$

$$\textsc{MeetAny}_2 \quad \tau \sqcap \mathtt{Any} = \tau$$

$$\textsc{MeetBasic}_= \quad \frac{\tau \in \{\mathtt{Bool}, \mathtt{Num}, \mathtt{Any}\}}{\tau \sqcap \tau = \tau}$$

$$\textsc{MeetObjEmp} \quad \{\} \sqcap \{\overline{mut_f\ f\!:\!\tau_f}\} = \{\overline{mut_f\ f\!:\!\tau_f}\}$$

$$\textsc{MeetObjConst} \quad \frac{\begin{array}{c}\{\overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \{\overline{mut_h\ h\!:\!\tau_h}\} \\ \tau_f \sqcap \tau'_f = \tau''_f \qquad \tau = \{\mathtt{const}\ f\!:\!\tau''_f, \overline{mut_h\ h\!:\!\tau_h}\}\end{array}}{\{\mathtt{const}\ f\!:\!\tau_f, \overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\mathtt{const}\ f\!:\!\tau'_f, \overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \tau}$$

$$\textsc{MeetObjLet} \quad \frac{\{\overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \{\overline{mut_h\ h\!:\!\tau_h}\} \qquad \tau = \{\mathtt{let}\ h\!:\!\tau_h, \overline{mut_h\ h\!:\!\tau_h}\}}{\{\mathtt{let}\ f\!:\!\tau_f, \overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\mathtt{let}\ f\!:\!\tau_f, \overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \tau}$$

$$\textsc{MeetObjLetConst} \quad \frac{\begin{array}{c}\{\overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \{\overline{mut_h\ h\!:\!\tau_h}\} \\ \tau_f <: \tau'_f \qquad \tau = \{\mathtt{let}\ f\!:\!\tau_f, \overline{mut_h\ h\!:\!\tau_h}\}\end{array}}{\{\mathtt{let}\ f\!:\!\tau_f, \overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\mathtt{const}\ f\!:\!\tau'_f, \overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \tau}$$

$$\textsc{MeetObjConstLet} \quad \frac{\begin{array}{c}\{\overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \{\overline{mut_h\ h\!:\!\tau_h}\} \\ \tau'_f <: \tau_f \qquad \tau = \{\mathtt{let}\ f\!:\!\tau'_f, \overline{mut_h\ h\!:\!\tau_h}\}\end{array}}{\{\mathtt{const}\ f\!:\!\tau_f, \overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\mathtt{let}\ f\!:\!\tau'_f, \overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \tau}$$

$$\textsc{MeetObjNo} \quad \frac{\begin{array}{c}\{\overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \{\overline{mut_h\ h\!:\!\tau_h}\} \\ f \notin \{\overline{g'}\} \qquad \tau'' = \{mut_f\ f\!:\!\tau_f, \overline{mut_h\ h\!:\!\tau_h}\}\end{array}}{\{mut_f\ f\!:\!\tau_f, \overline{mut_g\ g\!:\!\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'\!:\!\tau_{g'}}\} = \tau''}$$

$$\textsc{MeetFun} \quad \frac{\tau_1 \sqcup \tau_2 = \tau \qquad \tau'_1 \sqcap \tau'_2 = \tau'}{(\tau_1 \Rightarrow \tau'_1) \sqcap (\tau_2 \Rightarrow \tau'_2) = (\tau \Rightarrow \tau')}$$

Figure 8.8: Rules for computing meets.

f. The resulting mutability of f in τ is then **`let`**. The type of f in τ is the type of the **`let`** version in τ_1, respectively, τ_2 (which is also equal to the meet of τ_f and τ_f' in theses cases).