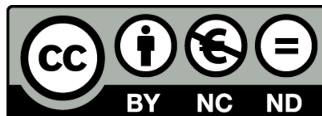
Principles of Programming Languages
– lecture notes for CS2105O2 –

Andrei Arusoaie

License

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

License Details: http://creativecommons.org/licenses/by-nc-nd/4.0/



The material included in these lecture notes is heavily based on several important books in the domain of programming languages: [7, 3, 2, 8]. The main purpose of the material is to be the primary learning resource for the students of enrolled for CS2105O2.

Contents

1 Introduction **4**
1.1 Short History 5

2 Basics **12**
2.1 Algebraic Data Types and Functions 12
2.1.1 Enumerated types 12
2.1.2 Inductive definitions 14
2.1.3 Recursive functions 16
2.1.4 **bool** and **nat** in Coq 18
2.1.5 Proofs 18
2.2 Induction 27
2.2.1 Note to the reader 32
2.3 Polymorphism 32
2.3.1 Polymorphic lists 33
2.4 Higher-Order Functions 36
2.4.1 Functions as parameters to other functions 36
2.4.2 Anonymous functions 37
2.4.3 Functions that return other functions 37
2.5 Logic in Coq and related tactics. 39
2.5.1 Implication 40
2.5.2 Conjunction 41
2.5.3 Disjunction 43
2.5.4 Negation 45
2.5.5 Existential quantification 47
2.5.6 Exercises 48

3 Syntax **50**
3.1 Derivations 51
3.2 Ambiguities 52
3.3 Abstract vs. Concrete Syntax 55
3.4 BNF 58

3.5 Abstract Syntax in Coq 58
3.5.1 Custom Grammar 60
3.6 A simple imperative language: **IMP** 60
3.7 Exercises 62

4 Semantics **63**
4.1 An evaluator for **IMP** 65
4.1.1 Environment 65
4.1.2 Evaluator for arithmetic expressions 67
4.1.3 Evaluator for boolean expressions 69
4.1.4 Evaluator for **IMP** statements 70
4.1.5 Exercises 74
4.2 Structural Operational Semantics 74

5 Big-step SOS **75**
5.1 Configurations 75
5.2 Sequents 76
5.3 Rule schemata 76
5.4 Big-step SOS for **IMP**: arithmetic expressions 77
5.5 Big-step SOS rules for arithmetic expressions in Coq 78
5.6 Big-step SOS for **IMP**: boolean expressions 80
5.7 Big-step SOS for boolean expressions in Coq 81
5.8 Big-step SOS for **IMP**: statements 82
5.9 Big-step SOS rules for statements in Coq 84
5.10 Proof search 86
5.11 Exercises 87
5.12 Improving **IMP** 88

6 Small-step SOS **91**
6.1 Small-step SOS for **IMP**: arithmetic expressions 92
6.2 Small-step SOS for arithmetic expressions in Coq 93
6.3 Small-step SOS for **IMP**: boolean expressions 95
6.4 Small-step SOS for boolean expressions in Coq 97
6.5 Small-step SOS for **IMP**: statements 99
6.6 Small-step SOS for statements in Coq 100
6.7 Big-step vs. Small-step SOS 101

7 Type systems **103**
7.1 Introduction 103
7.2 **IMP** Syntax of expressions 105
7.3 Small-step SOS for expressions 106

7.4 A type system for expressions 109
7.5 A type system for expressions in Coq 111
7.6 Properties of the typing relation 113
7.6.1 Progress 113
7.6.2 Type Preservation 115
7.6.3 Type Soundness 115

8 Compilation 117
8.1 Simple expressions 118
8.2 A stack machine 119
8.3 Certified compilation 122
8.4 Exercises 124

9 Untyped lambda calculus 125
9.1 Functional programming 125
9.2 Untyped lambda (λ) calculus 126
9.2.1 λ-terms 126
9.2.2 Free and bound variables. 127
9.2.3 Capturing substitution 128
9.2.4 Capture-avoiding substitution 129
9.2.5 Alpha equivalence 131
9.2.6 Beta-reduction 131
9.2.7 Normal forms 134
9.2.8 Evaluation strategies 134

Chapter 1

Introduction

Learning a new programming language is nowadays a must. Programmers need to adapt to a wide range of programming languages, tools and technologies really fast. Simply learning a programming language and hoping that this is *the way to go* is overrated.

This material is intended to highlight the fundamental and most common concepts that we encounter and use in the modern programming languages. We discuss various aspects, starting with the syntax of programming languages, traditional operational semantics styles, type systems, compilation, different programming paradigms (e.g., object oriented programming, functional programming and lambda calculus, logic programming) and others.

The goal is to give the reader an overview of what is a programming language and how various techniques and tools can impact a programming language. This material provides the reader with a different perspective about programming in general.

Organisation

Using proof assistants for teaching programming languages is very common [4, 5]. Our choice for this course is Coq. The first chapter contains a short history of programming languages that is heavily inspired from Section 13 in [2]. The second chapter presents the preliminary concepts and tools that we will use through this material. In the third chapter we recall aspects related to the syntax of programming languages. The fourth chapter includes various ways to define semantics of programming languages: first, we show how to define interpreters, and then we present the traditional operational semantics styles, that is, big-step and small-step SOS. The fifth chapter presents types and type checking. The sixth chapter addresses compilers and certified compilation. The seventh chapter includes a presentation

of the untyped version of the lambda calculus.

1.1 Short History

A modern computer is a digital machine, that is programmable and provides storage for programs and data. Among the first machines that fits into this definition is the EDSAC computer, which was designed and developed at the University of Cambridge. However, EDSAC was not the first successful attempt to build a computer.

ASCC/MARK I (Automatic Sequence Controlled Calculator) was a machine built in 1944 by IBM in collaboration with the Harward University, where H. Aiken was the principal investigator. The machine was used by the U.S. Navy for military tasks. ASCC/MARK I used punched tapes and other external physical devices (like switches) to obtain its instructions and to transmit data to its computing units. Programming was rudimentary: for instance, a decimal constant of 23 digits length needed a manual setup of 23 switches, each switch having 10 positions - one for each digit. The calculator was powered by a 5 horsepower electrical engine through a drive shaft. This engine was the main power source of the calculator and the system clock. The calculating units were synchronised using rotating shafts, clutches and other electromechanical components.

ENIAC (Electronic Numerical Integrator and Computer) was built at the Moore School at University of Pennsylvania by J. Mauchly and J.P. Eckert in 1946. For a short period of time, J. von Neumann was involved in the project too. ENIAC did not have program storage and was programmed using electrical cables that were connecting different part of the computer according to the input parameters. ENIAC was faster than ASCC/MARK I and this was a milestone for computation. Basically, it was the first computer that was able to tackle mathematical problems that required significant amounts of computational effort. ENIAC was used to compute complicated ballistic trajectories. The major improvement consisted in using individual, modular panels that were able to perform different functions. Instead of using electromechanical power, these modules were build using vacuum tubes, crystal diodes, relays, resistors and capacitors. This enabled ENIAC to hold ten-digits numbers in memory and pass them easily among the various modules in order to perform various computations. Moreover, the computer was able to branch, that is, it was able to trigger different computations depending on the sign of a computed results. This raised the interest of the researchers which started to actually design programs for this machine. One drawback of the ENIAC machine was the high electricity power consumption (150kW).

Later, the group led by Maurice Wilkes used ideas from J. Mauchly and J.P. Eckert (which in turn, used some ideas from J. von Neumann) to build the EDSAC computer. EDSAC was first used in 1949, although its construction started in 1947. It first calculated a table of square numbers and a list of prime numbers. This machine was more efficient than ENIAC (only 11kW of electricity needed); the cycle time for ordinary operations was 1.5 ms and 6 ms for multiplication. Initially, the input was via five-hole punched tape, and output was via a teleprinter. Although the computer was complex and could perform very complex tasks, the users complained about how difficult was to produce correct punched cards. They needed a novel way of introducing programs into a computer. This led to the invention of the first assembler: programs were written using the mnemonics and the assembler was used to translate that into machine code.

So far, we have not mentioned any programming language in the sense that we understand programming languages today. At that time, coding was strongly connected to the machines. Programs were basically low-level specifications using binary code, while the operations and calculations were machine specific. These specifications form the so-called *machine language*, which consists of elementary instructions (load a value into a register, add, multiply, etc.) that could be immediately executed by the computation unit. The entire programming process was manual and once the programs got bigger, programming became extremely hard. Machine languages are part of the *first-generation* languages (1GL).

The *second-generation* languages (2GL) are *assembly* languages, which were more human-oriented than machine languages. It sounds weird, but assembly programs were easier to handle by humans rather than programs written in machine language. These assembly languages are symbolic representations of the machine languages: there is a one-to-one correspondence between the machine instructions and assembly language codes. The *assembler* translates assembly programs into machine instructions. Unfortunately, each machine has its own assembly language, so portability is an issue.

In 1950s, abstract languages that ignored the physical specifics of a computer represented a huge step towards improving programming languages. The central idea was that programs can be written in *high-level languages* and then automatically be translated into executable instructions for machines. FORTRAN (FORmula TRANsformation) is among the first language that follows this philosophy. The language was proposed in 1957, but the same idea of translating high-level programs to low-level ones is still present today. There is a long list of programming languages that have emerged ever since. These languages form the *third-generation* languages (3GL).

In the beginning, programming languages were designed for maximum efficiency with respect to the existing hardware. The main reason was that hardware was extremely expensive. Nowadays, the paradigm has shifted: hardware is cheap, so programming languages are designed to increase the efficiency of the programming process rather than optimising programs for the existing hardware. Today, our focus is on different challenges, like development speed, security concerns, the correctness of the software.

The following list mentions several notable programming languages that have been developed over time.

- **1950s and 1960s**. This is was the era of mainframes that used *batch* processing: the computer takes a "batch" of data as input and produces another batch of data as output. Both data and programs are stored on a punched card, and they were read using special equipment. Once the punched card was provided, the interaction with the user was inexistent. The machine had to be capable of recovering itself from a program error during a run.

- **FORTRAN** (1957) is considered the first high-level imperative language. Compared to its predecessors, FORTRAN contained several language constructs that were independent from the characteristics of a particular machine. Also, it was the first language to allow the direct use of symbolic arithmetic expressions. Simple expressions like 'a + b' could not be used prior FORTRAN. The language was designed for applications of numerical-scientific type. Further versions of FORTRAN allowed the use of local and global environments, dynamic memory, more structured command (e.g., if-then-else), and other useful features that are still present today.

- **ALGOL** is a family of imperative languages that were introduced at the end of 1950s. These languages were predominant in the academic world and had an impact on the following modern languages. ALGOL is an acronym for ALGOrithmic Languages and was well-suited for expressing algorithms in general.

- **LISP** (LISt Processor) was designed in 1960 at MIT. This language was very different from the others at that time: it was designed to process lists instead of being designed for numeric computations and algorithms. In Lisp, the symbolic expressions (a.k.a. s-expressions) are manipulated with ease. The language itself was used a lot in Artificial Intelligence (for instance, in the automatic translation of texts). Lisp

was the precursor of Scheme - currently used in many functional programming courses. Both Lisp and Scheme are functional languages, where each program is a sequence of expressions that are evaluated. Higher-order programming, where functions are first-class citizens, is a concept very common in functional languages. Currently, many mainstream languages (e.g., C++, Java, C#) borrow this feature.

- **Cobol - COmmon Business Oriented Language.** This language was designed for business, especially for commercial applications, where the focus is to have a syntax that makes programmers more productive. The language is still revised and in use today.

- **Simula** - a descendent of Algol60. Developed in Norway, the language was considered to be too advanced for its time. It was the first language that mentions the concepts of classes, objects, subtypes, and dynamic method dispatch. This language was basically the precursor of Smalltalk and C++. Later versions of Simula come with other modern features: call-by-reference, pointers, coroutines, etc.

- **1970s**. This period was marked by the emergence of the microprocessors. Computers became smaller in size, but comparable in terms of computation power with the mainframes. Batch processing was now more interactive through terminals, where programmers could interact directly with the execution of the program. Programming languages became more and more programmer friendly. Besides improving the existing imperative programming paradigm, new paradigms emerged: object-oriented programming and declarative programming. The latter, can be actually divided into functional programming and logical programming.

- **C.** Probably the most important language for 1970s is the C programming language, designed by Dennis Ritchie and Ken Thompson at AT&T Bell Labs. C was designed as an operating systems language, being used by the UNIX operating system, but it became a general purpose language very quickly. C was better than the languages in the ALGOL family because it provided access to some low-level functionalities of the machine: for instance, it is possible to acces the characters emitted by the terminal, you can use command to process data in real time, it has a compact syntax, its programs are compiled into efficient machine code.

- **Pascal.** This language was developed by Niklaus Wirth around 1970s and was used mostly for educational purposes. Pascal introduced the

concept of intermediate code as an instrument for portability. Pascal programs were compiled into P-code, which is an intermediate language that was interpreted on the host machine. Another strong point in the favour of Pascal was the possibility to define functions and blocks of code which significantly increased the structuring of the code. The language also came with dynamic memory management using both a stack (for activation records) and a heap (for the explicitly allocated memory). Pascal has an extensive type system and a static type checker (err... there some checks performed at runtime). Moreover, it allows the users to define new types.

- **Smalltalk.** The principles of encapsulation and information hiding are not builtin in languages like C and Pascal. Smalltalk comes with an innovation with this respect: it provides mechanisms for encapsulation and information hiding using *classes* and *objects* (remember that these were introduced by Simula) and visibility modifiers for classes and members. Smalltalk was designed to include the concept of object as a primitive in the language. Moreover, it was designed to come with a programming environment that facilitates writing and executing programs.

- **Declarative languages.** Declarative programming is a different paradigm, where programming means to declare whats needs to be done rather than telling the computer how to do computations. A declarative language interpreter figures out by itself how to do things. In general, leaving an interpreter to decide how to do computations is not very efficient in terms of execution time and memory.

- **ML.** The Meta Language is a semi-automatic system for proving properties of programs and was developed by Robin Milner's group at Edinburgh. In ML, programs are just sets of function definitions. The language was featured with a safe type system which extends the type system of Pascal. The type safety in ML is a very useful machinery that can be used to exclude runtime errors that derive from type violations. Programs that can be type checked are guaranteed to evaluate correctly and return a result that has the type of the program. Besides type checking, ML supports type inference as well.

- **Prolog.** The Prolog language is the first logic-programming language that is still in use today. It is based on the advances on automatic deduction in first-order logic, more precisely, on unification algorithms and resolution. In particular, the SLD resolution (proposed by R.

Kowalski in 1974) allows to prove a formula by computing the values of the involved variables that make the formula true. This technique was implemented by Pierre Roussel and Alain Comerauer in their first prototype of Prolog.

- **1980s.** The 1980s era was dominated by the development of the PC. The first commercial PC is considered to be the Apple II (1978). In 1984, Apple released Macintosh - a computer with an operating system that included a graphical interface based on windows, icons and a mouse. In 1990s, Microsoft introduced their own windows system.

  PCs changed the world of programming. Operating systems became more and more complex which caused a revolution in terms of reusing code and organising such complex systems. This area was ideal for using object-oriented languages due to modularisation and abstractisation.

- **C++.** In 1986, Bjarne Stroustrup defined the first version of the C++ language at Bell Labs AT&T. It took him several years of work on finding how to add classes to the C language without losing the efficiency and the compatibility with the existing C infrastructure. C remained a subset of the C++ language, and the C++ compiler is able to compile C programs as well. C++ does not use a garbage collector and it remains compatible with C from this point of view.

  C++ was a long time the main programming language. It has a static type system, it provides support for templates (a kind of generic class) which supports polymorphism. In C++, objects can be allocated in activation records on the stack and they can be manipulated directly rather than via pointers. The lookup mechanism for methods is simpler and more efficient than in Smalltalk. Also, multiple inheritance is yet another feature of the language, but it is considered by many a bit problematic.

- Other languages that are worth to be mentioned are **Ada** (an OOP high-level language extended from Pascal) and **Constrain Logic Programming** languages (CLPs like Prolog II and III).

- **1990s.** This was a booming period due to the development of the Internet and World Wide Web. Many aspects of programming languages have been profoundly changed by the possibility of connecting computers all around the world. A lot of challenges needed to be faced:

efficiency, reliability, correctness and security. Low-level communication protocols have been developed, markup languages (HTML, XML), more advanced and portable languages have been proposed.

- **Java.** At Sun, Jim Gosling and his group (the *Green team*) developed an object-oriented language that was intended to be used on computing devices with limited power. These devices were meant to be used as internet navigation devices.

  Several key features of Java made it really popular. The first is *portability*. Java runs on a virtual machine which is implemented on top of various computer architectures. This virtual machine runs *Java bytecode*. The Java compilers translated the Java code into bytecode that can be executed by any virtual machine. So, one can compile Java code and run it on any virtual machine. The second important feature of Java is *security*. The existing virtual machine mechanisms behind Java allows one to compile and run code anywhere. Since Java was meant to execute code received on the network, security becomes an issue. But various techniques have been used to tackle it. They used type systems, and thus they could guarantee that runtime type errors cannot occur. Type safety was implemented at three levels: at the compiler level, at the bytecode level, and at the runtime level. Another important feature of Java is avoiding the explicit handling of the pointers by using a *garbage collector*. This made Java more reliable and simple to use when compared to C++. Other features of Java are: *dynamic method dispatch*, *dynamic loading of classes*, *exceptions* (which are arguably a good thing), *concurrency and threads*, web applications support, etc.

At the moment of writing these lecture notes, there is a huge list of programming languages that have been developed over the years: languages for mobile devices (Kotlin, Dart, ObjectiveC, Swift), general purpose (C#, Go, Rust), scripting for the web (Php, Javascript, Typescript), writing mechanised proofs (Coq, Isabelle/HOL, Lean), functional programming languages (Haskell, Ocaml, Scheme), webservice composition (e.g., BPEL), rewriting (Maude, CafeOBJ). Programmers need to always adapt to new languages and frameworks. This is why a deeper understanding of programming languages is needed and this is what these notes attempt to accomplish.

Chapter 2

Basics

In these notes we use a proof assistant as a framework for defining programming language semantics. In this framework we can execute programs written in the defined languages and we can prove properties about languages and programs. The proof assistant that we use is called Coq [1].

Coq allows us to use a functional programming style to define *algebraic data types* and *higher-order functions*, together with other features like *pattern matching* and *polymorphism*. In the following, we present these features in detail, and we also implement in Coq several concepts related to programming languages.

2.1 Algebraic Data Types and Functions

Algebraic data types are data types that are formed using other types. The term *algebraic data type* is typically linked to functional programming and type theory. We are going to explain how to define algebraic data types in Coq in the next sections.

2.1.1 Enumerated types

The simplest way to define an algebraic data type is to enumerate its values. Here is a simple enumeration of a type called **Season**:

```
Inductive Season :=
| Winter
| Spring
| Summer
| Fall.
```

The keyword `Inductive` introduces the name of our new type **Season**. The *values* of this type are Winter, Spring, Summer, and Fall. That's it! We have now defined our own type. These values are also called *constructors*. The notion of *constructor* will make more sense later on, when we will actually construct new values from other types. In Coq, we can check that these values are indeed of type **Season**:

```
Check Winter.
```

```
Winter
     : Season
```

Exercise 2.1.1 *Check the type of the other values.*

We can now write functions that operate on the values of type **Season**. For instance, we can write a function that, for a given season, returns the next season:

```
Definition next_season (s : Season) : Season :=
  match s with
  | Winter ⇒ Spring
  | Spring ⇒ Summer
  | Summer ⇒ Fall
  | Fall ⇒ Winter
  end.
```

Let us explain what we just did. The `Definition` keyword introduces a new function called next_season. This function has an argument s of type **Season** and it returns a value of type **Season**. Let us check that in Coq as well:

```
Check next_season.
```

```
next_season
     : Season → Season
```

In our definition of next_season, the types (that is, the type of the argument s and the return type) are explicitly declared. It turns out that Coq can actually figure out these types automatically. This feature of Coq is called *type inference*. In general, we prefer to include the types explicitly because it improves the reading of the code.

The implementation of next_season uses *pattern matching*. Our implementation of next_season performs case analysis on s and it returns specific values. The `match` construct actually looks at the possible values of s (that is, all constructors of type **Season**) and returns the value from the corresponding branch. Let us test our function:

```
Compute (next_season Winter).
    = Spring
    : Season
```

Exercise 2.1.2 *Execute* next_season *for the other possible inputs.*

2.1.2 Inductive definitions

Inductive definitions are a very important tool in the study of programming languages. An inductive definition is basically a set of inference rules of the form

$$\frac{I_1 \quad \ldots \quad I_n}{C}\ name,$$

where $I_1, \ldots, I_n$ are *premises*, C is a *conclusion* and *name* is the rule name. Optionally, we can attach a *condition* to these rules. An inference rule states that the premises are sufficient for the conclusion, that is, in order to show that C holds, we have to show that $I_1, \ldots, I_n$ hold. If $n = 0$ (i.e., there are no premises) then the rule is called *axiom*.

The next two inference rules form an inductive definition for natural numbers:

$$\frac{\cdot}{O \in \mathbb{N}}\ zero \qquad\qquad \frac{n \in \mathbb{N}}{S\ n \in \mathbb{N}}\ succ$$

These rules specify that the set $\mathbb{N}$ includes O (using the axiom *zero*) and $S\ n$, where $n \in \mathbb{N}$ (using the rule *zero*). This is also known as the Peano definition of natural numbers.

The rules *zero* and *succ* can be encoded in Coq using `Inductive` (note the correspondence in the comments):

```
Inductive Nat :=
| O : Nat          (* zero *)
| S : Nat → Nat. (* succ *)
```

Let us take a deeper look at this Coq definition. The **Nat** type can be built using two constructors: O and S. O is basically a constant value of type **Nat** the same way Winter is a constant value for type **Season**. But S is more special because it can construct an infinite number of expressions of type **Nat**: since O has type **Nat**, then S O has type **Nat**, and S (S O) has type **Nat**, and

so on. In fact, we have written down a *representation* of natural numbers. The symbols O and S are just names with no special meaning. However, the way we *interpret* them comes from how we use them for computing. The most obvious meaning is that O is 0, S O is 1, S (S O) is 2, and so on.

Given the definition of **Nat**, how does Coq know that S (S O) has type **Nat**?

```
Check S (S O).
```

```
S (S O)
    : Nat
```

What actually happens is that Coq is able to compute a *derivation* of the fact that S (S O) is of type **Nat**. Using our inference rules *zero* and *succ*, this derivation looks like this:

$$\dfrac{\dfrac{\dfrac{\cdot}{O \in \mathbb{N}}\,zero}{S\ O \in \mathbb{N}}\,succ}{S\ (S\ O) \in \mathbb{N}}\,succ$$

Derivations are finite compositions of inference rules. The sequence of rules that are composed in the derivation above is *succ*, *succ*, and *zero* - read from bottom to top. Note that derivations should end up with axioms. Otherwise they are not finite anymore, that is, we cannot even use the term *derivation* in such cases.

It is worth mentioning that the rule

$$\frac{n \in \mathbb{N}}{S\ n \in \mathbb{N}}\,succ$$

has been used twice, for different values of n. We extract the relevant parts of the above proof to explain this better. First, n is (**S O**):

$$\frac{\mathbf{S}\ \mathbf{O} \in \mathbb{N}}{S\ (\mathbf{S}\ \mathbf{O}) \in \mathbb{N}}\,succ,$$

and second, n is **O**:

$$\frac{\mathbf{O} \in \mathbb{N}}{S\ \mathbf{O} \in \mathbb{N}}\,succ$$

This is why we say that *succ* is in fact a *rule scheme* because it can be instantiated on an infinite number of values for the metavariable n.

Exercise 2.1.3 *Consider the following inductive definition of list of naturals* $\mathbb{L}$*:*

$$\frac{\cdot}{Nil \in \mathbb{L}}\ nil \qquad\qquad \frac{n \in \mathbb{N} \quad l \in \mathbb{L}}{(Cons\ n\ l) \in \mathbb{L}}\ cons$$

Write the corresponding Coq code for this definition.

Exercise 2.1.4 *Recall the inductive definition of lists of natural numbers from Exercise 2.1.3. A possible derivation for* $(Cons\ O\ nil) \in \mathbb{L}$ *is:*

$$\frac{\dfrac{\cdot}{O \in \mathbb{N}}\ zero \quad \dfrac{\cdot}{Nil \in \mathbb{L}}\ nil}{(Cons\ O\ nil) \in \mathbb{L}}\ cons$$

Write on a piece of paper a derivation for $\big(Cons\ (S\ O)\ (Cons\ O\ nil)\big) \in \mathbb{L}$*.*

Exercise 2.1.5 *Provide an inductive definition for binary trees of natural numbers. Then provide a Coq implementation of it and write several non-trivial examples of derivations.*

2.1.3 Recursive functions

Now that we have seen how to create an inductive definitions, we can write functions that operate over them. Typically, functions that operate on inductively defined types are recursive. To explain this better, we recall the inductive definition of $\mathbb{N}$ and in the same time we write the addition function $+$ on naturals as a recursive function:

$$\frac{\cdot}{O \in \mathbb{N}}\ zero \qquad\qquad \frac{n \in \mathbb{N}}{S\ n \in \mathbb{N}}\ succ$$

$$\underline{O} + m = m \qquad\qquad \underline{S\ n} + m = S\ (n + m)$$

There is a correspondence between each inference rule and each case of our recursive definition of $+$. Note that $+$ has two arguments, and it is defined recursively on the first argument. The recursive definition has two cases: the base case which says what happens when the first argument is O (i.e., it is constructed using the axiom *zero*), and the recursive case which specifies how addition is made when the first argument is $S\ n$. The latter case includes a recursion because the sum of $\underline{S\ n} + m$ is expressed in terms of $n + m$.

This idea can be generalised. For example, a function which returns the length of a list defined as in Exercise 2.1.3 can be defined recursively as below:

$$length(Nil) = O \qquad length(Cons\ n\ l) = (S\ O) + length(l)$$

Exercise 2.1.6 *Write a recursive function that returns the last element of a list. If the last element does not exist, return O.*

Exercise 2.1.7 *Write a recursive function that computes the number of nodes of a binary tree. Use the inductive definition that you provided in Exercise 2.1.5.*

This definition of addition of natural numbers:

$$O + m = m \qquad (S\ n) + m = S\ (n + m)$$

is encoded in Coq using `Fixpoint` – a keyword that instructs Coq that this is a recursive function:

```
Fixpoint plus (n m : Nat) : Nat :=
  match n with
  | O ⇒ m
  | (S n') ⇒ S (plus n' m)
  end.
```

This function will compute for us the addition of two **Nat**s, as shown in the following experiments:

```
Compute plus O O.
    = O
    : Nat
Compute plus O (S O).
    = (S O)
    : Nat
Compute plus (S O) O.
    = (S O)
    : Nat
Compute plus (S O) (S O).
    = (S (S O))
    : Nat
```

If we consider that O is 0, S O is 1, S (S O) is 2, etc., the plus function seems to exhibit the expected behavior.

Exercise 2.1.8 *Implement the recursive functions from Exercises 2.1.6 and 2.1.7 in Coq and test them on several non-trivial inputs.*

2.1.4 bool and nat in Coq

As expected the most common types are already available in Coq. For instance, **bool** is defined in Coq as:

```
Print bool.
```

```
Inductive bool : Set := true : bool | false : bool
```

The `Print` command shows the definition of booleans in Coq. The definition is the expected one: we have two constructors `true` and `false` that correspond to the well-known truth values. More details about booleans in Coq are available in the documentation `https://coq.inria.fr/library/Coq.Bool.Bool.html`.

Exercise 2.1.9 *Use `Print` to show the definition of `nat` in Coq. Also, read the documentation: `https://coq.inria.fr/refman/language/core/inductive.html#simple-inductive-types`.*

Exercise 2.1.10 *Write a recursive function that returns if a natural number is even.*

Exercise 2.1.11 *Write a recursive function that computes the factorial of a natural number.*

Exercise 2.1.12 *Write a function that returns `true` if the input natural numbers n_1 and n_2 are in a less than relation, i.e., $n_1 < n_2$. The function will return `false` if $n_1 \geq n_2$.*

2.1.5 Proofs

In Coq we can state and prove various properties of interest about our functions or relations. Coq provides a very interesting mechanism that allows one to write proofs using *tactics*. These tactics generate a *proof object* which can be automatically checked! This process of checking a proof is called certification. The major benefit of certification is that proofs cannot be incomplete or even incorrect once they are mechanically checked. It is often the case when proofs written on paper contain mistakes or various corner cases are overlooked. This does not happen in Coq due to its bookkeeping features.

For those familiar with lambda calculus and type systems, the Coq tactics actually generate a term. Then, the Coq kernel checks if this term has the desired type. If so, the corresponding proposition (via Curry-Howard isomorphism) holds. Since type checking is actually a very simple problem to solve, the Coq kernel is not very complex and can be trusted.

Simplification

We start by stating and proving a very simple theorem about natural numbers. The text appearing in the rectangles in the proofs below are just additional comments about the proofs. These comments are not part of the actual Coq proof.

```
Theorem plus_O_n :
  ∀ n : Nat, plus O n = n.
Proof.
```

> At this point we start our proof. The current goal to prove is:
>
> ```
> 1 subgoal (ID 3)
>
> ============================
> forall n : Nat, plus O n = n
> ```
>
> Above the horizontal line we find our hypotheses (none so far), and below the horizontal line we find our goal, i.e., the conclusion that we want to prove.
> Intuitively, the proof goes like this: pick an arbirary natural number n and try to compute the plus function for that n. The result should be n. It is like saying that: "I want to arbitrarily choose a natural number n and prove that adding O to it would return n".
> The first step that we do is to actually pick n. For this we use the tactic `intro`. This tactic introduces the variable n in the context:

```
  intro n.
```

> Now, our goal looks like this:
>
> ```
> 1 subgoal (ID 4)
>
> n : Nat
> ============================
> plus O n = n
> ```
>
> Note that we have one hypothesis, namely n : **Nat** and our goal is now plus O $n = n$. At this point, for this (arbitrarily chosen) n we want to apply the definition of plus. The tactic `simpl` does that for us:

```
  simpl.
```

The `simpl` tactic applied the definition of plus and reduced our goal to:

```
1 subgoal (ID 7)

n : Nat
============================
n = n
```

This is now obvious: we only need to show that $n = n$! We do this by simply using the `reflexivity` tactic:

```
reflexivity.
```

Voila! We now receive the message: `No more subgoals`. which means that our proof object is now generated. The only thing left to do is to actually check the generated proof using the Coq trusted kernel. This is done using the `Qed` command at the end of the proof:

```
Qed.
```

We have now finished our first proof! We have expressed our property in Coq and then we used tactics to write down the proof. Our proof is now mechanically checked by Coq using `Qed` and it does not include any errors. To summarize, here is the Coq proof without comments:

```
Theorem plus_O_n :
  ∀ n : Nat, plus O n = n.
Proof.
  intro n.
  simpl.
  reflexivity.
Qed.
```

Exercise 2.1.13 *Prove in Coq that plus O O = O.*

Rewriting

The next theorem states that it is safe to replace two equal natural numbers in a particular addition:

```
Theorem plus_eq:
  ∀ m n, m = n → plus O m = plus O n.
Proof.
```

`intros`.

Note that the `intros` tactic moves m, n and H in the hypotheses:

```
1 subgoal (ID 15)

  m, n : Nat
  H : m = n
  ============================
  plus 0 m = plus 0 n
```

In the proof of the `plus_O_n` theorem we used `intro` followed by the name of the universally quantified variable n to move it from the goal to the assumptions. Here, we have more variables and a hypothesis! So, if we want to use `intro` we have to write it three times:

`intro` m.
`intro` n.
`intro` H.

You can see now the advantage of `intros` vs. `intro`: `intros` is more compact. By default, `intros` moves all the variables and hypotheses from the goal in the assumptions using, as much as possible, their names from the goal. If similar names are already in the assumptions, Coq will generate fresh names. If you want specific names for the assumptions, these names can be explicitly provided as below:

`intros` x y I.

This will actually generate the goal:

```
1 subgoal (ID 15)

  x, y : Nat
  I : x = y
  ============================
  plus 0 x = plus 0 y
```

which is basically the same thing as the previous goal. Just the names are different!

Let us take a closer look at our current goal:

```
1 subgoal (ID 15)

  m, n : Nat
  H : m = n
  ============================
  plus O m = plus O n
```

It looks like the only thing that we have to do is to use the hypothesis *H* and replace *m* by *n* in our goal. This is done via the `rewrite` tactic followed by the name of the hypothesis that we want to rewrite:

```
rewrite H.
```

By default, the `rewrite` tactic applies the equality from left to right. This yields the following goal, where *m* is replaced by *n* :

```
1 subgoal (ID 16)

  m, n : Nat
  H : m = n
  ============================
  plus O n = plus O n
```

However, this tactic can be applied so that the rewrite is done from right to left.

```
rewrite ← H.
```

Now, the generated goal is different, that is, *n* is replaced by *m*:

```
1 subgoal (ID 16)

  m, n : Nat
  H : m = n
  ============================
  plus O m = plus O m
```

Either way, this can be discharged using `reflexivity`:

```
reflexivity.
Qed.
```

Exercise 2.1.14 *Prove that for all naturals m n, plus m n = plus n n when m = n.*

Exercise 2.1.15 *Prove that for all naturals m n, plus m n = plus m m when m = n.*

Case analysis, Inversion

So far we have discussed simplification and rewriting, but in practice, more tactics are needed to discharge more complex goals. A common proof technique is case analysis: sometimes we need to analyse the possible values of some quantity and show that in each situation, our target property holds.

For example, if we add 1 to any natural number k, then $k + 1$ cannot be equal to 0. How do we prove something like that when k is a **Nat**? It looks like we can perform a case analysis on k:

- $k = \mathsf{O}$: in this case $k + 1 = \mathsf{O} + 1 = 1 = \mathsf{S}\ \mathsf{O}$ which is different from O because these expressions are build using *different* constructors.
- $k = \mathsf{S}\ k'$, where k' is some **Nat**: in this case $k+1 = (\mathsf{S}\ k')+1 = \mathsf{S}\ (k'+1)$ which is again different from O because these expressions are build using *different* constructors.

So, case analysis is performed by looking at the possible values of k that are given by the two constructors O and S. Moreover, note that constructors build different values, and thus, $\mathsf{S}\ k'$ and O are different by construction, for any k'.

Now that we have sketched a proof of our property, let us formalise and prove it in Coq:

```
Theorem plus_c_a:
  ∀ k, plus k (S O) ≠ O.
Proof.
  intros.
```

At this point, our goal is:

```
1 subgoal (ID 17)

  k : Nat
  ============================
  plus k (S O) <> O
```

Note that `simpl` will do nothing to our goal because Coq cannot reduce plus k (S O). So, we are going to perform a case analysis on the possible values of k. This is done via the `destruct` tactic:

```
  destruct k as [ | k'] eqn:E.
```

The `destruct` k does the actual case analysis, while `as` [| *k'*] *eqn*:*E* instructs Coq to use some custom names in our proof. Without `as` [| *k'*] *eqn*:*E* Coq simply generates automatically some names. Leaving Coq to generate the names is typically a bad practice because Coq makes confusing choices of names.
The [| *k'*] tells Coq to use specific names for the fresh variables that come from constructors. Separated by a vertical line |, we specify the desired variable names for each constructor. Our first constructor O has no arguments, and thus, we do not add variable names for it (we just do not write anything before |). Our second constructor S has one argument and we will call this argument *k'*.
The *eqn*:*E* instructs Coq to use custom names for the assumptions that are generated by the case analysis. You can actually see in our goals listing below that we have an assumption named *E*, which corresponds to the first case in the case analysis:

```
2 subgoals (ID 29)

  k : Nat
  E : k = O
  ============================
  plus O (S O) <> O

subgoal 2 (ID 31) is:
 plus (S k') (S O) <> O
```

The second goal (i.e., `subgoal 2`) is not focused, but using the `Show` 2 command (here, 2 is the number of the goal), we can display that goal:

```
subgoal 2 (ID 31) is:

  k, k' : Nat
  E : k = S k'
  ============================
  plus (S k') (S O) <> O
```

The name choice for the variable *k'* is specified by [| *k'*] , while *E* is the custom name of the equation that captures our current case. Both goals correspond to our two items in our informal proof, i.e., the case $k = \mathsf{O}$ and the case $k = \mathsf{S}\ k'$.

Now that we have to solve two goals, we are going to use a syntactic marker to make our proof easier to read (they are not mandatory), namely, we use a dash – for each case:

```
- simpl. unfold not. intro H. inversion H.
```

The sequence of tactics solves the first case of our case analysis. Let us show the effect of each tactic on the current goal:

```
1 subgoal (ID 29)

  k : Nat
  E : k = 0
  ============================
  plus 0 (S 0) <> 0
```

First, `simpl` does the simplification because now we know the value of k:

```
1 subgoal (ID 34)

  k : Nat
  E : k = 0
  ============================
  S 0 <> 0
```

Note that `<>` is the ASCII notation for $\neq$, which in Coq is defined as not (A = B). Second, we unfold the not and we obtain:

```
1 subgoal (ID 36)

  k : Nat
  E : k = 0
  ============================
  S 0 = 0 -> False
```

It is interesting to see that in Coq not (A = B) actually means (A = B) $\rightarrow$ False.

Our goal is now an implication, so we can add an assumption using `intro`:

```
1 subgoal (ID 37)

  k : Nat
  E : k = O
  H : S O = O
  ============================
  False
```

At this point, we have to prove False! This is weird, but in fact, there are some inconsistencies in our assumption *H*: the values S O and O cannot be equal. So, if *H* is false then we can prove anything! We use the `inversion` tactic on the *H* assumption: `inversion` will search for the constructors in both sides of the equality and it will deduce these two values cannot be equal. In fact, what `inversion` does in this case is to actually discover (based on the inductive definition of **Nat**) what conditions are necessary for this hypothesis to be true. In this case, there isn't one, so *H* is actually False and the goal is now proved:

```
1 subgoal (ID 31)

subgoal 1 (ID 31) is:
 plus (S k') (S O) <> O

This subproof is complete, but there are some
unfocused goals.
Focus next goal with bullet -.
```

The proof of the second goal is identical:

```
  - simpl. unfold not. intro H. inversion H.
Qed.
```

We show now the complete proof without comments:

```
Theorem plus_c_a:
  ∀ k, plus k (S O) ≠ O.
Proof.
  intros.
  destruct k as [ | k'] eqn:E.
```

```
  - simpl. unfold not. intro H. inversion H.
  - simpl. unfold not. intro H. inversion H.
Qed.
```

Exercise 2.1.16 *Fill the ... in the following proof:*

```
Theorem plus_c_a':
  ∀ k, plus k (S O) ≠ O.
Proof.
  intros.
  unfold not.
  intro H.
  destruct ...
  - ...
  - ...
Qed.
```

2.2 Induction

Proofs by induction are very common in mathematics and it is the same in Coq. To understand why induction is so important, let us prove that *for all* n, $n+0 = n$ using the definitions for naturals and addition from Section 2.1.3:

$$\frac{\cdot}{O \in \mathbb{N}}\ zero \qquad\qquad \frac{n \in \mathbb{N}}{S\ n \in \mathbb{N}}\ succ$$

$$O + m = m \qquad\qquad (S\ n) + m = S\ (n + m)$$

Although $n + 0 = n$ is a property that we take for granted it is not that trivial to prove it. The main problem is that none of the rules for addition apply in the left hand side of the equality. So the addition cannot be reduced using directly the definition. Moreover, even if we try case analysis on n, we get into troubles in the second case:

- if $n = O$, then $n + 0 = n$ becomes $O + O = O$, which reduces to $O = O$ by the first equation in the definition of addition; this is true by reflexivity.

- if $n = S\,n'$, then $n+0 = n$ becomes $(S\,n')+O = (S\,n')$; using the second equation in the definition of addition we get $S\ (n' + O) = (S\ n')$. But how to continue from here? The issue is that there is no information about $(n' + O)$ being equal to n'!

Obviously, for those that are already familiar with induction, it looks like the equality $n' + 0 = n'$ should be given by an inductive hypothesis. What is certain is that case analysis alone is not sufficient to prove this obvious property. So, how do we prove this by induction? What does it mean to make a proof by induction, and what are the insights of proofs by induction?

As we previously said, an inductive definition is a collection of rules that define how certain values are constructed. If you want to prove a property $P(n)$ where n is in $\mathbb{N}$, then naturally, you want to show that P holds for any possible way of creating n. So, you want to show that $P(O)$ holds and you also want to show that $P(S\ n)$ holds when $P(n)$ holds. We actually summarize this idea below:

$$\frac{\cdot}{O \in \mathbb{N}}\ zero \qquad\qquad \frac{n \in \mathbb{N}}{S\ n \in \mathbb{N}}\ succ$$

$$\frac{\cdot}{P(O)}\ base \qquad\qquad \frac{P(n)}{P(S\ n)}\ ind$$

Now, this might look a bit more familiar: if you want to prove $P(n)$ you have to prove $P(O)$ and $P(n) \to P(S\ n)$. We can recognize where the mathematical induction on naturals comes from! The first part, $P(O)$ is the base case of the induction, while $P(n) \to P(S\ n)$ is the inductive step. In the inductive step, one needs to prove $P(S\ n)$ using the inductive hypothesis $P(n)$. This is also known and the induction principle for naturals and it can be stated as follows:

$$\forall P.\big(P(O) \land (\forall n.P(n) \to P(S\ n))\big) \to \forall n.P(n)$$

Now, we can go back to our property *for all n,* $n + 0 = n$ and prove it by induction on n:

- base case: $P(O) : O + O = O$, which reduces to $O = O$ by the first equation in the definition of addition; this is true by reflexivity.
- inductive step: we know $P(n) : n + O = n$ and we prove $P(S\ n) : (S\ n) + O = (S\ n)$. But, using the second equation in the definition of $+$ we get $P(S\ n) : S\ (n + O) = (S\ n)$, and further, by rewriting the inductive hypothesis $P(n) : n + O = n$, we get $P(S\ n) : (S\ n) = (S\ n)$; this is true by reflexivity.

Now that we have shown why induction is so important, we move on and we show how all these are handled by Coq. The first thing to notice is

that Coq is able to automatically generate the induction principle for a given inductive definition. Recall our definition of **Nat** :

```
Inductive Nat :=
| O : Nat          (* zero *)
| S : Nat → Nat.  (* succ *)
```

When this is loaded in Coq, the tool automatically generates an induction principle called Nat_ind. The type of Nat_ind is precisely the induction principle for Nat:

```
Check Nat_ind.
```

```
Nat_ind : ∀ P : Nat → Prop,
        P O → (∀ n : Nat, P n → P (S n)) → ∀ n : Nat, P n.
```

The reserved keyword for proofs by induction in Coq is `induction`. In every proof by induction on Nat, the `induction` tactic uses the Nat_ind induction principle. We are now ready to prove *for all* n*,* $n + 0 = n$ in Coq:

```
Theorem plus_n_O_is_n:
  ∀ n, plus n O = n.
Proof.
```

> The current goal is:
>
> ```
> 1 subgoal (ID 24)
>
>   ============================
>   forall n : Nat, plus n O = n
> ```
>
> We are now ready to apply induction on n:

```
  induction n.
```

> The induction will generate two goals. First, it generates the base case:
>
> ```
> 1 subgoal (ID 29)
>
>   ============================
>   plus O O = O
> ```
>
> which can be easily discharged by the following tactics:

```
  - simpl. reflexivity.
```

The inductive step goal is shown here:

```
1 subgoal (ID 32)

  n : Nat
  IHn : plus n O = n
  ============================
  plus (S n) O = S n
```

Note the presence of the inductive hypothesis *IHn* in the list of assumptions. We start by applying `simpl`, and we obtain:

```
1 subgoal (ID 39)

  n : Nat
  IHn : plus n O = n
  ============================
  S (plus n O) = S n
```

Then we can rewrite the *IHn*, and we get:

```
1 subgoal (ID 40)

  n : Nat
  IHn : plus n O = n
  ============================
  S n = S n
```

Finally, this is a trivial goal discharged using reflexivity.

```
  - simpl. rewrite IHn. reflexivity.
Qed.
```

Exercise 2.2.1 *Complete the proof of the following theorem:*

```
Theorem plus_n_Sm_is_S_n_m:
  ∀ n m,
    plus n (S m) = S (plus n m).
Proof.
  intro n.
  induction n.
  - ...
  - ...
```

```
Qed.
```

Exercise 2.2.2 *The following proof is a bit longer and uses the result proved in Exercise 2.2.1. Inspect all the intermediate goals generated by Coq (including the ones when* `rewrite plus_n_Sm_is_S_n_m` *is used) and fill the … when needed:*

```
Theorem plus_exercise_1:
  ∀ n, plus n (plus n O) = plus n n.
Proof.
  induction n.
  - simpl. trivial.
  - simpl.
    rewrite plus_n_Sm_is_S_n_m.
    rewrite plus_n_Sm_is_S_n_m.
    ...
Qed.
```

Exercise 2.2.3 *Coq has a builtin type for naturals called* **nat***:* `https://coq.inria.fr/library/Coq.Init.Datatypes.html#nat`*. Prove the following theorems*[1]*:*

```
Theorem nat_plus_c_a:
  ∀ k>0, k + 1 ≠ 1.
Proof.
Admitted.

Theorem nat_plus_n_0_is_n:
  ∀ n, n + 0 = n.
Proof.
Admitted.

Theorem nat_plus_n_Sm_is_S_n_m:
  ∀ n m,
    n + (m + 1) = (n + m) + 1.
Proof.
Admitted.

Theorem nat_plus_exercise_1:
  ∀ n, n + (n + 0) = n + n.
Proof.
Admitted.
```

[1]The *Admitted* keyword instructs Coq that the proof is incomplete and the compiler will ignore that proof.

Exercise 2.2.4 *Recall the list of naturals from Exercise 2.1.3. Also, recall the* length *function (discussed in Section 2.1.3):*

$$length(Nil) = O \qquad length(Cons\ n\ l) = (S\ O) + length(l).$$

We implement all these in Coq and we also include a new function for list concatenation called append*. Prove the lemma at the end:*

```
Inductive ListNat :=
| Nil : ListNat
| Cons : Nat → ListNat → ListNat.

Fixpoint length(l : ListNat) :=
  match l with
  | Nil ⇒ O
  | Cons _ l' ⇒ S (length l')
  end.

Fixpoint append(l1 l2 : ListNat) :=
  match l1 with
  | Nil ⇒ l2
  | Cons x l1' ⇒ Cons x (append l1' l2)
  end.

Lemma append_len:
  ∀ l1 l2,
    plus (length l1) (length l2) = length (append l1 l2).
Proof.
Admitted.
```

2.2.1 Note to the reader

So far, we covered some very basic Coq tactics that are used to prove simpler facts. Obviously, more difficult proofs require more complex tactics. We recommend the readers to read the chapter dedicated to tactics available here [6]: `https://softwarefoundations.cis.upenn.edu/lf-current/Tactics.html`.

2.3 Polymorphism

Polymorphism comes in many flavours [2]: overloading, universal parametric polymorphism (explicit vs. implicit), and subtype universal polymorphism. In this section we address polymorphism from a functional programming perspective, namely: abstracting functions over the types of the data that they manipulate [6]. In [2], this is known as universal parametric polymorphism.

2.3.1 Polymorphic lists

We recall here our definition of list of natural numbers from Exercise 2.2.4:

```
Inductive ListNat :=
| Nil : ListNat
| Cons : Nat → ListNat → ListNat.
```

The length function below computes the length of such a list:

```
Fixpoint length(l : ListNat) :=
  match l with
  | Nil ⇒ O
  | Cons _ l' ⇒ S (length l')
  end.
```

An obvious question here is whether the length depends on the fact that it operates over elements of type **ListNat**. We can actually write a type for list of booleans and a function length', and we can observe that the length is computed in a similar way:

```
Inductive ListBool :=
| NilB : ListBool
| ConsB : bool → ListBool → ListBool.
```

```
Fixpoint length'(l : ListBool) :=
  match l with
  | NilB ⇒ O
  | ConsB _ l' ⇒ S (length' l')
  end.
```

The _ is an anonymous variable, that is, a variable whose name is not important. Note that both length and length' functions have a very similar implementation. Naturally, we can ask the following question: is it possible to have a single implementation for length that operates on *polymorphic lists*? The answer is *yes* and below we show how this is done. First, we define the polymorphic lists in Coq:

```
Inductive List (T : Type) : Type :=
| Nil : List T
| Cons : T → List T → List T.
```

In Coq, this is the polymorphic type definition for lists. It looks the same as the definitions of ListNat or ListBool, except that the type of the elements of the list is given as a parameter T : Type. **List** itself can now thought as a function that takes a type and gives a new inductive definition.

```
Check List.
List :
  Type → Type
```

T : Type is now a parameter in the definition of **List** and automatically, T becomes a parameter for all constructors! The types of Nil and Cons are:

```
Check Nil.
Nil
     : ∀ T : Type, List T
Check Cons.
Cons
     : ∀ T : Type, T → List T → List T
```

Once we have polymorphic lists, it is really easy to get back our previous implementations for lists of naturals and lists of booleans:

```
Definition ListNat := list Nat.
Definition ListBool := list bool.
```

Here are several examples of concrete lists of naturals and booleans. Note that we explicitly write the type as the first parameter:

```
Check Nil Nat.
Nil Nat
     : List Nat.
Check (Cons Nat O (Nil Nat)).
Cons Nat O (Nil Nat)
     : List Nat.
```

Exercise 2.3.1 *Check the output produced by the following Coq code:*

```
Check Nil bool.
Check (Cons bool true (Nil bool)).
```

Now that we have a polymorphic type for lists, we can write polymorphic version of functions. The length of a list is now parametric in the type of the elements of lists:

```
Fixpoint length (T : Type) (l : List T) :=
  match l with
  | Nil _ ⇒ O
  | Cons _ _ l' ⇒ S (length T l')
  end.
```

The type of length is:

```
Check length.
length
  : ∀ T : Type, List T → Nat.
```

This implementation of length can now be used on **all** lists. From a software engineering perspective, this is an implementation of the DRY ("do not repeat yourself") principle.

In order to use the function, you simply have to provide its arguments:

```
Compute length Nat (Nil Nat).
     = O
     : Nat

Compute length Nat (Cons Nat O (Nil Nat)).
     = S O
     : Nat.
```

Exercise 2.3.2 *Check the result returned by the following code:*

```
Compute length bool (Cons bool true (Nil bool)).
```

Does it work as expected?

Exercise 2.3.3 *Write a function called* repeat *that takes a element* e *(of any type!) and a natural number* n*, and returns a list of length* n *where every element in the list is equal to* e*. Test your function on several relevant examples.*

Implicit Arguments It is really cumbersome to write down expressions like this length **Nat** (Cons **Nat** O (Nil **Nat**)), where **Nat** is almost everywhere. Fortunately, in Coq there is a way to deal with this: we use the `Arguments` directive:

```
Arguments Nil {T}.
Arguments Cons {T}.
Arguments length {T}.

Check length (Cons O Nil).

length (Cons O Nil)
  : Nat.
```

The `Arguments` keyword is followed by the name of the function or constructor and then the (leading) argument names to be treated as implicit are given (surrounded by curly braces). Having such declarations for all constructors of a polymorphic type is really useful. For instance, we can write a more elegant definition for the length of a list (without so many _s as in the definition of length):

```
Fixpoint length' {T : Type} (l : List T) :=
  match l with
  | Nil ⇒ O
  | Cons _ l' ⇒ S (length' l')
  end.
```

Also, note that $\{T$: Type$\}$ (the argument with curly braces) allows us to use directly (length' l') instead of (length' T l').

Exercise 2.3.4 *Implement a polymorphic function that concatenates two lists. Test this function on non-trivial examples.*

Exercise 2.3.5 *Implement a polymorphic function that reverses a list. Test this function on non-trivial examples.*

2.4 Higher-Order Functions

It is well-known that functions in functional programming are first-class citizens. This means that in Coq we can easily pass functions as arguments to other functions, we can store functions or we can return functions as results.

2.4.1 Functions as parameters to other functions

We start by showing how to pass functions as arguments to other functions. A classical example is the filter function below, which filters a list using a predicate that is given as a parameter:

```
Fixpoint filter {T : Type} (f : T → bool) (l : List T) : List T :=
  match l with
  | Nil ⇒ Nil
  | Cons x xs ⇒ if (f x)
                then Cons x (filter f xs)
                else filter f xs
  end.
```

The function (f : T → **bool**) is a parameter for the filter function and it acts as a predicate: for any input it returns a boolean value. Our filter functions simply applies f to each element in the list, and it only keeps the ones that satisfy the predicate. Let us see how this works. First, add this import statement at the begining of your file:

```
Require Import Nat.
```

Then, we write down an example and we define a predicate for filter:

```
Example num_list := Cons 2 (Cons 15 (Cons 7 (Cons 18 Nil))).
Definition has_one_digit (n : nat) := leb n 9.
```

The leb function is defined in the Nat module and it implements the less than relation ($\leq$) for naturals. Our predicate is has_one_digit, and it checks if the given argument is a digit. The type of has_one_digit is:

```
Check has_one_digit.
has_one_digit
     : nat → bool.
```

Exercise 2.4.1 *Test the function* has_one_digit *on several examples.*

We can now use our filter by passing as arguments the predicate and the list to be filtered:

```
Compute filter has_one_digit num_list.
    = Cons 2 (Cons 7 Nil)
    : List nat.
```

The result is the expected one: the numbers that do not satisfy our predicate are simply eliminated from our list.

Exercise 2.4.2 *Write a higher order function*[2] *that applies an operation* ($f : T \to T$) *to all elements of a list. Test this function on some non-trivial examples.*

Hint: as an example, the function ($f : T \to T$) *could be a function that multiplies all elements of a list of naturals by 3, or it can divide them by 2.*

2.4.2 Anonymous functions

It is possible to define functions "on the fly", that is, without declaring them explicitly and using them by their name. Our function has_one_digit is defined explicitly, but it can be defined as an anonymous function as well:

```
Check (fun n ⇒ leb n 9).
(fun n ⇒  n ≤? 9)
    : nat → bool.
```

The expression (fun n ⇒ leb n 10) is read as the function that takes the input n and returns the result of the expression leb n 10[3]. Basically, it does the same thing as has_one_digit, but now we have a way to handle it the same way we handle values. As a consequence, we can pass this anonymous function to filter like this:

```
Compute filter (fun n ⇒ leb n 9) num_list.
    = Cons 2 (Cons 7 Nil)
    : List nat.
```

Exercise 2.4.3 *Test the higher-order function that you defined in Exercise 2.4.2 using anonymous functions.*

2.4.3 Functions that return other functions

So far we have seen that functions can take other functions as arguments. But we can also write functions that return other functions. The simplest function is the id function below, which takes as input a function and returns the same function:

```
Definition id {A : Type} {B : Type} (f : A → B) := f.
```

[2] Those familiar with functional programming may recognise that this is the well-known `map` function.

[3] Here, ≤? is just an infix notation for leb. Notations are discussed in the next section of these notes.

When applied to a function, id returns the same function. So the return type should be the same as the type of the initial function. Coq confirms that:

```
Check has_one_digit.
has_one_digit
     : nat → bool.
Check (id has_one_digit).
     : nat → bool.
```

Exercise 2.4.4 *Execute the command below and inspect the output:*

```
Compute has_one_digit 10.
Compute (id has_one_digit) 10.
Compute has_one_digit 1.
Compute (id has_one_digit) 1.
```

The functions has_one_digit and (id has_one_digit) return the same results?

Exercise 2.4.5 *Execute the following commands in Coq:*

```
Check filter.
Check (id filter).
```

1. *The functions filter and (id filter) have the same type?*
2. *Do these functions return the same values for the same inputs? Write some tests to confirm your answer.*

Of course, the id function is not that spectacular, but it shows that functions can return other functions. A more interesting example is function composition:

```
Definition compose
   {A : Type} {B : Type} {C : Type} (f : B → C) (g : A → B) :=
  fun x ⇒ f (g x).
Check compose.
compose
   : (?B → ?C) → (?A → ?B) → (?A → ?C)
where
?A : [ ⊢ Type]
?B : [ ⊢ Type]
?C : [ ⊢ Type]
```

The type of compose looks a bit odd, but what we need to understand is that it takes two functions, a function $f : B \rightarrow C$ that maps elements of type B to elements of type C, and a function $g : A \rightarrow B$ that maps elements of type A into

elements of type B. The output of compose is a new function of type $A \to C$, that is, it maps elements of type A to elements of type C by applying g and then f.

Let us take a look at an example. We provide two anonymous functions to compose, one that multiplies a number with 2 and another one that adds 3 to a given number. First, we look at the type returned by compose in this case:

```
Check compose (fun x ⇒ x × 2) (fun x ⇒ x + 2).
```

```
compose (fun x : nat ⇒ x × 2) (fun x : nat ⇒ x + 2)
    : nat → nat.
```

So compose (fun x : **nat** ⇒ x × 2) (fun x : **nat** ⇒ x + 2) will take a **nat**, say n, it will first apply (fun x : **nat** ⇒ x + 2) to it, and then it will apply (fun x : **nat** ⇒ x × 2) to the result. In the end, we should get $(n + 2) * 2$. Here is what happens when $n = 3$:

```
Compute compose (fun x : nat ⇒ x × 2) (fun x : nat ⇒ x + 2) 3.
    = 10
    : nat
```

By composing the functions the other way around, we get a different answer:

```
Compute compose (fun x : nat ⇒ x + 2) (fun x : nat ⇒ x × 2) 3.
    = 8
    : nat
```

Exercise 2.4.6 *What is the type of compose (fun x : **bool** ⇒ if x then Nil else (Cons 1 Nil)) (fun x : **nat** ⇒ leb x 0) ? Justify your answer and test it on several examples.*

Is it possible to switch the arguments of compose in this case? Justify your answer.

2.5 Logic in Coq and related tactics.

One of the best features of Coq is the possibility to state various propositions about the values and the types that we define. Another cool feature of Coq is the possibility to prove such propositions using tactics. Coq is a strongly typed language, that is, every expression that we write has an associated type. Propositions themselves have a type called **Prop**. One important distinction that we need to make is that being a proposition is different from being a provable proposition! For example, the two examples below are both propositions:

```
Check 10 = 10.
10 = 10
  : Prop
```

```
Check 10 = 11.
10 = 11
  : Prop
```

However, only the first is provable:

```
Goal 10 = 10.
Proof.
  reflexivity.
Qed.
```

```
Goal 10 = 11.
Abort.
```

As expected, propositions of type **Prop** can be combined using the well-known logical operations: negations, conjunctions, disjunctions, implications, equivalences, and quantifiers. The most natural question for a newcomer is why don't we just use the **bool** to state such properties? The main difference between **bool** and **Prop** is *decidability*. Expressions of type **bool** can always be simplified to true or false. In contrast, **Prop** includes also undecidable mathematical propositions.

2.5.1 Implication

Probably the most used logical connective in Coq is implication. This is because most of our propositions are formulated as implications. For example, if n is 0 then $n + 3 = 3$. In Coq this is simply stated as:

```
Lemma simple_impl :
  ∀ n, n = 0 → n + 3 = 3.
Proof.
  intros n H.
  rewrite H.
  simpl.
  reflexivity.
Qed.
```

The Coq environment is very well suited for handling implications. For example, the `intros` tactic knows how to extract the hypothesis $H : n = 0$ directly as from the implication.

When multiple implications are involved, Coq handles them in a conveniently using `intros`:

```
Lemma not_so_simple_impl :
  ∀ m n, m = 0 → n = 0 → n + m = 0.
Proof.
  intros m n Hm Hn.
  rewrite Hn.
  rewrite Hm.
  simpl.
  reflexivity.
Qed.
```

Note that both Hm : $m = 0$ and Hn : $n = 0$ are introduced as hypotheses as expected.

2.5.2 Conjunction

Conjunctions are used to express that two propositions are true in the time. Proving conjunctions is as easy as it sounds: you have to prove that the two propositions involved are true. This can be done via the `split` tactic. Here is an example:

```
Lemma simple_conjunction :
  2 + 3 = 5 ∧ 5 + 5 = 10.
Proof.
  split.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

The rationale here is simple: `split` creates two new goals that correspond to the two propositions in the conjunction. Each proposition is then proved separately.

It is impossible not to observe that we have some duplicated code in our proof. Both goals can be solved using the *same* sequence of tactics: `simpl. reflexivity.` Can we do better than this? Yes, we can:

```
Lemma simple_conjunction' :
  2 + 3 = 5 ∧ 5 + 5 = 10.
Proof.
  split; simpl; reflexivity.
Qed.
```

The semicolon ';' tells Coq to apply the next tactic to all the goals generated by the previous tactic. In our case, `split; simpl ; reflexivity` instructs Coq to apply `simpl` to all goals generated by `split`, and then to apply `reflexivity` to all the generated goals.

Conjunctions are handled similarly even in the presence of implications:

```
Lemma implication_and_conjunction:
  ∀ n, n = 0 → n + 3 = 3 ∧ n + 5 = 5.
Proof.
  intros n Hn.
  split; rewrite Hn; simpl; reflexivity.
Qed.
```

Sometimes, conjunctions can occur in the left hand side of an implication. This raises an issue, namely, how to use a conjunction as a hypothesis? A relevant example is shown here:

```
Lemma conjunction_as_hypothesis:
  ∀ m n, n = 0 ∧ m = 0 → n + 3 = 3.
```

At this point, the most obvious thing to do is to use the `intros` tactic:

```
intros m n Hmn.
```

This will generate the following goal:

```
1 subgoal (ID 61)

  m, n : nat
  Hmn : n = 0 /\ m = 0
  ============================
  n + 3 = 3
```

However, the hypothesis Hmn : $n = 0 \wedge m = 0$ is not very useful as it is. We cannot apply `rewrite` Hmn because we cannot rewrite an entire conjunction! What we need to do is to break this conjunction in two smaller hypotheses, so that the part that we are interested in, namely $n = 0$, can be further used to replace n with 0 in our goal. This can be done in two ways:

1. Either we use `intros` like this:

   ```
   intros m n [Hm Hn].
   ```

   and we will get the desired hypotheses directly:

   ```
   1 subgoal (ID 65)

   m, n : nat
   Hm : n = 0
   Hn : m = 0
   ============================
   n + 3 = 3
   ```

2. Or we use the `destruct` tactic, as below:

   ```
   intros m n Hmn.
   destruct Hmn as [Hn Hm].
   ```

   This will generate basically the same goal as above:

   ```
   1 subgoal (ID 66)

   m, n : nat
   Hn : n = 0
   ```

```
Hm : m = 0
============================
n + 3 = 3
```

From here, the sequence `rewrite Hn. simpl. reflexivity.` discharges the goal.

Here are the two variants side by side:

<pre>Proof.
  intros m n [Hn Hm].
  rewrite Hn.
  simpl.
  reflexivity.
Qed.</pre>
	<pre>Proof.
	  intros m n Hnm.
	  destruct Hnm as [Hn Hm].
	  rewrite Hn.
	  simpl.
	  reflexivity.
	Qed.</pre>

It is now easier to see that the variant from the left is shorter than the one in the right. This does not make the `destruct` tactic worse than `intros`. The `destruct` tactic can be used in other contexts as well, for example, it can generate a subgoal for every constructor of an inductive type. Moreover, `destruct` can be used to tackle disjunctions as well.

2.5.3 Disjunction

Disjunctions is another useful connective that captures the logical or: we say that $P_1 \lor P_2$ is true when either P_1 or P_2 is true.

Let us start by proving a simple disjunction in Coq:

```
Lemma simple_disjunction:
  2 + 3 = 5 ∨ 5 + 5 = 10.
Proof.
  left.
  simpl.
  reflexivity.
Qed.
```

Our goal here is a disjunction: $2 + 3 = 5 \lor 5 + 5 = 10$. In order to prove it, we either prove $2 + 3 = 5$ or $5 + 5 = 10$. The `left` tactic instructs Coq that we want to prove the left side of the disjunction, namely, $2 + 3 = 5$.

Since $5 + 5 = 10$ is also true, we can prove the same lemma using the `right` tactic (which is a dual of `left`):

```
Lemma simple_disjunction:
  2 + 3 = 5 ∨ 5 + 5 = 10.
Proof.
```

```
  right.
  simpl.
  reflexivity.
Qed.
```

Recall that only one component of the disjunction needs to be proved. Here is a proof of another lemma, where only the left hand side of the disjunction can be proved:

```
Lemma simple_disjunction':
  2 + 3 = 5 ∨ 5 + 5 = 11.
Proof.
  left.
  simpl.
  reflexivity.
Qed.
```

We have seen how to prove a disjunction, but it is also important to understand how to use a disjunction as a hypothesis. Here is a tricky lemma that we use to illustrate how to work with disjunctions as hypotheses:

```
Lemma disjunction_as_hypothesis:
  ∀ n, n = 0 ∨ 5 + 5 = 11 → n + 3 = 3.
```

We need to prove that $n + 3 = 3$ when the disjunction $n = 0 \lor 5 + 5 = 11$ is true. Since this disjunction is true, either of its components can be true, but we do not know which one! So what we need to do is a case analysis: first, we need to show that if $n = 0$ then $n + 3 = 3$, and second, we need to show that if $5 + 5 = 11$ is true then $n + 3 = 3$.

As in the case of conjunction, we have two variants to consider here:

1. Either we use

   ```
   intros n [Hn | Hn].
   ```

2. Or we use

   ```
   intros n Hn.
   destruct Hn as [Hn | Hn].
   ```

The vertical bar | in [*Hn* | *Hn*] determines Coq to generate two goals, one for each case of the disjunction:

```
n : nat
Hn : n = 0
============================
n + 3 = 3
```

and

```
n : nat
Hn : 5 + 5 = 11
============================
n + 3 = 3
```

Now each case needs to be proved separately. The first one is discharged using `rewrite Hn. simpl. reflexivity.` The second goal contains a false premise, namely $5 + 5 = 11$. Fortunately, this is easily discharged using `inversion Hn.`

Here is a complete proof of our lemma:

```
Lemma disjunction_as_hypothesis:
  ∀ n, n = 0 ∨ 5 + 5 = 11 → n + 3 = 3.
Proof.
  intros n [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - inversion Hn.
Qed.
```

In case one of the generated goals cannot be proved, then we cannot prove the entire lemma. For instance, the lemma below cannot be proved:

```
Lemma disjunction_as_hypothesis_unprovable:
  ∀ n, n = 0 ∨ 5 + 5 = 10 → n + 3 = 3.
Proof.
  intros n [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - (*  stuck *)
Abort.
```

The second goal generated by our case analysis cannot be proved because there is no information about n in our hypotheses. Intuitively, this lemma cannot be proved, because the disjunction $n = 0 \lor 5 + 5$ can be true because when $n = 0$ is false. Therefore, the conclusion cannot be proved.

Recall that we use $[Hn\ Hn]$ (i.e., without the '|') to deal with conjunctions in hypotheses, while for disjunctions we use $[Hn \mid Hn]$.

2.5.4 Negation

Typically, negations are used when we are interested in expressing that some propositions are *not* true. In logic and in Coq we use the $\neg$ symbol in front of the proposition that we want to negate. However, in Coq, $\neg P$ is in fact understood as an implication $P \rightarrow$ False. This is basically an equivalent formulation of the negation: $P \rightarrow$ False can be true only when P is False, that is, $\neg P$ is true.

We start by showing how to prove a negation. We pick a simple trivial lemma:

```
Lemma simple_negation:
  ∀ (x : nat), ¬ x ≠ x.
```

Naturally, we start our proof using `intro`.

```
Proof.
  intros x.
```

The generated goal is now this one:

```
1 subgoal (ID 85)

  x : nat
  ============================
  ~ x <> x
```

The tilde symbol occurring in the goal is the ASCII notation for $\neg$ in Coq. The next tactic that we apply is `unfold`:

```
  unfold not.
```

Now, our goal is the same, but the notations are unfolded:

```
1 subgoal (ID 87)

  x : nat
  ============================
  (x = x -> False) -> False
```

Now that we have an implication we know how to deal with it:

```
  intros Hx.
```

```
  1 subgoal (ID 88)

  x : nat
  Hx : x = x -> False
  ============================
  False
```

This goal looks a bit odd because we need to prove False. We are going to apply the *Hx* hypothesis:

```
  apply Hx.
```

What happens is that we can prove False if the left hand side of the *Hx* implication can be proved. Basically, if we know $A \to B$ and we want to prove B, it is sufficient to prove A. So, `apply` *Hx* does this for us, and the new goal is:

```
1 subgoal (ID 89)

  x : nat
  Hx : x = x -> False
  ============================
  x = x
```

This can be easily discharged using `reflexivity` and this concludes our proof:

```
  reflexivity.
Qed.
```

We summarise the proof of our lemma here:

```
Lemma simple_negation:
  ∀ (x : nat), ¬ x ≠ x.
Proof.
  intros x.
  unfold not.
  intros Hx.
  apply Hx.
  reflexivity.
Qed.
```

2.5.5 Existential quantification

Last but not least, we discuss existentially quantified properties. Below is a very simple example that illustrates the use of the `exists` tactic. In Coq, proving the existence of some value so that some property holds, requires us to provide that specific value. In our example, we need to pass the value of n that satisfies the property $n = 0$. Fortunately, there is only one such value, namely 0:

```
Lemma exists_zero:
  ∃ (n : nat), n = 0.
Proof.
  ∃ 0.
  reflexivity.
Qed.
```

Conversely, if an existentially quantified property is among our hypotheses, then we can use that specific value that satisfies the property in our reasoning. Here is an example:

```
Lemma exists_as_hypothesis:
  ∀ m, (∃ n, m = 2 + n) → (∃ n', m = 1 + n').
Proof.
  intros m [n Hmn].
  ∃ (1 + n).
```

```
  rewrite Hmn.
  simpl.
  reflexivity.
Qed.
```

Note that `intros` m [n Hmn] breaks the existentially quantified hypothesis into n : **nat** and Hmn : $m = 2 + n$. The variable n : **nat** satisfies Hmn, and thus, it can be further used in our proof (that is, we use it in $\exists$ (1 + n)).

2.5.6 Exercises

Exercise 2.5.1 *Prove the following property:*

```
Theorem ex_falso:
  ∀ P, False → P.
```

Exercise 2.5.2 *Prove the following property:*

```
Theorem and_elim_1:
  ∀ (A B : Prop), A ∧ B → A.
```

Exercise 2.5.3 *Prove the following property:*

```
Theorem and_elim_2:
  ∀ (A B : Prop), A ∧ B → B.
```

Exercise 2.5.4 *Prove the following property:*

```
Theorem and_intro:
  ∀ (A B : Prop), A → B → (A ∧ B).
```

Exercise 2.5.5 *Prove the following property:*

```
Theorem or_elim:
  ∀ (A B C: Prop), (A → C) → (B → C) → (A ∨ B) → C.
```

Exercise 2.5.6 *Prove the following property:*

```
Theorem or_intro_1:
  ∀ (A B : Prop), A → (A ∨ B).
```

Exercise 2.5.7 *Prove the following property:*

```
Theorem or_intro_2:
  ∀ (A B : Prop), B → (A ∨ B).
```

Exercise 2.5.8 *Prove the following property:*

```
Theorem not_not:
  ∀ (P : Prop), P → ~~P.
```

Exercise 2.5.9 *Prove the following property:*

```
Lemma exists_positive:
  ∃ (n : nat), n + 1 = 2.
```

Exercise 2.5.10 *Prove the following property:*

```
Lemma exists_as_hypothesis:
  ∀ m, (∃ n, m = 4 + n) → (∃ n', m = 2 + n').
```

Chapter 3

Syntax

A programming language is basically a language that allows us to write algorithms. Compilers and interpreters are able to *understand* the programs that implement the algorithms. But what is the magic behind this *understanding*? How does the computer *read* a program?

To answer this question we first need to understand what *syntax* means. Any language that we know uses an *alphabet*. The alphabet is just a set of *symbols* (e.g., the Latin alphabet, the Cyrillic alphabet) that are used to construct *words* or *tokens*. Not all combinations of alphabet symbols are accepted in languages. For example, in english, *sad* is a a word, but *dsa* is not part of the english dictionary. In linguistics, the *lexical corpus* or *lexis* defines the complete set of words in a language. The words can be combined in order to create *sentences*. The *grammar* is just a set of rules that describes the meaningful phrases of the language. For instance, *I am sad* does have a meaning in english, but *Sad am* is meaningless.

So, for a programming language, we have a similar setup: the alphabet contains the set of symbols that can be used to write programs in that language; the tokens are the legal words admitted in programs (e.g., keywords), while programs themselves are constructed by following a set of syntactical rules. These rules form the *grammar* of the language. Basically, the *syntax* of a programming language has three components: an alphabet, a lexical component, and a grammar.

Nowadays, programming languages come with a well documented formal syntax specification. For instance, the Java programming language uses the *Unicode* characters `https://www.unicode.org/` as alphabet and the lexical structure is completely described at `https://docs.oracle.com/javase/specs/jls/se16/html/jls-3.html`. The grammar of Java is available at `https://docs.oracle.com/javase/specs/jls/se16/html/jls-2.html`.

The American linguist Noam Chomsky developed techniques, known as *generative grammars*, to describe syntax in a formal manner. His proposal of using a formal way to describe the syntax has a major impact: it avoids the ambiguities that are always present in natural languages. From a programming language design perspective, the generative grammars are a fundamental tool for describing

the syntax of programming languages.

In order to illustrate the power of generative grammars we pick a very simple language called *PAL*. The alphabet of this language contains only two symbols a and b. What is interesting about this language is that it contains all palindromic strings that can be formed using a and b. For example, aba is part of the language, while $abbaa$ is not, since it is not a palindrome. One may think that it is really easy to write a regular expression that captures the set of palindromes in *PAL*, but in fact, this is impossible! We are not going to explain the details here, but the regular expressions have an equivalent finite state machine that accepts/rejects the same words as the regular expressions. Since the set of palindromes over a,b is infinite, it is impossible to build a finite state machine (it would need an infinite number of states). But here comes the power of grammars! They allow recursive definition of strings. In our case, the palindromic strings can be expressed by the following grammar:

1. $P \rightarrow \epsilon$
2. $P \rightarrow a$
3. $P \rightarrow b$
4. $P \rightarrow aPa$
5. $P \rightarrow bPb$

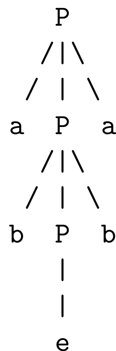
The P above is a *nonterminal* that stands for "any palindromic string". The a, and b symbols are called *terminals*. The arrow $\rightarrow$ indicates a grammar *production*. The production labelled 1 says that P can be an empty string (here denoted by ϵ). The productions 2 and 3 say that P can also be a or b. So far, we covered the base cases, where palindromes can be either ϵ, a or b. The recursive cases are captured by productions 4 and 5, where a new palindrome can be constructed by surrounding another palindrome with as or bs. In the literature, this is called a *context-free* grammar. More details about context-free grammars can be found in [2].

3.1 Derivations

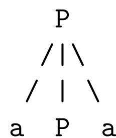
Given these grammar rules and an input, how can we establish whether the input is constructed according to the rules? The recursive nature of the grammar allows us to build *derivations*. The main idea is to read the productions (given by the arrows) as rewrite rules and try to find a finite sequence of rewrite steps for the given input. For instance, the following rewriting sequence is a derivation for $abba$:

- $\mathrm{P} \rightarrow^4 a\mathrm{P}a \rightarrow^5 ab\mathrm{P}ba \rightarrow^1 abba$.

Note that (finite) derivations that start with an initial nonterminal (e.g., P) and end up with a string that contains only terminal symbols actually define the set of all correct strings w.r.t. to our grammar. The above derivation indicates that *abba* is indeed a palindrome. Moreover, note that the derivation for the input *abba* is *unique*. For this derivation we have a corresponding *derivation tree* or *parse tree* (e stands for ϵ):



Parse trees are representations for derivations. These trees contain nodes labeled with terminals, nonterminals and ϵ. The interior nodes are always labeled with nonterminals. The root is labeled with P - our nonterminal for "any palindromic string". The children of an internal node are given by the corresponding production. For example, the production 4 yields this subtree:



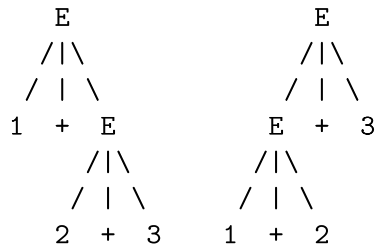
If a node has ϵ as a child, then ϵ is the *only* child.

3.2 Ambiguities

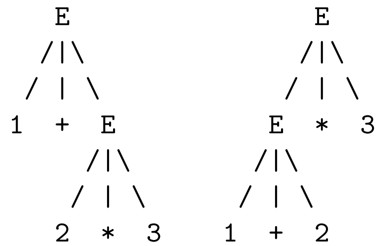
Grammars can also be ambiguous, that is, for some input, there are more than one possible parse trees. An example is the following grammar for simple arithmetic expressions with addition and multiplication:

1. $E \rightarrow nat$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$

We use *nat* to denote the natural numbers and we consider that any natural number is a terminal. For the input $1 + 2 + 3$ we obtain two parse trees:



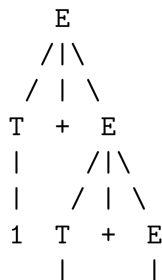
In this case we say that the parsing is ambiguous. An ambiguous grammar is useless from the perspective of a programming language designer because it cannot be used to compile/translate programs in a unique fashion. For our example, the evaluation of the addition produces the same result, i.e., 6, but for $1 + 2 \times 3$ the parse trees may yield different results (7 vs. 9):

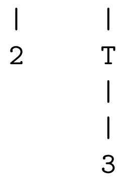


Obviously, we have to add to our grammars more information for disambiguation. One idea is to add more complexity to the grammar so that parsing will output only one possible parse tree:

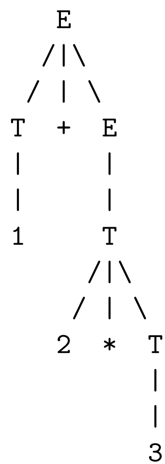
1. $E \to T$
2. $E \to T + E$
3. $T \to nat$
4. $T \to nat * T$

Now, the only parse tree for $1 + 2 + 3$ is (note: the parse tree mimics a right associative addition):



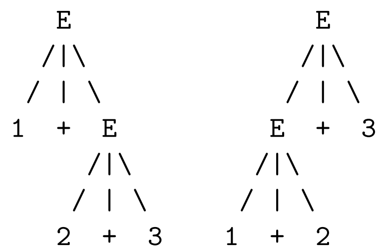


Moreover, the only parse tree for $1 + 2 \times 3$ is:

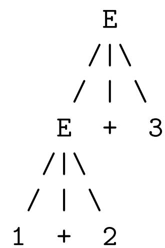


The parse trees are indeed larger than before, but at least the grammar is not ambiguous anymore.

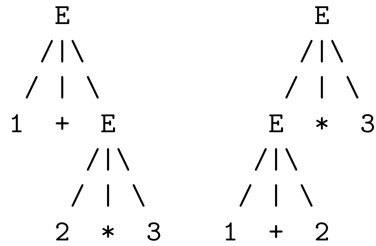
Another way to deal with ambiguities is to generate all the possible parse trees and then filter out the parse trees that do not meet a certain criteria. For example, one could keep only the parse trees that correspond to left associativity of +, that is, between the trees:



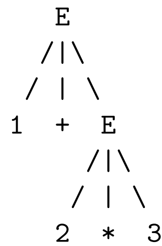
only the tree in right hand side is kept:



Also, the parse trees where + appears below * can be removed, that is, between:



we only keep the tree in the left hand side, which corresponds to the expected mathematical order of the operations:



The latter disambiguation solution is typically implemented using grammar annotations.

3.3 Abstract vs. Concrete Syntax

Context-free grammars define the *concrete syntax* of programming languages. The concrete syntax defines the way programs look like to the programmers.

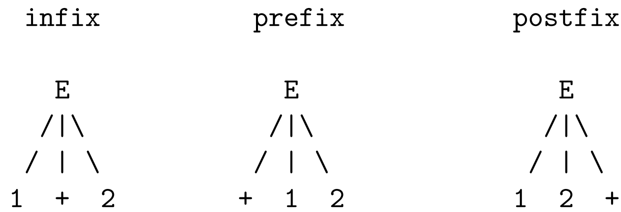
For instance, an arithmetic expression can be written in different ways:

- 1 + 2 – infix notation
- (+ 1 2) – prefix notation
- (1 2 +) – postfix notation

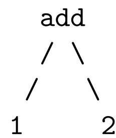
Obviously, this can be done by tweaking our grammar production rule for addition as below:

- $E \to E + E$ // infix
- $E \to + E\ E$ // prefix
- $E \to E\ E\ +$ // postfix

Now the corresponding parse trees for these expressions are:



However, a more intuitive way of expressing an addition is the following tree:



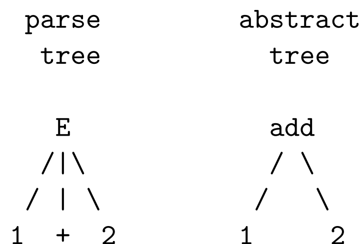
This is an equivalent *abstract* representation of **all** the parse trees above. This kind of structure is way more appropriate to be used by the compilers or interpreters. In practice, the concrete syntax is decoupled by the abstract representation which is given by an *abstract syntax*. The *abstract syntax* is the set of rules that define how programs look like to an interpreter or compiler.

This distiction between abstract and concrete syntax is essential because it separates the concerns. For instance, the abstract representation of an addition is basically the same in all programming languages, while the concrete syntax could be different (e.g., C vs. Lisp). Therefore, we can have a single abstract tree and many corresponding concrete syntax representations.

The transition from concrete syntax to abstract syntax can be done in various ways, but the most common approach is to annotate production rules in the concrete syntax with labels:

- $E \to E + E$ (label: *add*)

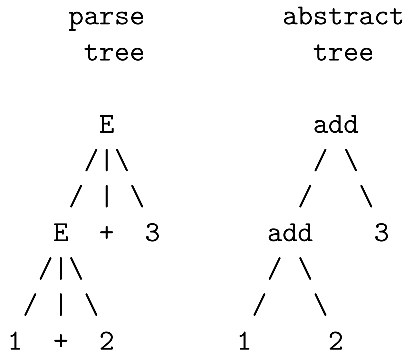
The label is a *constructor* name that will be part of the abstract syntax. Constructors have as many arguments as the number of nonterminals in the production (e.g., *add* has two arguments that need to be expressions E as well). The order of the arguments of the contructors is the same as the order of the nonterminals that appear in the production. Terminals are simply replaced by the constructor. An example of transformation is shown here:



Other annotations can indicate whether a certain language construct is left or right associative. For instance, the following production rule

- $E \rightarrow E + E$ (label: *add*, assoc: *left*)

yield the following trees for 1+2+3:

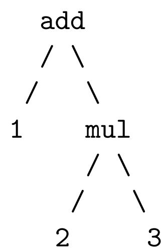


Recall that 1+2+3 is ambiguous without this associativity annotation in the production rule. An ambiguous grammar generates more than one parse tree for the same input. As a rule, in such cases, the parsers should report an error or they should pick one parse tree that will be then used to construct abstract syntax tree.

Other types of annotations in the grammars are precedence levels.

1. $E \rightarrow nat$
2. $E \rightarrow E + E$ (label: *add*, assoc: *left*, prec: *50*)
3. $E \rightarrow E * E$ (label: *mul*, assoc: *left*, prec: *40*)

Here, the numbers *50* and *40* are used to establish the precedence relationships between addition and multiplication. A lower number says that the annotated construct has a higher precedence than all the other constructs annotated with higher numbers. In our case, multiplication has a higher precedence than addition. With these annotations, the expression $1 + 2 \times 3$ is not ambiguous anymore, and the corresponding abstract syntax tree is:



From now on, we will only use abstract syntax trees. The transition from concrete syntax to abstract syntax is itself an interesting topic and there is a lot of literature covering techniques and algorithms for parsing (e.g., [**?**]).

3.4 BNF

Context-free grammars were used for the first time to define the syntax of Algol60. The notation used to express the grammar for Algol60 was different from the notation that we have seen previously, that is:

1. $E \rightarrow nat$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$

The new notation that was actually used for Algol60 is called BNF (Backus Naur Form) and it looks like this:

- E ::= *nat* | E + E | E * E

Indeed, this notation is way more compact and it has been widely adopted. Moreover, the BNF notation has been improved in various ways so that it can express optional constructs, alternative productions and repetitions. The family of extensions to BNF is known as Extended BNF (EBNF)[1].

In this material we are going to use the BNF notation.

3.5 Abstract Syntax in Coq

We are now ready to define the abstract syntax of a simple imperative language in Coq. Given the BNF grammar of arithmetic expressions

- E ::= *nat* | E + E | E * E

the corresponding Coq encoding is:

```
Inductive Exp : Type :=
| num : nat → Exp
| plus : Exp → Exp → Exp
| mul : Exp → Exp → Exp.
```

The num, plus, and mul are the constructors, while **nat** is the type of natural numbers. 1, 2, 1+2, and 1+2*3 are expressions which are written using the Coq abstract syntax as below:

```
Check (num 1).
Check (num 2).
Check (plus (num 1) (num 2)).
Check (plus (num 1) (mul (num 2) (num 3))).
```

[1]More details about BNF and EBNF can be found at http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html and https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

Coercion The Coq representation of an expression is now a bit more complicated than we expect: (plus (num 1) (mul (num 2) (num 3))) vs. $1 + 2 \times 3$. Fortunately, we can make some improvements so that programs are easier to read and write.

The first improvement is to get rid of the num construct when it can be automatically inferred. For instance, in this expression (plus (num 1) (num 2)) it is obvious that plus expects two arguments of type **Exp**. For this we are going to use Coq's Coercion mechanism which says that a constructor (or a function) can be used by the type system to coerce a value of the input type to a value of the output type. Let us declare a coercion for num:

```
Coercion num : nat >-> Exp.
```

The effect of this declaration is explained by the following Check:

```
Check (plus 1 2).
```

Using the above coercion Coq was able to infer the missing num before 1 and 2. If you try this before the Coercion declaration the type system will complain. In order to explicitly display the coercions this option is available:

```
Set Printing Coercions.
Check (plus 1 2).
```

Now, let us try the same on the other complicated expression:

```
Check (plus 1 (mul 2 3)).
```

This definitely looks better and it resembles the abstract syntax tree of the expression $1 + 2 \times 3$.

Notations The next thing that we are going to do is to add some notations so that our Coq representation looks almost like the initial expression.

```
Notation "X +^ Y" := (plus X Y) (at level 50, left associativity).
Notation "X *^ Y" := (mul X Y) (at level 40, left associativity).
```

These notations define our grammar together with the associativity and precedence annotations. Now we can easily write expressions as follows:

```
Check 1 +^ 2.
Check 1 +^ 2 *^ 3.
```

One can actually check if the parsed output is the expected one by enabling the following flag:

```
Set Printing All.
Check 1 +^ 2.
Check 1 +^ 2 *^ 3.
Unset Printing All.
```

3.5.1 Custom Grammar

So far we used +^ and *^ to avoid the conflict with the default notations + and * already available in Coq. However, we can actually use + and * if we really want to. This can be done via *Declare Custom Entry* which instructs Coq to use a custom grammar that we define for our language. Since we are going to define new interpretations for + and * we need to create our own scope. Then, we add the desired notations starting with a special notation: anything between `<[` and `]>` should be parsed with our custom grammar.

```
Declare Custom Entry exp.
Declare Scope exp_scope.
Notation "<[ E ]>" := E (at level 0, E custom exp at level 99) : exp_scope.
Notation "( x )" := x (in custom exp, x at level 99) : exp_scope.
Notation "x" := x (in custom exp at level 0, x constr at level 0) : exp_scope.
Notation "f x .. y" := (.. (f x) .. y)
                        (in custom exp at level 0, only parsing,
                        f constr at level 0, x constr at level 9,
                        y constr at level 9) : exp_scope.
Notation "X + Y" := (plus X Y) (in custom exp at level 50, left associativity).
Notation "X * Y" := (mul X Y) (in custom exp at level 40, left associativity).
```

Now, we can use our notation inside the *exp_scope*:

```
Open Scope exp_scope.
Check <[ 1 + 2 × 3 ]>.
Check <[ (1 + 2) × 3 ]>.
Close Scope exp_scope.
```

The **IMP** section in the second book of Software Foundations available at `https://softwarefoundations.cis.upenn.edu/current/lf-current/Imp.html` contains a more detailed description of the parsing features of Coq.

3.6 A simple imperative language: IMP

Now that we know how to define syntax in Coq, let us define the syntax of a simple imperative language, hereafter called **IMP**. This language includes arithmetic expressions, boolean expressions, assignment, conditionals[2], loops and sequences of statements. Our goal is to parse several meaningful programs using our Coq encoding of the syntax.

Arithmetic expressions have been defined previously. The only addition that we do here is a constructor for variables, that uses the builtin *string* type. So, our variables are actually strings for now:

```
Inductive AExp :=
```

[2]Conditionals are not added immediately, they are left as an exercise for the reader!

```
| avar : string → AExp
| anum : nat → AExp
| aplus : AExp → AExp → AExp
| amul : AExp → AExp → AExp.

Coercion anum : nat >-> AExp.
Coercion avar : string >-> AExp.
Notation "A +' B" := (aplus A B) (at level 50, left associativity).
Notation "A *' B" := (amul A B) (at level 40, left associativity).
```

Here are several examples of expressions that we can write so far:

```
Open Scope string_scope.
Check 2 +' (avar "x").
Check "i" +' 1.
Check "s" +' "i".
```

Boolean expressions include the btrue and bfalse constants together with the propositional operators bnot and band. Also, comparisons between arithmetic expressions are boolean expressions too:

```
Inductive BExp :=
| btrue : BExp
| bfalse : BExp
| bnot : BExp → BExp
| band : BExp → BExp → BExp
| blessthan : AExp → AExp → BExp
| bgreaterthan : AExp → AExp → BExp.
```

We add several useful notations for boolean expressions too:

```
Notation "A <' B" := (blessthan A B) (at level 80).
Notation "A >' B" := (bgreaterthan A B) (at level 80).
Infix "and'" := band (at level 82).
Notation "! A" := (bnot A) (at level 81).

Check btrue.
Check bfalse.
Check ! ("i" <' "n").
Check btrue and' ("n" >' 0).
```

We now define the statements of our language together with some useful notations:

```
Inductive Stmt :=
| assignment : string → AExp → Stmt
| while : BExp → Stmt → Stmt
| seq : Stmt → Stmt → Stmt.

Notation "X ::= A" := (assignment X A) (at level 85).
```

```
Notation "S1 ;; S2" := (seq S1 S2) (at level 99).
```

Finally, we can write some meaningful programs using our syntax. Here is the *sum* program which computes the sum of the first "*n*" natural numbers:

```
Check "n" ::= 10 ;;
      "s" ::= 0 ;;
      "i" ::= 0 ;;
      while ("i" <' "n" +' 1) (
            "s" ::= "s" +' "i" ;;
            "i" ::= "i" +' 1
      ).
```

3.7 Exercises

Exercise 3.7.1 *Extend the syntax of arithmetic expressions with operations for division, modulo, and minus.*

Exercise 3.7.2 *Extend the syntax of boolean expressions with disjunctions, equality for arithmetic expressions and other known comparison operators (e.g.,* <=*,* >=*).*

Exercise 3.7.3 *Append an* if_then_else_ *statement to the syntax of* **IMP** *and write a program that computes in a variable "max" the maximum between two inputs "x" and "y".*

Exercise 3.7.4 *Append an* if_then_ *statement to the syntax of* **IMP** *and rewrite the program you wrote for Exercise 3.7.3 so that it does not use an* else *branch.*

Exercise 3.7.5 *Append a* for *loop statement to the syntax of* **IMP** *and reformulate the* sum *program using* for*.*

Chapter 4

Semantics

Semantics (of programming languages) is concerned with the *meaning* of the syntactically valid sentences in programs. Each syntactical construct of a programming language is intended to exhibit a certain behavior. But how do we specify that behavior?

One approach is to create a reference document that explains, in natural language, the semantics of the programming language. For instance, the ISO standard for the C language [**?**] is a reference document with over 500 pages that defines the interpretation of C programs. Another approach is to provide a reference implementation (e.g., Perl [**?**]), which is a program that implements all the features of the language and demonstrates what constitutes correct behavior for a program.

A more systematic approach is offered by formal semantics, which encompasses the mathematical and logical tools and techniques needed to precisely define the meaning of programs. One of the primary advantages of formal semantics is precision. Unlike other approaches, where certain corner cases may be overlooked or under-specified, formal semantics allows for precise coverage of all aspects of programming languages. Relying solely on a natural language specification or an incomplete reference implementation may result in ambiguities in the specification, or various language constructs may have undefined behavior [**?**, **?**].

Another significant advantage of formal semantics is that it enables reasoning about programs. None of the previously mentioned approaches (a reference manual or a reference implementation) provides the means to reason about programs. Certain tools and techniques can be used in conjunction with formal semantics to detect various problems in programs, such as typing errors, runtime errors, compilation errors, memory errors, dead code, divisions by zero, and more. This stands as a substantial advantage of formal semantics when compared to non-formal methods for defining the semantics of programming languages.

In [**?**], a classification of the conventional types of formal semantics is presented: *static semantics*, *dynamic semantics*, and *equivalences* between programs. Here, we recapitulate this classification:

1. *Static semantics* which models compile-time checks. This includes well-formedness checks of the programs (i.e., the abstract syntax tree is correctly formed by the syntax rules) and type-checking (i.e., the program is correctly typed). Static semantics is concerned only with checks that do not require the execution of the program.

2. *Dynamic semantics* which models run-time behaviour. This kind of semantics concerns the observable behaviour when programs are executed. There are more styles of dynamic semantics:

   - *Operational Semantics* where computations are given as inference rules;
   - *Denotational Semantics* where each language construct is assigned with a computational meaning;
   - *Axiomatic Semantics* which models the relationships between preconditions and postconditions of each language construct; this style is more suitable for program verification.

3. *Equivalences* between programs require abstract models. Each program should have an abstract model that captures only the relevant features of all possible executions of that program. Programs are semantically equivalence when their corresponding abstract models are the same.

This classification shows us that each level of formal semantics is concerned with different aspects of programming languages (e.g., static semantics enables type-checking, dynamic semantics models program execution, axiomatic semantics models program verification). However, recent research in the domain of programming languages comes with a unified approach, where the one and only formal semantics of a language is the central pillar wherefrom all the other related tools and techniques should be derived: parsers, interpreters (*executable* semantics), compilers, static analysers, symbolic execution engines, testing tools, certified program verifiers, and others. This approach is yet incipient w.r.t. tool support, but the theoretical grounds and research have already been established: [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

In the rest of this chapter we discuss various ways of defining the semantics of programming languages. We start by building a simple evaluator for **IMP** in Coq. With this evaluator we are going to execute **IMP** programs and we emphasize on the limitations of this approach. To overcome these limitations we switch to *Structural Operational Semantics* (hereafter short-handed as SOS), a more general framework for defining programming languages. We emphasize on the main principles of the various SOS styles and we discuss their strengths and weaknesses. Again, we use the **IMP** language as a running example and we will provide the corresponding implementations. We use the Coq proof assistant for implementations, but keep in mind that all these SOS techniques are independent from Coq.

4.1 An evaluator for IMP

The easiest way to build an interpreter for a language is to write some functions that simply evaluate the programs written in that language. This is what we attempt to do in this section for the **IMP** language.

4.1.1 Environment

In order to evaluate an arithmetic expression, one needs an *environment*, that is, a binding map that maps program variables to values. This is because arithmetic expressions in **IMP** include variables and the evaluation of an arithmetic expression with variables requires the values of these variables. In Coq, binding maps can be modelled using functions:

```
Definition Env := string → nat.
```

Here is a particular example of a binding map (or environment) that maps the program variables "n" to 10, "i" to 0, and any other variable is mapped to 0:

```
Definition sigma1 := fun var ⇒ if (string_dec var "n")
                                 then 10
                                 else if (string_dec var "i")
                                      then 0
                                      else 0.
```

When interrogated, the environment returns the corresponding value for a given program variable:

```
Compute sigma1 "n".
    = 10
    : nat
Compute sigma1 "i".
    = 0
    : nat
Compute sigma1 "x".
    = 0
    : nat
```

Exercise 4.1.1 *Define an environment that returns values different from 0 for all the program variables occuring in the* sum *program below:*

```
"n" ::= 10 ;;
"s" ::= 0 ;;
"i" ::= 0 ;;
```

```
while ("i" <' "n" +' 1) (
        "s" ::= "s" +' "i" ;;
        "i" ::= "i" +' 1
).
```

The way we define the environment is by means of a *total* function, that returns **nat** for every possible input string. However, the environment is not necessarily a total function, but a *partial* one: we want it to return a **nat** only for a subset of program variables, i.e., the set of declared variables. In **IMP**, variables are declared when they appear in the left hand side of an assignment. In our current approach, we cannot know for sure whether a variable is declared and has value 0, or the variable is not declared and has the default value 0. For instance, the values of "i" and "x" in **sigma1** are equal to 0, but only "i" is declared.

A better formalisation for the environment is the use of the **option** type which is already defined in Coq. `Print` **option** will display this type:

```
Inductive option (A : Type) : Type :=
| Some : A → option A
| None : option A.
```

Using the **option** type we can define the above environment as below:

```
Definition Env' := string → option nat.
Definition sigma1' := fun var ⇒ if (string_dec var "n")
                                 then Some 10
                                 else if (string_dec var "i")
                                      then Some 0
                                      else None.
```

Now, it is easier to distinguish between declared (e.g., "i") and undeclared (e.g., "x") variables:

```
Compute sigma1' "n".
   = Some 10
   : nat
Compute sigma1' "i".
   = Some 0
   : nat
Compute sigma1' "x".
   = None
   : nat
```

Exercise 4.1.2 *Solve the Exercise 4.1.1 using the* **option** *type.*

Assignments in programs are meant to change the environment. Therefore we need a way to modify the environments. Since our environments are functions, we can write a higher-order function that takes an environment and returns a new one, given some additional arguments. The `update` function below, updates an environment by modifying the value of the given variable with a new value:

```
Definition update (env : Env) (var : string) (val : nat) : Env :=
  fun x ⇒ if (string_dec var x)
            then val
            else (env x).
```

The result of `update` is a new function that, for an input x, returns val (i.e., the new value of var) if var and x are the same; otherwise, it simply returns (env x), i.e., the value of x in the old environment.

Neat! We defined a higher-order function that builds new environments. The code below shows that our function works as expected:

```
Definition sigma2 := update sigma1 "n" 11.
Compute sigma2 "n".
   = 11
   : nat
Compute sigma2 "i".
   = 0
   : nat
```

Exercise 4.1.3 *Define an update function for the environment:*

```
Definition Env' := string → option nat.
```

Test the function on several relevant examples (declared variables, undeclared variables).

4.1.2 Evaluator for arithmetic expressions

We are now ready to evaluate arithmetic expressions with variables:

```
Inductive AExp :=
| avar : string → AExp
| anum : nat → AExp
| aplus : AExp → AExp → AExp
| amul : AExp → AExp → AExp.

Coercion anum : nat >-> AExp.
Coercion avar : string >-> AExp.
Notation "A +' B" := (aplus A B) (at level 50, left associativity).
Notation "A *' B" := (amul A B) (at level 40, left associativity).
```

A function that evaluates arithmetic expressions should take as input the arithmetic expression and an environment. The output should be a number, that is, the value of that expression. A simple recursive function that traverses the abstract syntax tree of the expression does the trick:

```
Fixpoint aeval (a : AExp) (sigma : Env) : nat :=
  match a with
  | avar x ⇒ (sigma x)
  | anum n ⇒ n
  | aplus a1 a2 ⇒ (aeval a1 sigma) + (aeval a2 sigma)
  | amul a1 a2 ⇒ (aeval a1 sigma) × (aeval a2 sigma)
  end.
```

The recursive aeval function has two base cases and two recursive cases. If the expression is a variable then its value is retrieved from the environment. If the expression is a number, then aeval returns that number. The cases corresponding to addition and multiplication require the recursive evaluation of the arguments before the actual operations are performed.

Note that aeval assigns a meaning to the syntactical constructs of the language: the syntactical construct aplus is associated with the mathematical addition (here, the + from Coq), and the syntactical construct amul is associated with the mathematical addition (here, the × from Coq).

Here are two examples that illustrate how aeval behaves when used with different environments:

```
Compute aeval ("n" +' 2 *' "n") sigma1.
  = 30
  : nat
Compute aeval ("n" +' 2 *' "n") sigma2.
  = 33
  : nat
```

Exercise 4.1.4 *Write a function that implements the addition of the two elements of type* **option nat**. *The returns type of this function is* **option nat**.

Exercise 4.1.5 *Write a function that implements the multiplication of the two elements of type* **option nat**. *The returns type of this function is* **option nat**.

Exercise 4.1.6 *Write a function aeval' with this signature:*

```
Fixpoint aeval' (a : AExp) (sigma : Env') : option nat :=
  ...
```

where Env' is defined as below:

```
Definition Env' := string → option nat.
```

Exercise 4.1.7 *Enrich the syntax of arithmetic expressions with subtraction, division, and modulo. Update the implementations of the functions* aeval *and* aeval' *so that these operations are evaluated as well.*

*Can you explain which version (*aeval *vs.* aeval'*) of the evaluation function is more appropriate to handle the possible errors that may occur when using subtractions and divisions (e.g., zero division, underflows)? Justify your answer.*

4.1.3 Evaluator for boolean expressions

Recall the syntax of the boolean expressions:

```
Inductive BExp :=
| btrue : BExp
| bfalse : BExp
| bnot : BExp → BExp
| band : BExp → BExp → BExp
| blessthan : AExp → AExp → BExp
| bgreaterthan : AExp → AExp → BExp.

Notation "A <' B" := (blessthan A B) (at level 80).
Notation "A >' B" := (bgreaterthan A B) (at level 80).
Infix "and'" := band (at level 82).
Notation "! A" := (bnot A) (at level 81).
```

As expected, an evaluator for boolean expressions is straightforward:

```
Fixpoint beval (b : BExp) (sigma : Env) : bool :=
  match b with
  | btrue ⇒ true
  | bfalse ⇒ false
  | bnot b' ⇒ negb (beval b' sigma)
  | band b1 b2 ⇒ andb (beval b1 sigma) (beval b2 sigma)
  | blessthan a1 a2 ⇒ Nat.ltb (aeval a1 sigma) (aeval a2 sigma)
  | bgreaterthan a1 a2 ⇒ negb (Nat.leb (aeval a1 sigma) (aeval a2 sigma))
  end.
```

The functions negb, andb, Nat.ltb and Nat.leb are builtin functions in Coq. We recommend the reader to test them separately on several examples.

Exercise 4.1.8 *Explain the semantics of the* bgreaterthan*. Can you justify why the combination of ⇒* negb *and* Nat.leb *implements the desired behaviour?*

Exercise 4.1.9 *Test the* beval *on the following examples:*

```
Compute beval ("n" <' "n") sigma1.
Compute beval ("n" >' "n") sigma1.
Compute beval ("n" <' "n" +' 1) sigma1.
```

```
Compute beval (bnot ("n" <' "n")) sigma1.
Compute beval ("n" +' 2 *' "n" <' "n") sigma1.
Compute beval ("n" +' 2 *' "n" >' "n") sigma1.
```

Exercise 4.1.10 *Write a function beval' with this signature:*

```
Fixpoint beval' (b : BExp) (sigma : Env') : option bool :=
  ...
```

where Env' is defined as below:

```
Definition Env' := string → option nat.
```

Test your function on various examples, including the cases that may cause it to return None.

4.1.4 Evaluator for IMP statements

First, recall the syntax of **IMP** statements:

```
Inductive Stmt :=
| assignment : string → AExp → Stmt
| while : BExp → Stmt → Stmt
| seq : Stmt → Stmt → Stmt.
Notation "X ::= A" := (assignment X A) (at level 85).
Notation "S1 ;; S2" := (seq S1 S2) (at level 99).
```

Although the evaluator for statements follows the same principle as the evaluators for arithmetic and boolean expressions, there is an interesting question that needs to be posed: what is returned by a statements evaluator? We have seen that the results of former evaluators are values (natural numbers or booleans). In contrast, the evaluator for statements returns the environment obtained after the execution of the statements.

It is now clear what an evaluator for statements should do: it should take a statement and an environment, and it should return a new environment. Let us discuss the effect of each statement over the environment:

- assignment *X A*: an assignment should `update` the current environment so that *X* is mapped to the value obtained by evaluating *A* (in the current environment).
- while *B S*: a while-loop should evaluate the boolean expression *B*, and
  - if *B* is evaluated to true, then execute *S* (which generates a new environment) and then execute the loop again in the new environment;
  - if *B* is evaluated to false, return the current environment because the loop does not produce any modifications to it.

- seq *S1* *S2*: when evaluating a sequence of statements *S1* *S2*, first *S1* is evaluated and produces a new environment, and then *S2* is evaluated in this new environment. In turn, *S2* produces yet another environment, which is returned as the result of evaluating the entire sequence.

Now that we have established what each **IMP** statement should do, we formally define their semantics in Coq:

```
Fixpoint eval (s : Stmt) (env : Env) : Env :=
match s with
| assignment x a ⇒ update env x (aeval a env)
| while b s ⇒ if (beval b env)
                then (eval (seq s (while b s)) env)
                else env
| seq s1 s2 ⇒ eval s2 (eval s1 env)
end.
```

But there is a problem! When we load this definition in Coq, the proof assistant complains:

```
Error: Cannot guess decreasing argument of fix.
```

This happens because our function is not guaranteed to terminate and Coq does not accept such functions. Indeed, it doesn't always terminate: if the function applies to the while true *S* program then the recursion loops forever.

This is a limitation of the Coq system, and the simplest solution here is to use an additional decreasing argument that limits the number of recursive calls. We call this new parameter *gas*, because the evaluator will run as long as there is *gas* available. Here is the patched version of the evaluator:

```
Fixpoint eval (s : Stmt) (env : Env) (gas : nat) : Env :=
  match gas with
  | 0 ⇒ env
  | S gas' ⇒ match s with
             | assignment x a ⇒ update env x (aeval a env)
             | seq s1 s2 ⇒ eval s2 (eval s1 env gas') gas'
             | while b s ⇒ if (beval b env)
                            then (eval (seq s (while b s)) env gas')
                            else env
             end
  end.
```

Note that the function terminates for sure when no more gas is available. In such cases, the current environment is returned, even if the program did not finish the normal execution. This sounds like a bad thing, doesn't it? For instance, a loop that has more steps than the value of *gas* will stop abruptly when the *gas* reaches

0. In such cases, the result of the computation is not the expected one. A quick fix is to increase the value of the *gas* parameter so that the eval function terminates for a different reason than the lack of *gas*.

We have now reached a milestone: we are able to execute **IMP** programs! A design choice that we made for **IMP** is that variables are declared on-the-fly. So, each time a variable occurs in the left-hand side of an assignment, the variable is automatically added to the environment. Initially, **IMP** programs are executed starting with an empty environment:

```
Definition emptySigma := fun (x:string) ⇒ 0.
```

This environment is updated during the execution of the program. In our simple language, updates are made only when assignments are executed.

Below is an example of an **IMP** program called pgm1 that includes only an assignment. The eval function takes the empty environment (where all variables are initialised with 0) and produces a new environment called sigma_pgm1. In this environment, the value of the program variable "n" is 1, as expected:

```
Definition pgm1 := "n" ::= 1.
Definition sigma_pgm1 := eval pgm1 emptySigma 100.
Compute sigma_pgm1 "n".
   = 1
   : nat
```

Here is another example of a program that contains a sequence of assignments:

```
Definition pgm2 := "n" ::= 1 ;; "i" ::= "n".
Definition sigma_pgm2 := eval pgm2 emptySigma 100.
Compute sigma_pgm2 "n".
   = 1
   : nat
Compute sigma_pgm2 "i".
   = 1
   : nat
```

Note that the generated environment contains the expected values for both variables “n“ and “i“.

In Section 3.6 we defined a program that computes the sum of the first “n“ numbers. We execute this program below:

```
Definition pgm_sum :=
  "n" ::= 10 ;;
  "i" ::= 1 ;;
  "sum" ::= 0 ;;
  while ( "i" <' "n" +' 1) (
```

```
        "sum" ::= "sum" +' "i" ;;
        "i" ::= "i" +' 1
  ).
```

```
Definition sum_env := (eval pgm_sum emptySigma 1000).

Compute sum_env "sum".
   = 55
   : nat
```

The result is the expected one: the sum of the first 10 numbers is indeed 55!

It is worth noting that executing the same program with insufficient gas could lead to bad results. We illustrate that on our sum program (note that the value of the *gas* parameter is 10):

```
Definition sum_env_bad := (eval pgm_sum emptySigma 10).

Compute sum_env_bad "sum".
   = 6
   : nat
```

Although having an evaluator as a function is an advantage (i.e., functions are automatically evaluated by Coq), the above situation is a major drawback of our evaluator and it could lead to serious troubles in practice. Besides the above troubles with Coq, this is not the only limitation of this approach. For instance, according to the C standard [?], the evaluation order of many expressions in C is unspecified. To have a better understanding of what this means, let us use this example:

```
 int main() {
   int x = 0;
   return (x = 1) + (x = 2); }
}
```

The operands of `+` are expressions with side-effects (that is, these expressions modify the value of `x` in the memory). The return value of this program depends on the which of the operands of `+` is evaluated first! One could argue that the return value should be 3, but in fact it can be 2 or 4[1]. Surprisingly, all these are valid return values according to the C standard! Why is this allowed in the C standard? The answer is quite simple: optimisations. Compilers are free to choose whatever evaluation strategy they want in order to increase compilation speed and execution speed. According to the design principles, a C program should be "[made] fast, even if is is not guaranteed to be portable," and implementations should always "trust the programmer" [?]. A very interesting paper about unspecified behaviour in C with tricky code samples is [?].

[1] As an exercise, we invite the reader to compile this program with different compilers and inspect the returned value of the above program.

The trouble is that our approach cannot capture such situations because our evaluator uses deterministic functions, that cannot return different values for the same input. In fact, the issue is deeper than we thought. Not only that we cannot model this C example with our approach, but it turns out that we cannot define any non-deterministic feature that programming languages might have. This limitation comes from the fact the we use functions. In the rest of these lecture notes we will present some techniques capable of handling such situations. These new methods use *relations* instead of functions.

4.1.5 Exercises

Exercise 4.1.11 *Design an imperative language that whose statements are: assignments, conditionals (i.e.,* if-then *and* if-then-else *statements), loops (* while *and* for*), and blocks of statements. The most common arithmetic (addition, multiplication, subtraction, division, modulo, shifts) and boolean expressions (negation, conjunction, disjunction) should be part of the language as well. For now, you can assume that program variables should take as values only natural numbers.*

1. *Implement an evaluator for this language.*

2. *Write a battery of tests that cover all the features of the language.*

Exercise 4.1.12 *Enrich the definition of the language that you have defined in Exercise 4.1.11 with program variables that can take booleans as values. Also, answer the next questions:*

1. *How does the syntax change?*

2. *How does the environment change?*

3. *The previous battery of tests (in Exercise 4.1.11) still execute as expected?*

4. *Add new tests to cover the new feature.*

4.2 Structural Operational Semantics

Structural Operational Semantics (SOS) is a framework proposed by Gordon Plotkin [**?**] in 1981. The author came up with a simple framework to describe the behaviour of the language constructs as inference rules. These rules specify transitions between program configurations, where configurations are typically tuples of various kinds of data structures (e.g., trees, sets, lists) that capture various components of program states (environment, memory, stacks, registers, etc.).

In the next chapters we discuss the two main SOS styles: *big-step* SOS and *small-step* SOS.

Chapter 5

Big-step SOS

The big-step SOS is probably the most natural way of defining structural operational semantics. Big-step definitions can be regarded as definitions of relations where each language construct is interpreted in an appropriate domain. Big-step SOS is also known in the literature as *natural semantics*, *relational semantics* or *evaluation semantics*.

A big-step SOS for a programming language is given as a set of inference rules. In our setting, this set of inference rules is an inductive definition, but some authors prefer to call it a *proof system*. Each inference rule uses as premises and conclusion the so-called *big-step SOS sequents*, where sequents are relations over *configurations*. We explain all these concepts below.

5.1 Configurations

Configurations are tuples of the form $\langle code, environment, ...\rangle$ that hold various information needed to execute a program. For instance, a configuration can contain the code that needs to be executed, the current environment, the program stack, the current program counter, and so on. Based on the amount of information we want to store, the configuration can take various forms. For instance, **IMP** configurations can store the following information:

- $\langle A, E\rangle$, where A has type **AExp** and E has type **Env**;
- $\langle N\rangle$, where N has type **nat**;
- $\langle B, E\rangle$, where B has type **BExp** and E has type **Env**;
- $\langle B\rangle$, where A has type **bool**;
- $\langle E\rangle$, where E has type **Env**;
- $\langle S, E\rangle$, where S has type **Stmt** and E has type **Env**;

- $\langle S \rangle$, where S has type **Stmt**.

These are all the configuration types needed to define a big-step SOS for **IMP**. You can observe that they are of different types and do not necessarily have the same number of components.

5.2 Sequents

Big-step SOS sequents are relations over configurations of the form $C \Downarrow R$, where C is a configuration, and R is a *result configuration* that is obtained after the complete evaluation of C. Result configurations are also called *irreducible configurations.* Informally, a sequent says that a configuration C evaluates/executes/transitions to a configuration R.

Here are some examples of big-step SOS sequents for **IMP**: $\langle 1, \sigma \rangle \Downarrow \langle 1 \rangle$, $\langle 1+2 \rangle \Downarrow \langle 3 \rangle$, $\langle x, \sigma \rangle \Downarrow \langle \sigma(x) \rangle$. The notation $\sigma(x)$ stands for the value of the program variable x the environment σ.

5.3 Rule schemata

The *Big-step SOS rules* have this form:

$$\frac{C_1 \Downarrow R_1 \quad C_2 \Downarrow R_2 \quad \cdots \quad C_n \Downarrow R_n}{C_0 \Downarrow R_0},$$

where $C_0, C_1, \ldots, C_n$ are configurations and $R_0, R_1, \ldots, R_n$ are result configurations. Occasionally, rules have a side condition and we typically attach them a label, that is, the rule name. Big-step SOS rules specify how sequents can be derived.

Here is an example of a big-step rule for **IMP**:

$$\frac{\langle a_1,\, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2,\, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 + a_2,\, \sigma \rangle \Downarrow \langle i_1 +_{nat} i_2 \rangle}$$

The rule says that we can derive $\langle a_1 + a_2,\, \sigma \rangle \Downarrow \langle i_1 +_{nat} i_2 \rangle$ (that is, the addition of two arithmetic expression is reduced to the usual addition over natural numbers i_1 and i_2) when $\langle a_1,\, \sigma \rangle \Downarrow \langle i_1 \rangle$ and $\langle a_2,\, \sigma \rangle \Downarrow \langle i_2 \rangle$ (that is, a_1 can be reduced to i_1 and a_2 can be reduced to i_2). The sequents $\langle a_1,\, \sigma \rangle \Downarrow \langle i_1 \rangle$ and $\langle a_2,\, \sigma \rangle \Downarrow \langle i_2 \rangle$ are derived with other big-step rules.

In the following we show all the big-step rules for IMP, and we also show how these are used to derive sequents.

5.4 Big-step SOS for IMP: arithmetic expressions

The big-step rules for **IMP**'s arithmetic expressions are shown below. The rules without premises are written without the horizontal line separator between premisses and the conclusion. Such rules are called *axioms*. BigStep-Const is an axiom saying that a constant evaluated in an environment σ is the actual constant. BigStep-Lookup evaluates a variable x in an environment σ to the value of that variable in the environment, i.e., $\sigma(x)$.

BigStep-Add and BigStep-Mul have premisses which ensure that the evaluation of the operands of the addition/multiplication is done first.

BigStep-Const: $\langle i,\, \sigma\rangle \Downarrow \langle i\rangle$

BigStep-Lookup: $\langle x,\, \sigma\rangle \Downarrow \langle \sigma(x)\rangle$ if $\sigma(x) \neq \bot$

BigStep-Add: $$\frac{\langle a_1,\, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2,\, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ +' } a_2,\, \sigma\rangle \Downarrow \langle i_1 +_{nat} i_2\rangle}$$

BigStep-Mul: $$\frac{\langle a_1,\, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2,\, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ *' } a_2,\, \sigma\rangle \Downarrow \langle i_1 *_{nat} i_2\rangle}$$

This set of inference rules forms a proof system, and thus, we can develop derivations for various sequents. Here is an example, where we assume that $\sigma(n) = 10$:

$$\dfrac{\dfrac{\cdot}{\langle 2, \sigma\rangle \Downarrow \langle 2\rangle}\ \text{BigStep-Const} \qquad \dfrac{\cdot}{\langle n, \sigma\rangle \Downarrow \langle 10\rangle}\ \text{BigStep-Lookup}}{\langle 2 \texttt{ +' } n, \sigma\rangle \Downarrow \langle 2 +_{nat} 10\rangle}\ \text{BigStep-Add}$$

This derivation is essentially a proof tree of the sequent $\langle 2 + n, \sigma\rangle \Downarrow \langle 2 +_{nat} 10\rangle$. The root of the tree is the sequent we want to prove, while the leaves are always derived using axioms. The dot "·" indicates that an axiom is applied, and thus, the proof on that branch is finished. The reason why big-step rules are rule schemas is because they establish how a rule looks like using metavariables. The following step:

$$\frac{\langle 2, \sigma\rangle \Downarrow \langle 2\rangle \quad \langle n, \sigma\rangle \Downarrow \langle 10\rangle}{\langle 2 \texttt{ +' } n, \sigma\rangle \Downarrow \langle 2 +_{nat} 10\rangle}\ \text{BigStep-Add}$$

is a an instance of the rule schema:

$$\frac{\langle a_1,\, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2,\, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ +' } a_2,\, \sigma\rangle \Downarrow \langle i_1 +_{nat} i_2\rangle}\ .$$

On the other hand, the following is not an instance of BigStep-Add:

$$\frac{\langle 2,\sigma\rangle \Downarrow \langle 2\rangle \quad \langle n,\sigma\rangle \Downarrow \langle 10\rangle}{\langle 2 \;\texttt{+'}\; n,\sigma\rangle \Downarrow \langle 2 +_{nat} 10,\sigma\rangle}\ ?$$

because $\langle 2 \;\texttt{+'}\; n,\sigma\rangle \Downarrow \langle 2 +_{nat} 10,\sigma\rangle$ is not an instance of the conclusion of the BigStep-Add (note the extra σ) or any other rule schema.

5.5 Big-step SOS rules for arithmetic expressions in Coq

The big-step of arithmetic expressions can be encoded as an inductive relation in Coq:

```
Reserved Notation "A =[ S ]=> N" (at level 60).
Inductive aeval : AExp → Env → nat → Prop :=
| const : ∀ n sigma, anum n =[ sigma ]=> n
| lookup : ∀ v sigma, avar v =[ sigma ]=> (sigma v)
| add : ∀ a1 a2 i1 i2 sigma n,
    a1 =[ sigma ]=> i1 →
    a2 =[ sigma ]=> i2 →
    n = i1 + i2 →
    a1 +' a2 =[sigma]=> n
| times : ∀ a1 a2 i1 i2 sigma n,
    a1 =[ sigma ]=> i1 →
    a2 =[ sigma ]=> i2 →
    n = i1 × i2 →
    a1 *' a2 =[sigma]=> n
where "a =[ sigma ]=> n" := (aeval a sigma n).
```

The $\Downarrow$ notation that we use for sequents is encoded using the _ =[_]=>_ notation in Coq. The inductive relation **aeval** has four cases: there is a one-to-one correspondence with the four big-step rules.

There are some implementation choices that we make: first, we explicitly use an additional variable n that holds either the addition or the product (e.g., $n = i1 \times i2$) of the evaluated operands. We use this additional variable to facilitate matching in proofs. With this trick, the domain reasoning is decoupled from the syntactical rule application. An additional goal will be generated during proofs, but this the goal is typically easy to discharge using `simpl` and `reflexivity`. Also, note the use of $\to$ instead of conjunctions: this is yet another Coq trick that helps during the proofs, since implications are handled automatically by the proof assistant.

The correspoding proof of the derivation:

$$\dfrac{\dfrac{\cdot}{\langle 2, \sigma\rangle \Downarrow \langle 2\rangle}\ \textsc{BigStep-Const} \qquad \dfrac{\cdot}{\langle n, \sigma\rangle \Downarrow \langle 10\rangle}\ \textsc{BigStep-Lookup}}{\langle 2 \ \texttt{+'}\ \ n, \sigma\rangle \Downarrow \langle 2 +_{nat} 10\rangle}\ \textsc{BigStep-Add}$$

is shown here:

```
Example ex1: 2 +' "n" =[ sigma1 ]=> 12.
Proof.
  apply add with (i1 := 2) (i2 := 10).
  - apply const.
  - apply lookup.
  - simpl. reflexivity.
Qed.
```

The reader is invited to take a look at the generated goals in Coq. First, we apply add which corresponds to BigStep-Add. This generates three new goals: two goals which are discharged using const and var, and one goal which corresponds to the domain reasoning condition.

Note that `apply add with (i1 := 2) (i2 := 10)` works only if we provide the values of i_1 and i_2. This might be inconvenient in practice, because one needs to guess or to precompute these values to write down the proof. But Coq has a tactic called `eapply` that allows us to postpone the passing of the concrete values of i_1 and i_2. This is how the same derivation can be written using the new tactic:

```
Example ex1': 2 +' "n" =[ sigma1 ]=> 12.
Proof.
  eapply add.
  - apply const.
  - apply lookup.
  - unfold sigma1. simpl. reflexivity.
Qed.
```

The only trouble here is that the domain reasoning requires an extra unfolding before simplication (e.g., `unfold sigma1`).

Exercise 5.5.1 *Replace the sequence of tactics: "`unfold sigma1. simpl. reflexivity.`" from example `ex1'` with "`eauto.`" Does it solve our goal?*

Read the documentation of `eauto` available here:`https://coq.inria.fr/refman/proofs/automatic-tactics/auto.html#coq:tacn.eauto`.

Exercise 5.5.2 *Write down a sequent that reduces an* **IMP** *arithmethic expression to a value, given a concrete environment. The arithmetic expression should include at least two different variables, two constants, an addition and a multiplication. Then, write on a piece of paper the proof derivation of that sequent (this way you will find out whether it is derivable or not!) and then do the Coq proof (based on the paper proof).*

5.6 Big-step SOS for IMP: boolean expressions

The big-step SOS rules for boolean expressions are shown below:

$$\textsc{BigStep-True}: \quad \langle \mathsf{btrue}, \sigma \rangle \Downarrow \langle \mathit{true} \rangle$$

$$\textsc{BigStep-False}: \quad \langle \mathsf{bfalse}, \sigma \rangle \Downarrow \langle \mathit{false} \rangle$$

$$\textsc{BigStep-NotTrue}: \quad \frac{\langle b,\, \sigma \rangle \Downarrow \langle \mathit{true} \rangle}{\langle \mathtt{!}\, b,\, \sigma \rangle \Downarrow \langle \mathit{false} \rangle}$$

$$\textsc{BigStep-NotFalse}: \quad \frac{\langle b,\, \sigma \rangle \Downarrow \langle \mathit{false} \rangle}{\langle \mathtt{!}\, b,\, \sigma \rangle \Downarrow \langle \mathit{true} \rangle}$$

$$\textsc{BigStep-AndTrue}: \quad \frac{\langle b_1,\, \sigma \rangle \Downarrow \langle \mathit{true} \rangle \quad \langle b_2,\, \sigma \rangle \Downarrow \langle b \rangle}{\langle b_1 \;\mathsf{and'}\; b_2,\, \sigma \rangle \Downarrow \langle b \rangle}$$

$$\textsc{BigStep-AndFalse}: \quad \frac{\langle b_1,\, \sigma \rangle \Downarrow \langle \mathit{false} \rangle}{\langle b_1 \;\mathsf{and'}\; b_2,\, \sigma \rangle \Downarrow \langle \mathit{false} \rangle}$$

$$\textsc{BigStep-Lt}: \quad \frac{\langle a_1,\, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2,\, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \;\mathtt{<'}\; a_2, \sigma \rangle \Downarrow \langle i_1 <_{nat} i_2 \rangle}$$

$$\textsc{BigStep-Gt}: \quad \frac{\langle a_1,\, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2,\, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \;\mathtt{>'}\; a_2, \sigma \rangle \Downarrow \langle i_1 >_{nat} i_2 \rangle}$$

The constants btrue and bfalse are evaluated to their corresponding boolean values *true* and *false*. The evaluation of the negation of a boolean expression requires the big-step (complete) evaluation of the expression under the negation: if the result is *true* then the entire boolean expression is evaluated to *false*, and viceversa. The semantics for and' is short-circuited: if the first operand is evaluated to *true*, then the result is the evaluation of the second operand; if the first operand is evaluated to *false*, then the entire expression is evaluated to *false*. This is an optimisation that avoids the evaluation of the other operand of the conjunction when the first one is evaluated to *false*.

The comparison operators use the big-step semantics of the arithmetic expressions in order to obtain the result for the boolean expression. This is the place where we can observe the composition of big-step rules: the semantics of arithmetic expressions is plugged-in the semantics of boolean expressions.

We show now a derivation tree for the sequent $\langle 2 \;\mathtt{+'}\; n \;\mathtt{<'}\; 10, \sigma \rangle \Downarrow \langle (2 +_{nat} 10) <_{nat} 10 \rangle$ (which is the same as $\langle 2 \;\mathtt{+'}\; n \;\mathtt{<'}\; 10, \sigma \rangle \Downarrow \langle \mathit{false} \rangle$), where $\sigma(n) = 10$:

$$\dfrac{\dfrac{\dfrac{\cdot}{\langle 2,\sigma\rangle \Downarrow \langle 2\rangle}\ \textsc{Const} \quad \dfrac{\cdot}{\langle n,\sigma\rangle \Downarrow \langle 10\rangle}\ \textsc{Lookup}}{\langle 2\ \texttt{+'}\ n,\sigma\rangle \Downarrow \langle 2 +_{nat} 10\rangle}\ \textsc{Add} \quad \dfrac{\cdot}{\langle 10,\sigma\rangle \Downarrow \langle 10\rangle}\ \textsc{Const}}{\langle 2\ \texttt{+'}\ n\ \texttt{<'}\ 10,\sigma_1\rangle \Downarrow \langle (2 +_{nat} 10) <_{nat} 10\rangle}\ \textsc{Lt}$$

Because the derivation tree is a bit bigger, we used abbreviations for the rule names (that is, CONST instead of BIGSTEP-CONST, LOOKUP instead of BIGSTEP-LOOKUP, ADD instead of BIGSTEP-ADD, LT instead of BIGSTEP-LT).

5.7 Big-step SOS for boolean expressions in Coq

In Coq, the big-step SOS for boolean expressions is implemented as shown below:

```
Reserved Notation "B ={ State }=> B'" (at level 91).

Inductive beval : BExp → Env → bool → Prop :=
| bigstep_true : ∀ state,
    btrue ={ state }=> true
| bigstep_false : ∀ state,
    bfalse ={ state }=> false
| bigstep_lt: ∀ a1 a2 i1 i2 state b,
    a1 =[state]=> i1 →
    a2 =[state]=> i2 →
    b = Nat.ltb i1 i2 →
    (a1 <' a2) ={ state }=> b
| bigstep_gt: ∀ a1 a2 i1 i2 state b,
    a1 =[state]=> i1 →
    a2 =[state]=> i2 →
    b = negb (Nat.leb i1 i2) →
    (a1 >' a2) ={ state }=> b
| bigstep_nottrue : ∀ b state,
    b ={ state }=> true →
    (bnot b) ={ state }=> false
| bigstep_notfalse : ∀ b state,
    b ={ state }=> false →
    (bnot b) ={ state }=> true
| bigstep_andtrue : ∀ b1 b2 state b,
    b1 ={state}=> true →
    b2 ={state}=> b →
    (band b1 b2) ={ state }=> b
| bigstep_andfalse : ∀ b1 b2 state,
```

```
    b1 ={state}=> false →
    (band b1 b2) ={state}=> false
where "B ={ State }=> B'" := (beval B State B').
```

As in the case of arithmetic expressions, we introduce here a notation for sequents reducing boolean expressions. For bigstep_lt and bigstep_gt we use an additional *b* to keep the result of the comparisons. Here is our derivation example proved in Coq:

```
Example ex2 :
  2 +' "n" <' 10 ={ sigma1 }=> false.
Proof.
  eapply bigstep_lt.
  - eapply add.
    + eapply const.
    + eapply lookup.
    + eauto.
  - eapply const.
  - trivial.
Qed.
```

Again, the Coq proof uses the same derivation rules plus some tactics that handle the domain specific proofs. We also use the `trivial` tactic which implements a non-recursive Prolog-like resolution to solve the current goal.

Exercise 5.7.1 *Append to the syntax and semantics of the boolean expressions support for disjunctions of boolean expressions and the missing comparators for arithmetic expressions: equal to ("=="), not equal to ("!="), less or equal than ("<="), greater or equal then (">=").*

For each new language construct write at least one test and the corresponding derivation. Implement all these in Coq as well.

5.8 Big-step SOS for IMP: statements

We are now ready to define the big-step SOS rules for the statements of **IMP**:

BigStep-Assign:
$$\frac{\langle a,\, \sigma\rangle \Downarrow \langle i\rangle}{\langle x \;\; ::= \;\; a,\, \sigma\rangle \Downarrow \langle \sigma[i/x]\rangle} \quad \mathit{if}\ \sigma(x) \neq \bot$$

BigStep-Seq:
$$\frac{\langle s_1,\, \sigma\rangle \Downarrow \langle \sigma_1\rangle \qquad \langle s_2,\, \sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle s_1 \;;;\; s_2,\, \sigma\rangle \Downarrow \langle \sigma_2\rangle}$$

BigStep-WhileFalse:
$$\frac{\langle b,\, \sigma\rangle \Downarrow \langle \mathit{false}\rangle}{\langle \mathsf{while}\, b\, s,\, \sigma\rangle \Downarrow \langle \sigma\rangle}$$

BigStep-WhileTrue:
$$\frac{\langle b,\, \sigma\rangle \Downarrow \langle \mathit{true}\rangle \qquad \langle s \;\; ;; \;\; \mathsf{while}\, b\, s,\, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \mathsf{while}\, b\, s,\, \sigma\rangle \Downarrow \langle \sigma'\rangle}$$

The big-step SOS rules for statements are more complex than the rules for expressions. The BigStep-Assign captures the essence of an assignment statement: the value of the program variable x in the environment σ is changed to the value i obtained by evaluating a in σ. The updated environment is denoted by $\sigma[i/x]$. In conclusion, the configuration $\langle x \;\; ::= \;\; a;,\, \sigma\rangle$ transits to a configuration $\langle \sigma[i/x]\rangle$, where the environment is updated accordingly.

The rule for sequence of statements BigStep-Seq is quite intuitive: the evaluation of the first statement s_1 in the environment σ produces a new environment σ_1; then, the evaluation of the second statement s_2 in σ_1 produces σ_2; in conclusion, the evaluation of the entire sequence of statements produces a configuration that contains σ_2.

The loop has two rules: one for the case when the condition is false, which leaves the environment untouched, and one rule for the case when the condition is true. In the latter, the body of the loop is executed once and then the loop is executed again. This sequence produces a new environment σ', which forms the result configuration for the entire loop. This technique for defining the semantics of loops is called loop unrolling.

Example We show here a complete derivation of the sequent

$$\langle i::=0;;\mathsf{while}\ i\texttt{<'}1\ (i::=i\texttt{+'}1), \sigma\rangle \Downarrow \langle(\sigma[0/i])[1/i]\rangle.$$

This a program where the loop should execute once and then stop. We use big-step rules from all categories (arithmetic expressions, boolean expressions, statements) to build a derivation tree for this sequent. Because the size of the derivation tree is bigger, we use uppercase letters to denote parts of it. The trees corresponding to each letter are explicitly provided, so that the entire proof can be reconstructed. To save some space, we drop the BigStep- prefix for the rule names (e.g., we use Seq instead of BigStep-Seq, and so on).

$$\dfrac{\dfrac{\cdot}{\langle i::=0, \sigma\rangle \Downarrow \langle \sigma[0/i]\rangle}\ \textsc{Assign} \qquad A}{\langle i::=0;;\mathsf{while}\ i\texttt{<'}1\ (i::=i\texttt{+'}1), \sigma\rangle \Downarrow \langle(\sigma[0/i])[1/i]\rangle}\ \textsc{Seq}$$

The derivation tree corresponding to A (note that the loop is evaluated in the environment $\sigma[0/i]$) is shown here:

$$\dfrac{\dfrac{\dfrac{\cdot}{\langle i, \sigma[0/i]\rangle \Downarrow \langle 0\rangle}\textsc{Lookup} \qquad \dfrac{\cdot}{\langle 1, \sigma[0/i]\rangle \Downarrow \langle 1\rangle}\textsc{Const}}{\langle i <' 1, \sigma[0/i]\rangle \Downarrow \langle \mathit{true}\rangle}\textsc{Lt} \qquad B}{\langle \mathsf{while}\ i <' 1\ (i ::= i +' 1), \sigma[0/i]\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle}\textsc{WhileTrue}$$

Here is the derivation tree corresponding to B:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\langle i, \sigma[0/i]\rangle \Downarrow \langle 0\rangle}\textsc{Lookup} \qquad \dfrac{\cdot}{\langle 1, \sigma[0/i]\rangle \Downarrow \langle 1\rangle}\textsc{Const}}{\langle i +' 1\rangle \Downarrow \langle 1\rangle}\textsc{Add}}{\langle i ::= i +' 1\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle}\textsc{Assign} \qquad C}{\langle i ::= i +' 1 ;; \mathsf{while}\ i <' 2\ (i ::= i +' 1), \sigma[0/i]\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle}\textsc{Seq}$$

Finally, the derivation tree corresponding to C is:

$$\dfrac{\dfrac{\dfrac{\cdot}{\langle i, (\sigma[0/i])[1/i]\rangle \Downarrow \langle 1\rangle}\textsc{Lookup} \qquad \dfrac{\cdot}{\langle 1, (\sigma[0/i])[1/i]\rangle \Downarrow \langle 1\rangle}\textsc{Const}}{\langle i <' 1, (\sigma[0/i])[1/i]\rangle \Downarrow \langle \mathit{false}\rangle}\textsc{Lt}}{\langle \mathsf{while}\ i <' 1\ (i ::= i +' 1), (\sigma[0/i])[1/i]\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle}\textsc{WhileFalse}$$

Obviously, the derivation tree may be a little hard to read in this form, but it is even harder to read it if we display it entirely. Just for fun (there is no need to read it) take a look at the assembled derivation tree:

$$\dfrac{\dfrac{\cdot}{\langle i ::= 0, \sigma\rangle \Downarrow \langle \sigma[0/i]\rangle} \qquad \dfrac{\dfrac{\dfrac{\cdot}{\langle i, \sigma[0/i]\rangle \Downarrow \langle 0\rangle} \quad \dfrac{\cdot}{\langle 1, \sigma[0/i]\rangle \Downarrow \langle 1\rangle}}{\langle i <' 1, \sigma[0/i]\rangle \Downarrow \langle \mathit{true}\rangle} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\langle i, \sigma[0/i]\rangle \Downarrow \langle 0\rangle} \quad \dfrac{\cdot}{\langle 1, \sigma[0/i]\rangle \Downarrow \langle 1\rangle}}{\langle i +' 1\rangle \Downarrow \langle 1\rangle}}{\langle i ::= i +' 1\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle} \quad \dfrac{\dfrac{\dfrac{\cdot}{\langle i, (\sigma[0/i])[1/i]\rangle \Downarrow \langle 1\rangle} \quad \dfrac{\cdot}{\langle 1, (\sigma[0/i])[1/i]\rangle \Downarrow \langle 1\rangle}}{\langle i <' 1, (\sigma[0/i])[1/i]\rangle \Downarrow \langle \mathit{false}\rangle}}{\langle \mathsf{while}\ i <' 1\ (i ::= i +' 1), (\sigma[0/i])[1/i]\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle}}{\langle i ::= i +' 1 ;; \mathsf{while}\ i <' 2\ (i ::= i +' 1), \sigma[0/i]\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle}}{\langle \mathsf{while}\ i <' 1\ (i ::= i +' 1), \sigma[0/i]\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle}}{\langle i ::= 0 ;; \mathsf{while}\ i <' 1\ (i ::= i +' 1), \sigma\rangle \Downarrow \langle (\sigma[0/i])[1/i]\rangle}$$

As we can observe in our example, writing derivations by hand is cumbersome. Fortunately, Coq is really useful when it comes about writing proofs.

5.9 Big-step SOS rules for statements in Coq

Here is the big-step semantics of **IMP** statements in Coq:

```
Reserved Notation "Stmt -{ Env }-> Env' " (at level 99).
Inductive eval : Stmt → Env → Env → Prop :=
| bigstep_assign: ∀ var a state state' i,
    a =[state]=> i →
    state' = update state var i →
    (var ::= a) -{state}-> state'
| bigstep_seq : ∀ s1 s2 state1 state2 state,
    s1 -{state}-> state1 →
    s2 -{state1}-> state2 →
    (s1 ;; s2) -{state}-> state2
| bigstep_whilefalse: ∀ state b s,
    b ={state}=> false →
    (while b s) -{state}-> state
| bigstep_whiletrue: ∀ state state' b s,
    b ={state}=> true →
    (s ;; (while b s)) -{state}-> state' →
    (while b s) -{state}-> state'
where "Stmt -{ Env }-> Env' " := (eval Stmt Env Env').
```

The evaluation of a statement in a given environment produces another environment. Note that only the assignments modify the environments and they are the basic building bricks for the other statements as well.

The Coq proof of our previous example in Coq is below:

```
Example ex3:
  "i" ::= 0 ;;
  while ("i" <' 1) (
      "i" ::= "i" +' 1
  )
  -{ sigma1 }->
  update (update sigma1 "i" 0) "i" 1.
Proof.
  eapply bigstep_seq.
  - eapply bigstep_assign.
    + eapply const.
    + eauto.
  - eapply bigstep_whiletrue.
    -- eapply bigstep_lt.
       ++ eapply lookup.
       ++ eapply const.
       ++ eauto.
    -- eapply bigstep_seq.
       --- eapply bigstep_assign.
           eapply add.
```

```
        eapply lookup.
        eapply const.
        eauto.
        eauto.
    — eapply bigstep_whilefalse.
        eapply bigstep_lt.
        eapply lookup.
        eapply const.
        auto.
Qed.
```

The above Coq proof is as complex as the corresponding derivation tree. However, there are some big advantages when using a proof assistant to encode a big-step SOS semantics. First, the formulation of the semantics is straightforward. Second, the Coq support for interactive proofs allows us to focus on smaller pieces of our proofs, and it takes care of combining them to get the final proof. Finally, it does *check* the proof for us. The latter feature is really important since it ensures that our proof does not contain mistakes or forgotten corner cases. In our paper proof above, we do not explicitly show the basic operations over naturals or boolean comparisons, but in Coq these are not overlooked. There are other language frameworks (e.g., K, Maude, CafeObj, Rascal, Spoofax, Redex, PLanCompS) that are even better when it comes about automation (i.e., program execution using the semantics). However, at the time of writing these notes, these tools do not have tool support for proving and certification for proofs.

5.10 Proof search

In terms of automation, Coq has some features that can improve the user experience by making the proofs (as the one above) shorter via proof search. The `auto` and `eauto` tactics implement Prolog-like resolution procedures. They try other tactics like `assumption` and `intros` and use the introduced hypothesis as hints. Also, `auto` and `eauto` search for the tactics to be applied and tries them by prioritising the ones with a lower cost. The process is recursive and it does have a default search depth of value 5.

The nice thing about these tactics is that we can customise them. Using the `with` clause we can specify a hint database that is custom to our proof needs. For our example, this is done easily by creating a hint database:

```
Create HintDb mydb.
Hint Constructors aeval : mydb.
Hint Constructors beval : mydb.
Hint Constructors eval : mydb.
Hint Unfold update : mydb.
```

The first command (*Create HintDb*) creates a custom hint database called *mydb*.

The `Hint Constructors` command adds all the constructors of the given inductive definition as hints in the database. `Hint Unfold` adds the tactic `unfold` for the given argument. Just by adding these to our database we can use `eauto` to solve our example immediately:

```
Example ex3':
  "i" ::= 0 ;;
  while ("i" <' 1) (
      "i" ::= "i" +' 1
  )
  -{ sigma1 }->
  update (update sigma1 "i" 0) "i" 1.
Proof.
  eauto 10 with mydb.
Qed.
```

The depth of the proof search is 10 in this case. However, if we increase the size of the loop, a bigger value is needed, and more time is consumed until Coq finds the proof.

Although it is useful, the proof search capability of Coq is quite limited and sometimes simply writing the proof takes less than leaving your computer to search for one. From this point of view, the Isabelle/HOL theorem prover, provides a more powerful tactic called `sledgehammer`.

5.11 Exercises

Exercise 5.11.1 *Prove that the evaluation of arithmetic expressions is deterministic:*

```
Lemma aeval_is_deterministic:
  ∀ aexp sigma n n',
    aexp =[ sigma ]=> n →
    aexp =[ sigma ]=> n' →
    n = n'.
Proof.
   ...
Qed.
```

Exercise 5.11.2 *Given the function for evaluating the arithmetic expressions of* **IMP** *(aeval_fun), prove the following lemmas:*

```
Fixpoint aeval_fun (a : AExp) (sigma : Env) : nat :=
  match a with
  | avar x ⇒ (sigma x)
  | anum n ⇒ n
  | aplus a1 a2 ⇒ (aeval_fun a1 sigma) + (aeval_fun a2 sigma)
```

```
  | amul a1 a2 ⇒ (aeval_fun a1 sigma) × (aeval_fun a2 sigma)
  end.

Lemma equiv :
  ∀ aexp sigma n,
    n = aeval_fun aexp sigma →
    aexp =[ sigma ]=> n.
Proof.
   ...
Qed.

Lemma equiv' :
  ∀ aexp sigma n,
    aexp =[ sigma ]=> n →
    n = aeval_fun aexp sigma.
Proof.
   ...
Qed.
```

5.12 Improving IMP

So far, **IMP** does not include some very common arithmetic expressions, like subtraction, division or modulo. Also, boolean expressions like disjunction, less or equal than ($\leq$), and greater or equal than ($\geq$) are not yet part of the language. Besides the syntax additions, the big-step SOS rules for these well-known constructs are straightforward:

$$\text{BIGSTEP-SUB:} \quad \frac{\langle a_1,\, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2,\, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \;\texttt{-'}\; a_2,\, \sigma\rangle \Downarrow \langle i_1 -_{nat} i_2\rangle} \quad \text{if } i_1 \geq i_2$$

$$\text{BIGSTEP-DIV:} \quad \frac{\langle a_1,\, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2,\, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \;\texttt{/'}\; a_2,\, \sigma\rangle \Downarrow \langle i_1 /_{nat} i_2\rangle} \quad \text{if } i_2 \neq 0$$

$$\text{BIGSTEP-MOD:} \quad \frac{\langle a_1,\, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2,\, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \;\texttt{\%'}\; a_2,\, \sigma\rangle \Downarrow \langle i_1 \%_{nat} i_2\rangle} \quad \text{if } i_2 \neq 0$$

$$\text{BIGSTEP-ORFALSE:} \quad \frac{\langle b_1,\, \sigma\rangle \Downarrow \langle \mathit{false}\rangle \quad \langle b_2,\, \sigma\rangle \Downarrow \langle b\rangle}{\langle b_1 \;\mathsf{or'}\; b_2,\, \sigma\rangle \Downarrow \langle b\rangle}$$

$$\text{BIGSTEP-ORTRUE:} \quad \frac{\langle b_1,\, \sigma\rangle \Downarrow \langle \mathit{true}\rangle}{\langle b_1 \;\mathsf{or'}\; b_2,\, \sigma\rangle \Downarrow \langle \mathit{true}\rangle}$$

$$\text{BIGSTEP-LEQ:} \quad \frac{\langle a_1,\, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2,\, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \;\texttt{<='}\; a_2, \sigma\rangle \Downarrow \langle i_1 \leq_{nat} i_2\rangle}$$

$$\text{BIGSTEP-GEQ:} \quad \frac{\langle a_1,\, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2,\, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \;\texttt{>='}\; a_2, \sigma\rangle \Downarrow \langle i_1 \geq_{nat} i_2\rangle}$$

Exercise 5.12.1 *Implement the above rules in Coq and write some relevant examples. Use Coq to derive sequents that include the new expressions.*

Besides expressions, some useful statements are missing as well. For instance, we have sequences of statements, but we do not have a way to specify an empty sequence of statements. This can be useful, for example, to write a loop with no body. Also, we do not have conditional statements at all: there is no *if-then* or *if-then-else* statement.

To improve our language, we extend its syntax with a statement skip that denotes the empty sequence of statements, and we also add ite that denotes the well-known *if-then-else* statement. The simple *if-then* statement, which is just a particular case of the *if-then-else* statement is denoted in our syntax by if. The big-step SOS rules of these new constructs are shown below:

BigStep-Skip: $\langle \mathsf{skip},\, \sigma\rangle \Downarrow \langle\sigma\rangle$

BigStep-IteTrue: $$\frac{\langle b,\, \sigma\rangle \Downarrow \langle \mathit{true}\rangle \qquad \langle s_1, \sigma\rangle \Downarrow \langle\sigma_1\rangle}{\langle \mathsf{ite}\, b\, s_1\, s_2,\, \sigma\rangle \Downarrow \langle\sigma_1\rangle}$$

BigStep-IteFalse: $$\frac{\langle b,\, \sigma\rangle \Downarrow \langle \mathit{false}\rangle \qquad \langle s_2, \sigma\rangle \Downarrow \langle\sigma_2\rangle}{\langle \mathsf{ite}\, b\, s_1\, s_2,\, \sigma\rangle \Downarrow \langle\sigma_2\rangle}$$

BigStep-IfTrue: $$\frac{\langle b,\, \sigma\rangle \Downarrow \langle \mathit{true}\rangle \qquad \langle s, \sigma\rangle \Downarrow \langle\sigma'\rangle}{\langle \mathsf{if}\, b\, s,\, \sigma\rangle \Downarrow \langle\sigma'\rangle}$$

BigStep-IfFalse: $$\frac{\langle b,\, \sigma\rangle \Downarrow \langle \mathit{false}\rangle}{\langle \mathsf{if}\, b\, s,\, \sigma\rangle \Downarrow \langle\sigma\rangle}$$

Exercise 5.12.2 *Implement the above rules in Coq and write some relevant examples. Use Coq to derive sequents that include the new statements.*

Chapter 6

Small-step SOS

Small-step SOS is a variant of SOS that captures the notion of one computational step. This is why Small-step SOS is also called *one-step operational semantics* or *reduction/transition semantics*. The transitions between small-step configurations are denoted by a simple arrows "$\rightarrow$", which is more intuitive: it captures only *one* computation step at a time, while the big-step transition denoted by $\Downarrow$ captures *all* computation steps in one transition. Therefore, we need to compose multiple small steps to simulate one big step. This is done by computing a transitive closure $\rightarrow^*$ of the small step relation $\rightarrow$.

Small-step SOS is well suited to capture non-deterministic behaviour in programs. For instance, programming languages that allow concurent programs can be easily formalised using small-step SOS.

The *small-step SOS sequents* are binary relations over configurations: $C \rightarrow C'$. The meaning of $C \rightarrow C'$ is the following: the configuration C' is obtained from C after *one* step of computation. *Small-step SOS rules* have the following general form:

$$\frac{C_1 \rightarrow C_1' \quad C_2 \rightarrow C_2' \quad \cdots \quad C_n \rightarrow C_n'}{C_0 \rightarrow C_0'} \; cond.$$

The small-step SOS rules are more verbose than the big-step SOS rules. Also, we do not have to define result configurations explicitly: here, result configurations are the ones that cannot be reduced anymore w.r.t the one step relation.

The small-step SOS configurations for **IMP** are a subset of the big-step SOS configurations:

- $\langle A, E \rangle$, where A is of type **AExp** and E is of type **Env**;
- $\langle B, E \rangle$, where B is of type **BExp** and E is of type **Env**;
- $\langle S, E \rangle$, where S is of type **Stmt** and E is of type **Env**.

6.1 Small-step SOS for IMP: arithmetic expressions

As in the case of big-step SOS we start with arithmetic expressions. There are some noticeable differences w.r.t. big-step SOS:

- The lookup rule that describes a one step transition from $\langle x, \sigma\rangle$ to $\langle \sigma(x), \sigma\rangle$ instead of $\langle \sigma(x)\rangle$ as in big-step SOS;
- There is no rule for evaluating constants;
- We have more rules for evaluating addition/multiplication of arithmetic expressions. The reason is that both operands of an addition/multiplication can be evaluated using the one step relation $\langle a, \sigma\rangle \to \langle a', \sigma\rangle$. Therefore, one can evaluate either the first argument or the second argument, and this is why we have two rules - one for each operand. Also, when both operands are evaluated we use an additional rule to obtain the result.

Here is the small-step SOS for **IMP** arithmetic expressions:

SmallStep-Lookup: $\langle x,\, \sigma\rangle \to \langle \sigma(x), \sigma\rangle$ if $\sigma(x) \neq \bot$

SmallStep-Add1: $\dfrac{\langle a_1,\, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \texttt{ +' } a_2,\, \sigma\rangle \to \langle a_1' \texttt{ +' } a_2, \sigma\rangle}$

SmallStep-Add2: $\dfrac{\langle a_2,\, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \texttt{ +' } a_2,\, \sigma\rangle \to \langle a_1 \texttt{ +' } a_2', \sigma\rangle}$

SmallStep-Add: $\langle i_1 \texttt{ +' } i_2, \sigma\rangle \to \langle i_1 +_{nat} i_2, \sigma\rangle$

SmallStep-Mul1: $\dfrac{\langle a_1,\, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \texttt{ *' } a_2,\, \sigma\rangle \to \langle a_1' \texttt{ *' } a_2, \sigma\rangle}$

SmallStep-Mul2: $\dfrac{\langle a_2,\, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \texttt{ *' } a_2,\, \sigma\rangle \to \langle a_1 \texttt{ *' } a_2', \sigma\rangle}$

SmallStep-Mul: $\langle i_1 \texttt{ *' } i_2, \sigma\rangle \to \langle i_1 *_{nat} i_2, \sigma\rangle$

Note that the rules SmallStep-Add1 and SmallStep-Add2 can be applied non-deterministically. Therefore, the evaluation strategy for addition is non-deterministic - which is not possible via a big-step SOS. The same applies to multiplication.

Let us consider $\sigma(n) = 10$. Here is a derivation for sequent $\langle 2 \texttt{ +' } n, \sigma\rangle \to \langle 2 + 10, \sigma\rangle$:

$$\dfrac{\dfrac{\cdot}{\langle n, \sigma \rangle \to \langle 10, \sigma \rangle}\ \textsc{SmallStep-Lookup}}{\langle 2\ \texttt{+'}\ \ n, \sigma \rangle \to \langle 2 +_{nat} 10, \sigma \rangle}\ \textsc{SmallStep-Add2}$$

It is worth noting that SMALLSTEP-ADD2 is the only rule that can be applied in this context. SMALLSTEP-ADD1 cannot be applied here because it is impossible to derive its hypothesis.

Exercise 6.1.1 *Is it possible to derive the sequent* $\langle 2\ \texttt{+'}\ n, \sigma \rangle \to \langle 12, \sigma \rangle$[1] *using the small-step SOS rules from above?*

Since small-step SOS rules specify only one computation at a time, we need a way to compose these steps. The relation $\to^*$ is the reflexive and transitive closure of $\to$ and it is defined below:

$$\dfrac{\cdot}{a \to^* a}\ \textsc{Refl} \qquad \dfrac{a_1 \to a_2 \qquad a_2 \to^* a_3}{a_1 \to^* a_3}\ \textsc{Tran}$$

Using $\to^*$ we are now able to derive the sequent $\langle 2\ \texttt{+'}\ \ n, \sigma \rangle \to^* \langle 12, \sigma \rangle$ (again, we drop the SMALLSTEP- prefix from the rule names to save some space):

$$\dfrac{\dfrac{\dfrac{\cdot}{\langle n, \sigma \rangle \to \langle 10, \sigma \rangle}\ \textsc{Lookup}}{\langle 2\ \texttt{+'}\ \ n, \sigma \rangle \to \langle 2\ \texttt{+'}\ \ 10, \sigma \rangle}\ \textsc{Add2} \qquad \dfrac{\dfrac{\cdot}{\langle 2\ \texttt{+'}\ \ 10, \sigma \rangle \to \langle 12, \sigma \rangle}\ \textsc{Add} \qquad \dfrac{\cdot}{\langle 12, \sigma \rangle \to^* \langle 12, \sigma \rangle}\ \textsc{Refl}}{\langle 2\ \texttt{+'}\ \ 10, \sigma \rangle \to^* \langle 12, \sigma \rangle}\ \textsc{Tran}}{\langle 2\ \texttt{+'}\ \ n, \sigma \rangle \to^* \langle 12, \sigma \rangle}\ \textsc{Tran}$$

Exercise 6.1.2 *Write a derivation for the sequent* $\langle n\ \texttt{+'}\ n, \sigma \rangle \to^* \langle 20, \sigma \rangle$.

Exercise 6.1.3 *Write down the small-step SOS rules for subtraction, division and modulo operations. Write some relevant examples in order to test each small-step rule.*

6.2 Small-step SOS for arithmetic expressions in Coq

The Coq implementation of the Small-step SOS rules for arithmetic expressions is given below:

```
Reserved Notation "<| A , S |> => <| A' , S' |>" (at level 60).

Inductive aeval_small_step : AExp → Env → AExp → Env → Prop :=
| lookup : ∀ v st, <| avar v, st |> ⇒ <| st v , st |>
```

[1]We strongly encourage the reader to explain why it is impossible to derive this sequent. Hint: it is strongly related to the fact that small-step SOS rules capture only one computational step, while here we need more computational steps.

```
| add_1 : ∀ a1 a2 a1' st,
    <| a1 , st |> ⇒ <| a1' , st |> →
    <| a1 +' a2 , st |> ⇒ <| a1' +' a2 , st |>
| add_2 : ∀ a1 a2 a2' st,
    <| a2 , st |> ⇒ <| a2' , st |> →
    <| a1 +' a2 , st |> ⇒ <| a1 +' a2' , st |>
| add : ∀ i1 i2 st n,
    n = anum (i1 + i2) →
    <| anum i1 +' anum i2 , st |> ⇒ <| n , st |>
| mul_1 : ∀ a1 a2 a1' st,
    <| a1 , st |> ⇒ <| a1' , st |> →
    <| a1 *' a2 , st |> ⇒ <| a1' *' a2 , st |>
| mul_2 : ∀ a1 a2 a2' st,
    <| a2 , st |> ⇒ <| a2' , st |> →
    <| a1 *' a2 , st |> ⇒ <| a1 *' a2' , st |>
| mul : ∀ i1 i2 st n,
    n = anum (i1 × i2) →
    <| anum i1 *' anum i2 , st |> ⇒ <| n , st |>
where "<| A , S |> => <| A' , S' |>" := (aeval_small_step A S A' S').
```

With these rules we can simulate the next derivation in Coq:

```
Example e1 :
  <| 2 +' "n" , sigma1 |> ⇒ <| 2 +' 10, sigma1 |>.
Proof.
  eapply add_2.
  eapply lookup.
Qed.
```

The reflexive and transitive closure of → is encoded in Coq as below:

```
Reserved Notation "<| A , S |> =>* <| A' , S' |>" (at level 60).
Inductive aeval_clos : AExp → Env → AExp → Env → Prop :=
| a_refl : ∀ a st, <| a , st |> =>* <| a , st |>
| a_trans : ∀ a1 a2 a3 st,
            (<| a1 , st |> ⇒ <| a2 , st |>) →
            <| a2 , st |> =>* <| a3 , st |> →
            <| a1 , st |> =>* <| a3 , st |>
where "<| A , S |> =>* <| A' , S' |>" := (aeval_clos A S A' S').
```

Now, the Coq derivation for $\langle 2 \texttt{ +' } n, \sigma\rangle \rightarrow^* \langle 12, \sigma\rangle$ looks like this:

```
Example e2 :
  <| 2 +' "n" , sigma1 |> =>* <| anum 12 , sigma1 |> .
Proof.
 - eapply a_trans with (a2 := 2 +' 10).
```

```
  + eapply add_2.
    × eapply lookup.
  + eapply a_trans.
    × eapply add. eauto.
    × simpl. eapply a_refl.
Qed.
```

Exercise 6.2.1 *Continue Exercise 6.1.2 and write the derivation of the given sequent in Coq.*

Exercise 6.2.2 *Implement the solution for Exercise 6.1.3 in Coq, including the examples.*

6.3 Small-step SOS for IMP: boolean expressions

As in the case of arithmetic expressions, the small-step SOS rules for boolean expressions are different from the big-step SOS rules. Here are the differences:

- there are no rules for evaluating the boolean constants;
- there are three rules for negation: the first one performs a one step evaluation of the negated boolean expression, the other two rules correspond to the evaluation of the base cases;
- there are three rules for conjunction: one which evaluates the first argument, one which evaluates the conjunction when the first argument is false (remember the shortcircuited semantics of the conjunction), and the last rule is for evaluating the conjunction when the first argument is true;
- the rules for comparator operators look like the small-step rules for addition/multiplication: each argument can be evaluated by a separate rule and there is one rule which compares two concrete values.

The small-step SOS rules are shown here:

SMALLSTEP-NOT: $$\frac{\langle b,\, \sigma\rangle \rightarrow \langle b', \sigma\rangle}{\langle \texttt{!}\, b,\, \sigma\rangle \rightarrow \langle \texttt{!}\, b', \sigma\rangle}$$

SMALLSTEP-NOTTRUE: $$\langle \texttt{!}\, \mathit{true},\, \sigma\rangle \rightarrow \langle \mathit{false}, \sigma\rangle$$

SMALLSTEP-NOTFALSE: $$\langle \texttt{!}\, \mathit{false},\, \sigma\rangle \rightarrow \langle \mathit{true}, \sigma\rangle$$

SMALLSTEP-AND1: $$\frac{\langle b_1, \sigma\rangle \rightarrow \langle b_1', \sigma\rangle}{\langle b_1 \ \texttt{and'}\ b_2, \sigma\rangle \rightarrow \langle b_1' \ \texttt{and'}\ b_2, \sigma\rangle}$$

SMALLSTEP-ANDFALSE: $$\langle \mathit{false}\ \texttt{and'}\ b_2,\, \sigma\rangle \rightarrow \langle \mathit{false}, \sigma\rangle$$

SMALLSTEP-ANDTRUE: $$\langle \mathit{true}\ \texttt{and'}\ b_2,\, \sigma\rangle \rightarrow \langle b_2, \sigma\rangle$$

SMALLSTEP-LT1: $$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 \ \texttt{<'}\ a_2, \sigma\rangle \rightarrow \langle a_1' \ \texttt{<'}\ a_2, \sigma\rangle}$$

SMALLSTEP-LT2: $$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle i_1 \ \texttt{<'}\ a_2, \sigma\rangle \rightarrow \langle i_1 \ \texttt{<'}\ a_2, \sigma\rangle}$$

SMALLSTEP-LT: $$\langle i_1 \ \texttt{<'}\ i_2,\, \sigma\rangle \rightarrow \langle i_1 <_{nat} i_2, \sigma\rangle$$

SMALLSTEP-GT1: $$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 \ \texttt{>'}\ a_2, \sigma\rangle \rightarrow \langle a_1' \ \texttt{>'}\ a_2, \sigma\rangle}$$

SMALLSTEP-GT2: $$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 \ \texttt{>'}\ a_2, \sigma\rangle \rightarrow \langle a_1 \ \texttt{>'}\ a_2, \sigma\rangle}$$

SMALLSTEP-GT: $$\langle i_1 \ \texttt{>'}\ i_2,\, \sigma\rangle \rightarrow \langle i_1 >_{nat} i_2, \sigma\rangle$$

The rules for conjunction evaluate the left-hand side argument first. In this case we say that conjunction is *strict* in the first argument.

The rules SMALLSTEP-LT1 and SMALLSTEP-LT2 implement a *sequentially strict* evaluation strategy for the operands of `<'`: SMALLSTEP-LT1 evaluates a_1 as much as possible, while SMALLSTEP-LT2 can be applied only when the first operand i_1 is a concrete value. This is a design choice that we made in order to illustrate how such evaluation strategies can be implemented using small-step SOS. Also, note that this is not possible using big-step SOS.

To summarise, we have illustrated the *strict* evaluation on conjunction, *sequentially strict* evaluation for the less than operator (`<'`), and non-strict or non-deterministic evaluation for addition, multiplication, and the greater than operator (`>'`).

Below is an example of a derivation for the sequent $\langle 1 \ \texttt{+'} \ 3 \ \texttt{<'} \ 5, \sigma\rangle \rightarrow^* \langle \mathit{true}, \sigma\rangle$ using the set of small-step SOS rules discussed so far:

$$\dfrac{\dfrac{\dfrac{\cdot}{\langle 1 \ \texttt{+'} \ 3, \sigma\rangle \rightarrow \langle 4, \sigma\rangle}\textsc{Add}}{\langle 1 \ \texttt{+'} \ 3 \ \texttt{<'} \ 5, \sigma\rangle \rightarrow \langle 4 \ \texttt{<'} \ 5, \sigma\rangle}\textsc{Lt1} \quad \dfrac{\dfrac{\cdot}{\langle 4 \ \texttt{<'} \ 5, \sigma\rangle \rightarrow \langle \mathit{true}, \sigma\rangle}\textsc{Lt} \quad \dfrac{\cdot}{\langle \mathit{true}, \sigma\rangle \rightarrow^* \langle \mathit{true}, \sigma\rangle}\textsc{Refl}}{\langle 4 \ \texttt{<'} \ 5, \sigma\rangle \rightarrow^* \langle \mathit{true}, \sigma\rangle}\textsc{Tran}}{\langle 1 \ \texttt{+'} \ 3 \ \texttt{<'} \ 5, \sigma\rangle \rightarrow^* \langle \mathit{true}, \sigma\rangle}\textsc{Tran}$$

A disadvantage of small-step SOS is that derivations are bigger in size compared to big-step SOS. On the other hand, using a proof-assistant, there is no need to write such tedious proofs on paper. The proof assistant helps the users to write the proofs and also checks them for errors. Recall that Coq cannot accept incorrect proofs.

Exercise 6.3.1 *Write down the small-step SOS rules for disjunctions, less or equal than ($\leq$) and greater or equal than ($\geq$). Write some relevant examples in order to test each small-step rule.*

6.4 Small-step SOS for boolean expressions in Coq

In Coq, the rules for boolean expressions are encoded as follows:

```
Reserved Notation "<| B , S |> -> <| B' , S' |>" (at level 90).
Print BExp.
Inductive beval : BExp → Env → BExp → Env → Prop :=
| not : ∀ b b' s,
    <| b, s |> → <| b', s|> →
    <| ! b, s |> → <| ! b', s|>
| not_true : ∀ s,
    <| ! btrue , s |> → <| bfalse , s |>
| not_false : ∀ s,
    <| ! bfalse , s |> → <| btrue , s |>

| and_1 : ∀ b1 b1' b2 s,
    <| b1 , s |> → <| b1' , s |> →
    <| b1 and' b2 , s |> → <| b1' and' b2 , s |>
| and_false : ∀ b2 s,
    <| bfalse and' b2 , s |> → <| bfalse , s |>
| and_true : ∀ b2 s,
    <| btrue and' b2 , s |> → <| b2 , s |>

| lt_1 : ∀ a1 a2 a1' s,
    (<| a1 , s |> ⇒ <| a1' , s |>) →
```

```
    <| a1 <' a2 , s |> → <| a1' <' a2 , s |>
| lt_2 : ∀ (i1 : nat) a2 a2' s,
    (<| a2 , s |> ⇒ <| a2' , s |>) →
    <| i1 <' a2 , s |> → <| i1 <' a2' , s |>
| lt : ∀ (i1 i2 : nat) s b,
    b = (if Nat.ltb i1 i2 then btrue else bfalse) →
    <| i1 <' i2 , s |> → <| b , s |>

| gt_1 : ∀ a1 a2 a1' s,
    (<| a1 , s |> ⇒ <| a1' , s |>) →
    <| a1 >' a2 , s |> → <| a1' >' a2 , s |>
| gt_2 : ∀ a1 a2 a2' s,
    (<| a2 , s |> ⇒ <| a2' , s |>) →
    <| a1 >' a2 , s |> → <| a1 >' a2' , s |>
| gt : ∀ (i1 i2 : nat) s b,
    b = (if negb (Nat.leb i1 i2) then btrue else bfalse) →
    <| i1 >' i2 , s |> → <| b , s |>
where "<| B , S |> -> <| B' , S' |>" := (beval B S B' S').
```

The reflexive and transitive closure of this relation is encoded as expected:

```
Reserved Notation "<| B , S |> ->* <| B' , S' |>" (at level 90).
Inductive beval_clos : BExp → Env → BExp → Env → Prop :=
| b_refl : ∀ b st, <| b , st |> ->* <| b , st |>
| b_tran : ∀ b1 b2 b3 st,
    (<| b1 , st |> → <| b2 , st |>) →
    (<| b2 , st |> ->* <| b3 , st |>) →
    (<| b1 , st |> ->* <| b3 , st |>)
where "<| B , S |> ->* <| B' , S' |>" := (beval_clos B S B' S').
```

Our previous derivation of $\langle 1 \texttt{ +' } 3 \texttt{ <' } 5, \sigma\rangle \to^* \langle \mathit{true}, \sigma\rangle$ looks like this in Coq:

```
Example ex4 :
  <| 1 +' 3 <' 5, sigma1 |> ->* <| btrue, sigma1 |>.
Proof.
  eapply b_tran with (b2 := 4 <' 5).
  - eapply lt_1.
    eapply add. simpl. reflexivity.
  - eapply b_tran.
    + eapply lt. simpl. reflexivity.
    + eapply b_refl.
Qed.
```

Note that in this proof, `eapply b_tran with (b2 := 4 <' 5)` specifies the value of the *b2* variable from the definition `b_tran` constructor. This is because it is

not obvious for Coq what is the right configuration to use for the *b2* placeholder. A possible improvement for Coq would be to implement some tactics than can automatically detect the right configurations to work with. Other tools (e.g., K, Maude) use matching or unification to tackle this problem.

Exercise 6.4.1 *Implement the solution for Exercise 6.3.1 in Coq, including the examples.*

6.5 Small-step SOS for IMP: statements

The small-step SOS for **IMP** statements is shown below. The first two rules define the semantics of the assignment: there is one rule to evaluate the right-hand side of the assignment, and another rule for performing the actual update of the left-hand side variable in the environment. The next two rules handle sequences by defining a sequentially strict evaluation strategy: the first statement is executed first; when skip is obtained, we move to the second statement. The semantics of ite is the expected one: we evaluate the condition in a small-step fashion, and then we use two rules to distinguish between the two cases determined by the condition value. Finally, the loop is simply a sintactic sugar for an ite statement.

SMALLSTEP-ASSIGN-2: $$\frac{\langle a, \sigma \rangle \to \langle a', \sigma \rangle}{\langle x \;\texttt{::=}\; a,\, \sigma \rangle \to \langle x \;\texttt{::=}\; a',\, \sigma \rangle}$$

SMALLSTEP-ASSIGN: $$\langle x \;\texttt{::=}\; i,\, \sigma \rangle \to \langle \mathsf{skip}, \sigma[i/x] \rangle$$

SMALLSTEP-SEQ: $$\frac{\langle s_1, \sigma \rangle \to \langle s_1', \sigma' \rangle}{\langle s_1 \;\texttt{;;}\; s_2,\, \sigma \rangle \to \langle s_1' \;\texttt{;;}\; s_2,\, \sigma' \rangle}$$

SMALLSTEP-SKIP: $$\langle \mathsf{skip} \;\texttt{;;}\; s_2,\, \sigma \rangle \to \langle s_2, \sigma \rangle$$

SMALLSTEP-ITE: $$\frac{\langle b,\, \sigma \rangle \to \langle b', \sigma \rangle}{\langle \mathsf{ite}\, b\, s_1\, s_2,\, \sigma \rangle \to \langle \mathsf{ite}\, b'\, s_1\, s_2,\, \sigma \rangle}$$

SMALLSTEP-ITETRUE: $$\langle \mathsf{ite}\; \mathsf{btrue}\, s_1\, s_2,\, \sigma \rangle \to \langle s_1,\, \sigma \rangle$$

SMALLSTEP-ITEFALSE: $$\langle \mathsf{ite}\; \mathsf{bfalse}\, s_1\, s_2,\, \sigma \rangle \to \langle s_2,\, \sigma \rangle$$

SMALLSTEP-WHILE: $$\langle \mathsf{while}\, b\, s,\, \sigma \rangle \to \langle \mathsf{ite}\, b\, (s \;\texttt{;;}\; \mathsf{while}\, b\, s)\; \mathsf{skip},\, \sigma \rangle$$

Exercise 6.5.1 *Write the small-step SOS rules for the* if-then *statement.*

We show here a derivation example for the sequent $\langle x\texttt{::=}n, \sigma_1\rangle \rightarrow \langle \texttt{skip}, \sigma_2\rangle$, where $\sigma_1(n) = 10$ and $\sigma_2 = \sigma_1[10/x]$.

$$\dfrac{\dfrac{\dfrac{\cdot}{\langle n, \sigma_1\rangle \rightarrow \langle 10, \sigma_1\rangle}\ \textsc{Lookup}}{\langle x\texttt{::=}n, \sigma_1\rangle \rightarrow \langle x\texttt{::=}10, \sigma_1\rangle}\ \textsc{Assign2} \qquad \dfrac{\dfrac{\cdot}{\langle x\texttt{::=}10, \sigma_1\rangle \rightarrow \langle \texttt{skip}, \sigma_2\rangle}\ \textsc{Assign} \qquad \dfrac{\cdot}{\langle \texttt{skip}, \sigma_2\rangle \rightarrow^* \langle \texttt{skip}, \sigma_2\rangle}\ \textsc{Refl}}{\langle x\texttt{::=}10, \sigma_1\rangle \rightarrow^* \langle \texttt{skip}, \sigma_2\rangle}\ \textsc{Tran}}{\langle x\texttt{::=}n, \sigma_1\rangle \rightarrow^* \langle \texttt{skip}, \sigma_2\rangle}\ \textsc{Tran}$$

As noted in the previous section, even for simple sequents, the derivations are bigger in size. Fortunately, all these can be encoded in Coq, and the proof assistant helps the users to write correct derivations.

6.6 Small-step SOS for statements in Coq

The Coq encoding of the small-step SOS for statements is shown below, together with the reflexive and transitive closure:

```
Reserved Notation "<{ S , E }> ↷ <{ S' , E' }>" (at level 90).

Inductive eval : Stmt → Env → Stmt → Env → Prop :=
| eassign_2: ∀ var a a' st,
    <| a , st |> ⇒ <| a' , st |> →
    <{ var ::= a , st }> ↷ <{ var ::= a' , st }>
| eassign: ∀ var st st' n,
    st' = update st var n →
    <{ var ::= (anum n), st }> ↷ <{ skip , st' }>
| eskip: ∀ s2 st,
    <{ skip ;; s2 , st }> ↷ <{ s2 , st }>
| eseq: ∀ s1 s1' s2 st st',
    <{ s1 , st }> ↷ <{ s1' , st' }> →
    <{ s1 ;; s2 , st }> ↷ <{ s1' ;; s2 , st' }>
| eite: ∀ b b' st s1 s2,
    <| b , st |> → <| b' , st |> →
    <{ ite b s1 s2 , st }> ↷ <{ ite b' s1 s2 , st }>
| eite_true : ∀ s1 s2 st,
    <{ ite btrue s1 s2 , st }> ↷ <{ s1 , st }>
| eite_false : ∀ s1 s2 st,
    <{ ite bfalse s1 s2 , st }> ↷ <{ s2 , st }>
| ewhile: ∀ b b' s st,
    <{ while b s , st }> ↷ <{ ite b (s ;; while b' s) skip , st }>
where "<{ S , E }> ↷ <{ S' , E' }>" := (eval S E S' E').

Reserved Notation "<{ S , E }> ↷* <{ S' , E' }>" (at level 90).

Inductive eval_clos : Stmt → Env → Stmt → Env → Prop :=
| refl : ∀ S E, <{ S , E }> ↷* <{ S , E }>
```

```
| tran : ∀ s1 s2 s3 st1 st2 st3,
    <{ s1 , st1 }> ↷ <{ s2 , st2 }> →
    <{ s2 , st2 }> ↷* <{ s3 , st3 }> →
    <{ s1 , st1 }> ↷* <{ s3 , st3 }>
where "<{ S , E }> ↷* <{ S' , E' }>" := (eval_clos S E S' E').
```

The derivation for the sequent $\langle x\texttt{::=}n, \sigma_1\rangle \to \langle \texttt{skip}, \sigma_2\rangle$ (where $\sigma_1(n) = 10$ and $\sigma_2 = \sigma_1[10/x]$) in Coq is the following:

```
Example assign_1 :
    <{ "x" ::= "n" , sigma1 }> ↷* <{ skip, update sigma1 "x" 10}>.
Proof.
  apply tran with (s2 := "x" ::= 10) (st2 := sigma1).
  - eapply eassign_2.
    eapply lookup.
  - apply tran with (s2 := skip) (st2 := update sigma1 "x" 10).
    + eapply eassign.
      reflexivity.
    + eapply refl.
Qed.
```

Exercise 6.6.1 *Implement the solution of Exercise 6.5.1 in Coq.*

6.7 Big-step vs. Small-step SOS

The two SOS semantic styles discussed so far are meant to solve slightly different purposes. Small-step SOS is clearly more suitable to model concurrency, divergence, run-time errors, exceptions, breaks, jumps. This is because the one-step evaluation allows us to express the semantics of such complex features. On the other hand, it is a bit too detailed and implies a greater effort on the language designer side.

Big-step semantics is like defining a recursive interpreter: for a given language, its behaviour can be defined using several recursive rules. Reasoning using big-step semantics is definitely easier than reasoning with small-step rules. The main disadvantage, is that big-step is not as expressive as small-step. Concurrency and other complex features cannot be modelled using big-step, and the evaluation skips intermediate steps.

Exercise 6.7.1 *The goal of this exercise is to improve the small-step SOS of* **IMP** *with concurrency. Basically, you have to add a new language construct* com s_1 s_2 *in* **IMP** *that can execute the two statements* s_1 *and* s_2 *in the same time.*

It is possible to change the big-step SOS of **IMP** *in the same way? Justify your answer.*

Exercise 6.7.2 *For the* **IMP** *language, what is the relationship between big-step SOS and small-step SOS? Does the execution of a program yield the* same *result when executed using big-step SOS vs. small-step SOS?*

One big disadvantage of both big-step and small-step SOS is the lack of modularity. A small change in the configuration (e.g., adding a new component like a stack or a program counter) requires the modification of *all* rules! Most of the times, the modification is not even relevant to the majority of the rules, and only a few rules may refer to that new component of the configuration. This is probably the main issue with these two SOS styles. The issue is serious and most language designers don't find these techniques useful in practice. A very good reading material on the limitations of big-step and small-step SOS styles is given by Peter Mosses in [**?**]. In the same material the reader can find a solution to the lack o modularity of Big-step and Small-step semantics: MSOS - modular structural operational semantics.

Although big-step and small-step semantical styles have certain limitations, these establish the basic grounds to define programming languages. Other later approaches simply try to fix these traditional styles by adding modularity or embedding it in logics that facilitate execution (e.g., Rewriting Logic [**?**], Reachability Logic [**?**, **?**]).

Chapter 7

Type systems

7.1 Introduction

Type systems are essential in programming languages. Most programming languages have values categorised in certain *types* so that programmers can distinguish them and use them in particular contexts. For instance, Coq is *strongly-typed*: it does have very strict typing rules that do determine the compiler to throw errors at compile time. If a Coq function takes two integers, but it is called with a boolean argument, the compiler `coqc` will complain:

```
-# cat coq_error.v
Definition f (a b : nat) : nat :=
  a + b.
Check f.
Compute f true.
-# coqc coq_error.v
f
     : nat -> nat -> nat
File "./coq_error.v", line 4, characters 10-14:
Error:
The term "true" has type "bool" while it is expected to have type
"nat".
```

Other programming languages are not that strict. The popular Javascript language is *weakly-typed*: the typing rules are loose and they typically allow values to be interpreted as having a particular type depending on the context. A disadvantage is that weak typing can lead to errors during the execution of a program or some unexpected behaviours in the eye of the unexperienced programmer. Here are some interesting examples using the Nodejs interpreter in command line:

```
-# node
```

```
Welcome to Node.js v14.17.5.
Type ".help" for more information.
> 2 + '4'
'24'
> 2 * '4'
8
> '2' * '4'
8
> 2 * 'a'
NaN
> '2' * 'a'
NaN
> 2 + false
2
> 2 + true
3
> false - true
-1
> 2 && true
true
> true && 2
2
> 0 && false
0
> false && 0
false
> [] == ![]
true
> [] == false
true
> ![] == false
true
```

Compared to Coq, the above example seem to be a bit silly: why on earth would `2 + '4'` return `'24'`, while `2 * '4'` returns `8`? In the first case it looks like `2` is converted to `'2'` and then the strings are concatenated. In the second case, it looks like `'4'` is converted to `4` and the result is indeed `2 * 4`, i.e., `8`. At a closer look, this could make sense if `+` is used for both string concatenation and addition of numbers. Also, booleans are treated as numbers when convenient, and numbers can be treated as booleans. However, the last four examples are confusing for a newcomer.

Using numbers as booleans is common in C as well. For instance, in the program below, the value printed by the program depends on the value of the variable `x`:

```
#include <stdio.h>
int main() {
  int x = _;
  printf("x = %d, ", x);
  if (x)
    {
      printf("took the 'then' branch\n");
    }
  else
    {
      printf("took the 'else' branch\n");
    }
}
```

If the _ is zero, the execution of the `if` statement will print `took the 'else' branch`. Otherwise, it will print `took the 'then' branch`. It is obvious that `x` is interpreted by the `if` statement as a boolean: zero is used to represent false and anything else is used to represent true.

Based on the above examples, one might be tempted to say that strongly-typed is good and weakly-typed is bad. But both approaches have their advantages and disadvantages. A strongly-typed approach provides some advantages in terms of correctness, documentation, safety, optimisation, abstraction. A weakly-typed approach provides the possibility to use the same value in different contexts and leaves the interpreter to choose the type. In the right hands, this can be turned into an advantage and can improve the productivity.

In this chapter we mix the arithmetic and boolean expressions of **IMP**, and we present a type system for them. We show the rules of the type system, we integrate them in Coq, and then we use them to prove properties of the proof system and typing properties for programs.

7.2 IMP Syntax of expressions

The syntax of expressions that we are going to use in this chapter is shown here:

```
Inductive Exp :=
| avar : string → Exp
| anum : nat → Exp
| aplus : Exp → Exp → Exp
| amul : Exp → Exp → Exp
| btrue : Exp
| bfalse : Exp
| bnot : Exp → Exp
| band : Exp → Exp → Exp
```

```
| blessthan : Exp → Exp → Exp
| bgreaterthan : Exp → Exp → Exp.
```

```
Coercion anum : nat >-> Exp.
Coercion avar : string >-> Exp.
Notation "A +' B" := (aplus A B) (at level 50, left associativity).
Notation "A *' B" := (amul A B) (at level 40, left associativity).
Notation "A <' B" := (blessthan A B) (at level 80).
Notation "A >' B" := (bgreaterthan A B) (at level 80).
Infix "and'" := band (at level 82).
Notation "! A" := (bnot A) (at level 81).
```

It is now possible to mix booleans with numbers or other arithmetic expressions. The following checks do not fail:

```
Compute (2 +' 2).
    = 2 +' 2
    : Exp
Compute (2 +' btrue).
    = 2 +' btrue
    : Exp
Compute (band 2 2).
```

Exercise 7.2.1 *Rewrite the syntax of the* **IMP** *statements using the expressions above. Is it now possible to use numbers as booleans in conditional statements as we do in C? What about writing expressions as we did in our Javascript examples?*

7.3 Small-step SOS for expressions

The small-step SOS semantics of the expressions is straightforward: we simply assemble all the small-step rules for arithmetic and boolean expressions:

```
Reserved Notation "A =[ S ]=> N" (at level 60).
```

```
Inductive exp_eval_small_step : Exp → Env → Exp → Prop :=
| const : ∀ n st, anum n =[ st ]=> n
| lookup : ∀ v st, avar v =[ st ]=> (st v)
| add_1 : ∀ a1 a2 a1' st a,
    a1 =[ st ]=> a1' →
    a = a1' +' a2 →
    a1 +' a2 =[ st ]=> a
| add_2 : ∀ a1 a2 a2' st a,
    a2 =[ st ]=> a2' →
    a = a1 +' a2' →
    a1 +' a2 =[ st ]=> a
| add : ∀ i1 i2 st n,
```

```
      n = anum (i1 + i2) →
      anum i1 +' anum i2 =[ st ]=> n
  | times_1 : ∀ a1 a2 a1' st a,
      a1 =[ st ]=> a1' →
      a = a1' *' a2 →
      a1 *' a2 =[ st ]=> a
  | times_2 : ∀ a1 a2 a2' st a,
      a2 =[ st ]=> a2' →
      a = a1 *' a2' →
      a1 *' a2 =[ st ]=> a
  | times : ∀ i1 i2 st n,
      n = anum (i1 + i2) →
      anum i1 *' anum i2 =[ st ]=> n
  | true : ∀ st, btrue =[ st ]=> btrue
  | false : ∀ st, bfalse =[ st ]=> bfalse
  | lessthan_1: ∀ a1 a2 a1' st,
      a1 =[ st ]=> a1' →
      (a1 <' a2) =[ st ]=> (a1' <' a2)
  | lessthan_2: ∀ i1 a2 a2' st,
      a2 =[st]=> a2' →
      ((anum i1) <' a2) =[ st ]=> ((anum i1) <' a2')
  | lessthan: ∀ i1 i2 st b,
      b = (if Nat.leb i1 i2 then btrue else bfalse) →
      ((anum i1) <' (anum i2)) =[ st ]=> b
  | greaterthan_1: ∀ a1 a2 a1' st,
      a1 =[ st ]=> a1' →
      (a1 >' a2) =[ st ]=> (a1' >' a2)
  | greaterthan_2: ∀ i1 a2 a2' st,
      a2 =[st]=> a2' →
      ((anum i1) >' a2) =[ st ]=> ((anum i1) >' a2')
  | greaterthan: ∀ i1 i2 st b,
      b = (if negb (Nat.leb i1 i2) then btrue else bfalse) →
      ((anum i1) >' (anum i2)) =[ st ]=> b
  | not : ∀ b b' st,
      b =[ st ]=> b' →
      (bnot b) =[ st ]=> (bnot b')
  | nottrue : ∀ st,
      (bnot btrue) =[ st ]=> bfalse
  | notfalse : ∀ st,
      (bnot bfalse) =[ st ]=> btrue
  | and_1 : ∀ b b1 b1' b2 st,
      b1 =[st]=> b1' →
      b = (band b1' b2) →
```

```
    (band b1 b2) =[ st ]=> b
| andtrue : ∀ b2 st,
    (band btrue b2) =[st]=> b2
| andfalse : ∀ b2 st,
    (band bfalse b2) =[st]=> bfalse
where "A =[ S ]=> N" := (exp_eval_small_step A S N).

Reserved Notation "A =[ S ]>* A'" (at level 60).
Inductive eval_clos : Exp → Env → Exp → Prop :=
| refl : ∀ a st, a =[ st ]>* a
| tran : ∀ a1 a2 a3 st, (a1 =[st]=> a2) → a2 =[ st ]>* a3 → a1 =[ st ]>* a3
where "A =[ S ]>* A'" := (eval_clos A S A').
```

The semantics behaves as expected. Here are some examples:

```
Example e1 :
  2 +' "n" =[ sigma1 ]=> 2 +' 10.
Proof.
  eapply add_2.
  eapply lookup. unfold sigma1, update. simpl. reflexivity.
Qed.

Example e2 :
  2 +' "n" =[ sigma1 ]>* anum 12.
Proof.
  eapply tran.
  - eapply add_2.
    eapply lookup. eauto.
  - eapply tran.
    + eapply add. eauto.
    + simpl. eapply refl.
Qed.
```

However, this allows us to write some ill formed expressions. For instance:

```
Example ill-formed :
  2 +' btrue =[ sigma1 ]>* ?.
```

In a weakly-typed language, the compiler or the interpreter does some automatic conversions (e.g., remember the Javascript examples). For a strongly-typed language, an error is thrown. For instance, in Coq you get an error right away, since Coq checks the types of the expressions *statically* (remember the Coq example at the beginning of this chapter). In Python, the error is thrown *dynamically*, that is, the types are checked during the execution:

```
-# python3
Python 3.9.6 (default)
```

```
[Clang 12.0.5 (clang-1205.0.22.9)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> 3 + 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Definitely, getting such errors during execution is not desirable. It would be easier for the programmer to a have a tool that signals whether such terms could lead to runtime errors. Our semantics of the mixed expressions simply does not care about reducing such terms. So, 2 `+'` `btrue` cannot be reduced to a value, and in the same time, it is not itself a value. We say that such terms (that are not values and cannot be reduced to a value) are *stuck*. How do we detect stuck terms? The answer is simple: define a type system for our language. The type system helps us to exclude nonsensical terms by simply checking their types.

7.4 A type system for expressions

Our expressions include both numbers and boolean values. In order to exclude terms that we do not want to have a meaning (e.g., 2 `+'` `btrue`), we define a typing relation that relates terms to the types of their final (evaluated) results.

We are going to use some conventional notations. The set of basic types is denoted by $\mathcal{T}$. For our expressions, it is sufficient to choose the set $\mathcal{T} = \{Nat, Bool\}$. For more complex languages, this set can be larger. By $e : Nat$ we denote the fact that the expression e has type Nat. The typing rules are given using inference rules. Checking whether a given expression has a certain type reduces to finding a derivation (i.e., a proof) using the typing rules.

The typing rules for our expressions are shown below. The conclusions of all rules cover the entire syntax of our expressions. The rules are straightforward. The constants have the expected types. Unlike TTrue and TFalse, TNum and TVar have side conditions because the set of naturals and the set of variables are infinite. All the other rules have the expected premisses. For instance, TLt reads as follows: the expression e_1 `<'` e_2 has type $Bool$ if both e_1 and e_2 are of type Nat. Similarly, TPlus says that e_1 `+'` e_2 has type Nat when e_1 and e_2 are of type Nat. Therefore, it is impossible to find a derivation for 2 `+'` `btrue` : Nat or 2 `+'` `btrue` : $Bool$.

TNum: $$\frac{\cdot}{n : \mathit{Nat}}\ n \in \mathbb{N}$$

TVar: $$\frac{\cdot}{x : \mathit{Nat}}\ x \in \mathit{Var}$$

TPlus: $$\frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1\ \texttt{+'}\ e_2 : \mathit{Nat}}$$

TMul: $$\frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1\ \texttt{*'}\ e_2 : \mathit{Nat}}$$

TTrue: $$\mathsf{btrue} : \mathit{Bool}$$

TFalse: $$\mathsf{bfalse} : \mathit{Bool}$$

TNot: $$\frac{e : \mathit{Bool}}{\mathsf{bnot}\ e : \mathit{Bool}}$$

TAnd: $$\frac{e_1 : \mathit{Bool} \qquad e_2 : \mathit{Bool}}{\mathsf{band}\ e_1 e_2 : \mathit{Bool}}$$

TLt: $$\frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1\ \texttt{<'}\ e_2 : \mathit{Bool}}$$

TGt: $$\frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1\ \texttt{>'}\ e_2 : \mathit{Bool}}$$

Here is an example of a derivation for $2\ \texttt{+'}\ x : \mathit{Nat}$, where $2 \in \mathbb{N}$ and $x \in \mathit{Var}$ (i.e., x is a variable):

$$\dfrac{\dfrac{\cdot}{2 : \mathit{Nat}}\ \text{TNum} \qquad \dfrac{\cdot}{x : \mathit{Nat}}\ \text{TVar}}{2\ \texttt{+'}\ x : \mathit{Nat}}\ \text{TPlus}$$

Here is a another derivation for $2\ \texttt{+'}\ x\ \texttt{<'}\ x : \mathit{Bool}$:

$$\dfrac{\dfrac{\dfrac{\cdot}{2 : \mathit{Nat}}\ \text{TNum} \qquad \dfrac{\cdot}{x : \mathit{Nat}}\ \text{TVar}}{2\ \texttt{+'}\ x : \mathit{Nat}}\ \text{TPlus} \qquad \dfrac{\cdot}{x : \mathit{Nat}}\ \text{TVar}}{2\ \texttt{+'}\ x\ \texttt{<'}\ x : \mathit{Bool}}\ \text{TLt}$$

An important observation is that typing derivations are different from evaluating expressions. Obviously, if x is of type *Nat* then the expression $2\ \texttt{+'}\ x\ \texttt{<'}\ x$ cannot be true. But the expression $2\ \texttt{+'}\ x\ \texttt{<'}\ x$ is still typable and it does have type *Bool*. So, type checking is not the same thing as evaluating expressions.

Exercise 7.4.1 *What is the type of the expression:*

$$\mathsf{band}\ (3\ \texttt{*'}\ x\ \texttt{+'}\ 7\ \texttt{<'}\ x)\ (\mathsf{bnot}\ (y\ \texttt{>'}\ x))?$$

Write down a derivation that justifies your answer.

Exercise 7.4.2 *We define the syntax of an expression language called* PLAYWITHTYPES*. The BNF syntax of expressions is shown below:*

$$E ::= \mathsf{O} \mid \mathsf{S}\ E \mid \mathsf{isZero}\ E \mid \mathsf{T} \mid \mathsf{F} \mid \mathsf{ite}\ EEE.$$

The constructors are written using this font*. We notice that the syntax of* PLAYWITHTYPES *is in fact a mix of Peano naturals (*O *and* S *are the usual constructors) and booleans (*T *and* F *are well-known boolean constants). The* ite *construct is an* if-then-else*-like statement that returns an expression: it takes a boolean and two natural expressions and it should return a natural expression. The predicate* isZero *should tell us whether the given natural expression is zero or not.*

Based on the above assumptions, write down the typing rules for this language. Also, write examples that cover all the constructs of the language.

7.5 A type system for expressions in Coq

All the above definitions and rules can be easily encoded in Coq. We start by defining the set of basic types **Typ** (that is, the set $\mathcal{T}$):

```
Inductive Typ : Type :=
| Bool : Typ
| Nat : Typ.
```

The typing rules are shown below. In Coq we use **type_of** instead of ':'. So, **type_of** x Nat stands for $x : Nat$.

```
Inductive type_of : Exp → Typ → Prop :=
| t_num : ∀ n, type_of (anum n) Nat
| t_var : ∀ x, type_of (avar x) Nat
| t_plus : ∀ a1 a2,
    (type_of a1 Nat) →
    (type_of a2 Nat) →
    (type_of (a1 +' a2) Nat)
| t_mul : ∀ a1 a2,
    (type_of a1 Nat) →
    (type_of a2 Nat) →
    (type_of (a1 *' a2) Nat)
| t_true : type_of btrue Bool
| t_false : type_of bfalse Bool
| t_not : ∀ b,
```

```
      (type_of b Bool) →
      (type_of (bnot b) Bool)
| t_and : ∀ b1 b2,
      (type_of b1 Bool) →
      (type_of b2 Bool) →
      (type_of (band b1 b2) Bool)
| t_lt : ∀ a1 a2,
      (type_of a1 Nat) →
      (type_of a2 Nat) →
      (type_of (a1 <' a2) Bool)
| t_gt : ∀ a1 a2,
      (type_of a1 Nat) →
      (type_of a2 Nat) →
      (type_of (a1 >' a2) Bool).
```

Note that the side conditions of TNum and TVar are enforced by the fact that the constructors anum and avar accept the right types for their arguments.

Our simple derivation example for 2 +' x : *Nat* is trivial in Coq:

```
Example well_typed:
  type_of (2 +' "x") Nat.
Proof.
  eapply t_plus.
  - eapply t_num.
  - eapply t_var.
Qed.
```

We can define a hint database and improve our experience by letting Coq search for the derivation. Here is the same example, but we use `auto`:

```
Create HintDb types.
Hint Constructors type_of : types.

Example well_typed':
  type_of (2 +' "x") Nat.
Proof.
  auto with types.
Qed.
```

In addition, here are two other examples that we have previously mentioned:

```
Example well_typed_but_false:
  type_of (2 +' "x" <' "x") Bool.
Proof.
  auto with types.
Qed.

Example complex_derivation:
  type_of (band (3 *' "x" +' 7 <' "x") (bnot ("y" >' "x"))) Bool.
```

```
Proof.
  auto with types.
Qed.
```

Remember that we started this section by discussing the 2 +' btrue example. Now that we have types and a Coq implementation, our proof search fails to find a derivation for 2 +' btrue : *Nat*:

```
Example ill_typed:
  type_of (2 +' btrue) Nat.
Proof.
  (*  can't prove that *)
  auto with types.
Abort.
```

In general, the fact that `auto` fails to find a proof does not mean that there isn't one. Our point here is to show that `auto` fails. Even if we increase the search depth as much as we want, it will still fail. The same happens for 2 +' btrue : *Bool*. We address the properties of our type system later in a dedicated section.

Exercise 7.5.1 *Implement in Coq the syntax and the type system that you developed for the* PlayWithTypes *language from Exercise 7.4.2. Also, use your implementation to automatically find the derivations for the examples that you are asked to write in Exercise 7.4.2.*

7.6 Properties of the typing relation

The syntax of our expressions allows us to write various expressions. Repetitive applications of the small step rules to an expression may produce either an infinite loop or a term that cannot be reduced anymore (i.e., none of the existing rules can be applied to it). Terms that cannot be reduced anymore are said to be in a *normal form*. Values (e.g., natural numbers or boolean constants) are in a normal form. Terms that cannot progress (e.g., 2 +' btrue) are also in normal form. Such terms are stuck: they have no successors w.r.t. small-step rules.

Our goal is to use the typing relation to distinguish between expressions, so that expressions that do not make sense are excluded. But how do we know that our typing relation does this in a correct way? Can we prove some properties about our typing relation so that we can guarantee that it is correct? What are these properties?

In the rest of this section we are going to formulate and prove some common properties of typing relations: progress, type preservation and soundness.

7.6.1 Progress

Progress is a property that captures the fact that well-typed expressions make progress w.r.t. the small-step rules. Obviously, values cannot make progress be-

cause they are already reduced. First, we define what is a value in Coq:

```
Inductive nat_value : Exp → Prop :=
| nat_val : ∀ n, nat_value (anum n).

Inductive b_value : Exp → Prop :=
| b_true : b_value btrue
| b_false : b_value bfalse.

Definition value (e : Exp) := nat_value e ∨ b_value e.
```

The above definitions of the values agree with the typing relation. This is captured by the following lemmas:

```
Lemma bool_canonical :
  ∀ e, type_of e Bool → value e → b_value e.
Proof.
  intros e H H'.
  inversion H'; trivial.
  inversion H; subst; inversion H0.
Qed.

Lemma nat_canonical :
  ∀ e, type_of e Nat → value e → nat_value e.
Proof.
  intros e H H'.
  inversion H'; trivial.
  inversion H; subst; inversion H0.
Qed.
```

Now, we are ready to formulate and prove the progress property in Coq:

```
Theorem progress :
  ∀ t T state,
    type_of t T →
    value t ∨ ∃ t', t =[ state ]=> t'.
Proof.
  intros t T state H.
  induction H; eauto with types.
  - destruct IHtype_of1 as [Ha1 | [t' Ha1]]; eauto with types.
    apply nat_canonical in H; auto.
    inversion H; eauto with types.
  - destruct IHtype_of1 as [Ha1 | [t' Ha1]] eauto with types.
    apply nat_canonical in H; auto.
    inversion H; eauto with types.
  - destruct IHtype_of as [Hb | [t' Hb]] eauto with types.
    apply bool_canonical in H; auto.
    inversion H; eauto with types.
  - destruct IHtype_of1 as [Hb1 | [t' Hb1]] eauto with types.
```

```
    apply bool_canonical in H; auto.
    inversion H; eauto with types.
  - destruct IHtype_of1 as [Hb1 | [t' Hb1]] eauto with types.
    apply nat_canonical in H; auto.
    inversion H; eauto with types.
  - destruct IHtype_of1 as [Hb1 | [t' Hb1]] eauto with types.
    apply nat_canonical in H; auto.
    inversion H; eauto with types.
Qed.
```

The proof is by induction on the typing relation. Note that our hint database has been enriched with new hints in order to shorten the length of the Coq proof. Here is the complete hint database that we used:

```
Hint Constructors exp_eval_small_step : types.
Hint Constructors type_of : types.
Hint Constructors nat_value : types.
Hint Constructors b_value : types.
Hint Unfold update.
```

7.6.2 Type Preservation

The type preservation property is critical: it says that if a well-typed expression is reduced in one step using the small-step rules, the obtained expression is also well-typed. The theorem is formulated below and its proof is by induction on the typing relation:

```
Theorem preservation :
  ∀ t T t' state,
    type_of t T →
    t =[ state ]=> t' →
    type_of t' T.
Proof.
  intros t T t' state H H0. revert t' H0.
  induction H; intros t' H'; inversion H'; subst; eauto with types;
    case_eq (Nat.leb i1 i2); intros H1; rewrite H1 in *; eauto with types.
Qed.
```

7.6.3 Type Soundness

Finally, we are ready to formulate and prove the soundness of our typing relation.

```
Corollary soundness :
  ∀ t t' state T,
    type_of t T →
    t =[ state ]>* t' →
```

```
    value t' ∨ ∃ t'', t' =[ state ]=> t''.
Proof.
  intros t t' state T H S.
  induction S.
  - apply progress with (state := st) in H; eauto.
  - apply IHS. eapply preservation; eauto.
Qed.
```

The property is essentially saying that a well-typed expression cannot reach a stuck state. In other words, if we reduce a well-typed expression using our small-step semantics we always reach a value (that is, a result) or an expression that can be further reduced.

The implications of this formal property are very strong: basically, if a program can be proved to be well-typed (an we can do that using the typing relation) then it is guaranteed not to produce erroneous results that can lead to unexpected behaviours in programs. Typing improves the quality of our programs and some common errors in programs can be avoided using type systems.

Exercise 7.6.1 *Formulate and prove the progress, type preservation and type soundness properties for the typing relation that of the* PlayWithTypes *language (see Exercises 7.4.2 and 7.5.1).*

Chapter 8

Compilation

Compilers are programs that we use to *translate* computer code written in a *source* language into code written in a *target* language. Usually, source languages are high-level languages, while target languages are low-level languages (e.g., object code, machine code, assembly).

The main goal of a compiler is to facilitate the work of the programmer: a high-level language has the advantage of leaving the programmer to focus on the task to be solved rather than dealing with cumbersome low-level programming. You can imagine that writing a complex system, e.g. an operating system, into assembly language is really hard. Actually, the first operating systems were written in assembly languages. Before 1960's, the main issue with compilers was the lack of resources (processing power and memory). Afterwards, companies started to develop machines with better performances and it became more and more appealing to create compilers for high-level portable languages.

The earliest computers (mainframes) did not have any form of operating system. Programs were written on punched cards or magnetic tapes. The program was loaded on the computer and the programmer had to wait until the program completed or crashed. Later on, symbolic languages and assemblers were developed. Programs written in these languages were translated into machine code. This was the very beginning of compilers. In 1952, Grace Hopper created a compiler for the A-0 language. Then, in 1957 at IBM, John Backus and his team FORTRAN introduced the first complete compiler. Later, COBOL was among the languages compiled on multiple architectures.

The idea of having a program that translates high-level code into low-level code caught quickly. New programming languages have been proposed, the software became more complex and compilers started to be themselves really complex. Tim Hart and Mike Levin at MIT created a compiler for the Lisp language in 1962. In 1970s it was very common to implement compilers for a language in the same language or a different common high-level language (e.g., Pascal, C).

A compiler needs to take care of many things: preprocessing, lexical analysis, parsing (and disambiguation), semantic analysis (e.g., type inference, type check-

ing, object binding, various initialisations for local variables, etc), control-flow graphs and corresponding analysis (dependencies, alias analysis, pointer analysis), optimisations (inline expansion, macros, dead code elimination, constant propagation, loop transformation, automatic parallelisation), code generation (machine dependent optimisations, machine dependent generation). Implementing a compiler is really hard and one of the greatest challenges is to make sure that compilers generate programs that are equivalent with the intended behaviour of the compiled programs. Unfortunately, compilers can have bugs and this equivalence is hard to achieve. The interested reader can investigate such bugs in this paper [?]. However, there are approaches that prove their compilers correct. One such project is CompCert [?]: a verified compiler for C (ISO C99) that generates efficient code for PowerPC, ARM, RISC-V and x86 processors. In the CompCert manual, the first chapter (`https://compcert.org/man/manual001.html`) answers the question *Can you trust a compiler?* The CompCert manual explains how important is correct compilation and why miscompilation can insert bugs in the generated compiled code. The CompCert compiler distinguishes itself from the other compilers by the fact that it provides a proof for the *semantics preservation theorem*: intuitively, this theorem says that the semantics of the compiled program is the same as the semantics of the code generated by the compiler for that program. This theorem is formulated and proved in Coq, and so, its proof comes with a high-level of trust.

In this chapter we are going to measure the depth of the ocean using a finger: we will define a compiler for a very simple language of arithmetic expressions. Obviously, our is not as complex as CompCert, but our goal is to explain the principle of building a correct compiler.

8.1 Simple expressions

The language of expressions that we are going to use is very simple: it contains variables, numbers, addition an multiplication:

```
Require Import String.
Require Import List.
Import ListNotations.
Open Scope string_scope.

Inductive Exp :=
| avar : string → Exp
| anum : nat → Exp
| aplus : Exp → Exp → Exp
| amul : Exp → Exp → Exp.

Coercion anum : nat >-> Exp.
Coercion avar : string >-> Exp.
Notation "A +' B" := (aplus A B) (at level 50).
```

```
Notation "A *' B" := (amul A B) (at level 40).
```

Exercise 8.1.1 *Execute the following commands to test the above definition:*

```
Compute (anum 5).
Compute (avar "a").
Compute "a" +' 4.
Compute "a" *' 4.
```

The next obvious step is to define an interpreter for our language. To serve our purpose, it is sufficient to define an interpreter function (rather than a more complex SOS semantics):

```
Fixpoint interpret (e : Exp)
          (env : string → nat) : nat :=
  match e with
  | anum c ⇒ c
  | avar x ⇒ (env x)
  | aplus e1 e2 ⇒ (interpret e1 env) + (interpret e2 env)
  | amul e1 e2 ⇒ (interpret e1 env) × (interpret e2 env)
  end.
```

Exercise 8.1.2 *Test the above function in the environment* Env *(defined below) using the following commands:*

```
Definition Env :=
  fun x ⇒ if string_dec x "a" then 10 else 0.

Compute (Env "a").
Compute (Env "b").

Compute (interpret 5 Env).
Compute (interpret "a" Env).
Compute (interpret ("a" +' 4) Env).
Compute (interpret ("a" *' 4) Env).
```

8.2 A stack machine

Compiled code runs on a very basic stack machine that we need to define in Coq. This machine will execute some low level assembly-like instructions. Here is the syntax of our low level instructions:

```
Inductive Instruction :=
| push_const : nat → Instruction
| push_var : string → Instruction
| add : Instruction
| mul : Instruction.
```

The first two instructions push_const and push_var are meant to push values onto a stack. The latter is a bit more special, because it has to read the value of the given variable and then put this value on the top of the stack. Note that the stack cannot hold variables, but only values. The last two instructions add and mul are meant to add, and respectively, multiply two values from the top of the stack. The precise semantics of these instructions is given below:

```
Fixpoint run_instruction
           (i : Instruction)
           (env : string → nat)
           (stack : list nat) :=
  match i with
  | push_const c ⇒ (c :: stack)
  | push_var x ⇒ ((env x) :: stack)
  | add ⇒ match stack with
           | n1 :: n2 :: stack' ⇒ (n1 + n2) :: stack'
           | _ ⇒ stack
           end
  | mul ⇒ match stack with
           | n1 :: n2 :: stack' ⇒ (n1 × n2) :: stack'
           | _ ⇒ stack
           end
  end.
```

Note that run_instruction can be executed in the presence of an environment and a stack. The execution of run_instruction produces a new stack, that is, the old stack modified according to the instruction that was executed.

Exercise 8.2.1 *Run the commands below in Coq. The results are the expected ones (pay attention to the last two)?*

```
Compute (push_const 10).
Compute (run_instruction
           (push_const 10)
           Env
           nil).
Compute (run_instruction
           (push_var "x")
           Env
           nil).
Compute (run_instruction
           (push_var "a")
           Env
           nil).
Compute (run_instruction
           add
```

```
                    Env
                    (5 :: 6 :: 1 :: 4 :: nil)
                  ).
Compute (run_instruction
                    mul
                    Env
                    (5 :: 6 :: 1 :: 4 :: nil)
                  ).
Compute (run_instruction
                    add
                    Env
                    (4 :: nil)
                  ).
Compute (run_instruction
                    add
                    Env
                    nil
                  ).
```

The run_instruction function executes only one instruction at a time. A program for our stack machine is a list of instructions, so we define function that can execute programs below:

```
Fixpoint run_instructions
          (is' : list Instruction)
          (env : string → nat)
          (stack : list nat) :=
  match is' with
  | nil ⇒ stack
  | i :: is'' ⇒ run_instructions
                  is''
                  env
                  (run_instruction i env stack)
  end.
```

This new function allows us to execute a list of instructions: when the list of instructions is empty, the stack remains unchanged; if the list is not empty, then execute the first instruction to get a new stack, and then execute recursively the rest of the instructions on the new stack. The function terminates when the list of instructions is empty, that is, all instructions have been executed.

Exercise 8.2.2 *Test the run_instructions function by executing the commands below:*

```
Definition pgm1 := [(push_const 5); (push_var "a") ; add].
Definition pgm2 :=
```

```
  [ (push_const 10) ; (push_var "a") ; (push_var "b") ; add ; mul].
Compute run_instructions pgm1 Env nil.
Compute run_instructions pgm2 Env nil.
```

8.3 Certified compilation

We are now ready to define a compiler. This compiler will translate expressions into programs for the stack machine. The goal is to obtain a certified compiler.

First, we define our compiler as a function which does the translation:

```
Fixpoint compile (e : Exp) : list Instruction :=
  match e with
  | anum c ⇒ [push_const c]
  | avar x ⇒ [push_var x]
  | aplus e1 e2 ⇒ (compile e1) ++ (compile e2) ++ (add :: nil)
  | amul e1 e2 ⇒ (compile e1) ++ (compile e2) ++ (mul :: nil)
  end.
```

This function compiles expressions into programs that can be executed using our the stack machine. The compilation of expressions that are numbers and variables generates singleton lists that contain the corresponding push instruction. Addition and multiplication are a bit more complex since their compilation requires recursive compilation of the operands. Once these are compiled, the resulting pieces of code are concatenated into the final program. Note that add and mul are added at the end of the generated compiled program. This is because the execution is based on stack and the addition (or multiplication) will be made after the operands are evaluated and pushed to the stack.

Here is an example of what our compilation produces:

```
Compute compile ("a" +' 4 *' "a").
     [push_var "a"; push_const 4; push_var "a"; mul; add]
     : list Instruction.
```

The execution of the generated program seems to produce the expected result. Here is the execution of the initial program (recall that in Env the variable “a” has value 10):

```
Compute interpret ("a" +' 4 *' "a") Env.
     50
     : nat.
```

Here is the confirmation that the compiled code returns the expected stack:

```
Compute
    run_instructions
    (compile ("a" +' 4 *' "a"))
    Env
```

```
nil.
  [50]
  : list nat.
```

Based on the above example, it looks like our compiler does the job well: the compiled program produces the same result as the code generated by our compiler.

Our goal is to build a certified compiler, that is, a compiler that is guaranteed to produce code that preserves the behaviour of the initial program. For our simple compiler we can formulate a theorem that captures the fact that both programs (the one to be compiled and the corresponding code generated by the compiler) produce the same results. Since our compiler is a recursive function, we need to prove first an invariant property, which states that partially compiled programs produce the same results as their corresponding compiler generated code:

```
Lemma soundness_helper :
  ∀ e env stack is',
    run_instructions (compile e ++ is') env stack =
    run_instructions is' env ((interpret e env) :: stack).
Proof.
  induction e; intros; simpl; trivial.
  - rewrite ← app_assoc.
    rewrite ← app_assoc.
    rewrite IHe1.
    rewrite IHe2.
    simpl.
    rewrite PeanoNat.Nat.add_comm.
    reflexivity.
  - rewrite ← app_assoc.
    rewrite ← app_assoc.
    rewrite IHe1.
    rewrite IHe2.
    simpl.
    rewrite PeanoNat.Nat.mul_comm.
    reflexivity.
Qed.
```

This invariant is essential for proving the next theorem:

```
Theorem soundness :
  ∀ e env,
    run_instructions (compile e) env nil =
    [interpret e env].
Proof.
  intros.
  rewrite ← app_nil_r with (l := compile e).
  rewrite soundness_helper. simpl.
```

```
  trivial.
Qed.
```

The soundness theorem states that executing the code generated by compiling an expression produces the same result as executing the expression. The proof uses the invariant lemma soundness_helper which is essentially a generalisation of the soundness theorem.

Conclusions

Let us draw some conclusions on what we did. First, we created an interpreter for our expressions language. This interpreter is the intended formal semantics of our language and can be used as a reference implementation for the language. Second, we defined a stack machine which is totally unrelated to our expressions interpreter. Third, we implemented a compiler which is able to translate expressions into lists of instructions that can be executed using our stack machine. In a real-world implementation, the generated machine code should be faster than the interpreter. Finally, we proved that the code generated by our compiler produces the same results as the reference interpreter for any expression. Therefore, we developed a certified compiler for our language. This is not very common for the current main stream compilers (e.g., `gcc`, `javac`, and others): these compilers are very well-tested, but not certified. Only a few compilers can actually provide formal guarantees that they are indeed correct (e.g., CompCert).

8.4 Exercises

Exercise 8.4.1 *Create an interpreter and a compiler for a boolean expressions language. The language should include at least the following expressions: the boolean constants* true *and* false*, negation, conjunction and disjunction. For compilation you can use the same stack machine (that we used in this chapter) or an improved one, depending on your needs. In the end, formulate and prove properties that increase the confidence in your compiler (i.e., soundness).*

Chapter 9

Untyped lambda calculus

9.1 Functional programming

Functional programming is a programming paradigm where programs are made out of functions which can be composed to obtain the desired behavior. Instead of using a sequence of statements (like in imperative programming) which alter the memory (typically using assignments), in functional programming functions are treated as *first-class citizens*. This allows functions to be bound to a particular name, passed as arguments to other functions, or returned from other functions.

The most important trait of functional programming is that, in its pure implementations, it does not allow side-effects which definitely improves the quality of the code (i.e., less bugs) and in the same time it makes programs more amenable for program verification.

How is functional programming different from the other paradigms? All the other programming paradigms (imperative, object-oriented) share the same computational model: modify values stored in (memory) locations. This model is known as the *von Neumann machine* and it is the model that originates the modern computer. On the other hand, this is not the only model that can be used for computation. Functional programming does not refer to the concept of state. Instead, it uses *rewriting*: changes happen by transforming expressions. Therefore, in functional programming there is no state, which implies that there is no assignment. Moreover, the classical loops which modify the state as long as a guard is satisfied, also losses its real sense. In functional programming, recursion becomes one of the most important ingredients for sequence control.

Functional programming has its roots in *lambda calculus*, a very simple but powerful language.

9.2 Untyped lambda (λ) calculus

Lambda calculus (a.k.a. λ-calculus) was introduced in 1930s by Alonzo Church. The first version of λ-calculus was shown inconsistent by Stephen Kleene and J. B. Rosser (check the Kleene-Rosser paradox). However, in 1936, Church publishes an untyped version of λ-calculus, which is relevant computationally. A typed version appeared in 1940s, and it is shown to be logically consistent.

Functions that can be *computable* are extremely important in computer science. Lambda calculus does provide a simple semantics for computation. This enables a formal approach when studying properties of computations. In 1935, Church argued that any computable function on the natural numbers can be computed using lambda calculus. In 1936, Turing developed what is now called the *Turing Machine* and he showed that the two models (λ-calculus and the Turing Machine) are equivalent[1]. This is one of the most important ideas on computer science because it unifies two fields that developed somehow independently: on the one side we have algorithms and complexity (fields developed from the Turing Machine) and on the other side we have programming language theory (field developed from the λ-calculus).

9.2.1 λ-terms

Lambda calculus has a very simple syntax. Given a possibly infinite set of variables X, the BNF syntax of the λ-*terms* t is defined as:

$$
\begin{array}{lcll}
t & ::= & x & // \text{ where } x \in X \\
 & | & \lambda x.t & // \text{ where } x \in X \text{ and } t \text{ is a } \lambda\text{-term} \\
 & | & t\ s & // \text{ where } t \text{ and } s \text{ are } \lambda\text{-terms}
\end{array}
$$

Abstractions The construction $\lambda x.t$ is called *abstraction* and it is a definition of an anonymous function. For example, if in mathematics we define

$$f(x) = x^2 + 2$$

then in λ-calculus this function can be encoded as an abstraction:

$$\lambda x.x^2 + 2.$$

Note that the function $f(x) = x^2+2$ takes an input, namely x, and then it performs some calculation that involves x (i.e., compute the square of x and adds 2). The abstraction $\lambda x.x^2 + 2$ does the same thing, except that we do not indicate a name for the function. In lambda calculus, the abstraction operator in the term $\lambda x.t$ *binds* x, that is, in t, x is seen as an input. In $\lambda x.x^2 + 2$, the input is x and t is $x^2 + 2$.

[1] This is know as the Church-Turing thesis.

Not surprisingly, $\lambda y.y^2 + 2$ denotes the *exact* same function: it takes an input and performs the *same* calculation that involves that input. Later on we will see that these two terms ($\lambda x.x^2 + 2$ and $\lambda y.y^2 + 2$) are called *α-equivalent*.

Application The construction $t\ s$, where t and s are λ-terms, is called *application* and it represents the application of the function t to the function s.

Here are some examples of λ-terms:

- x - is a λ-term because it is a variable;
- $\lambda x.x$ - it is an abstraction which represents the identity function;
- $\lambda x.y$ - a function that always returns y, no matter what is the input x;
- $\lambda x.(\lambda y.x)$ - a function which takes two arguments and returns the first;
- $(\lambda x.x)\ y$ - an application of the identity function $(\lambda x.x)$ to an argument y; later on this chapter we will see that this will evaluate to y, as expected.

Notational conventions In the BNF syntax of λ-terms there are no explicit parentheses. However, we used parentheses in the examples shown above (e.g., $\lambda x.(\lambda y.x)$, $(\lambda x.x)\ y$) to avoid ambiguities.

To keep things as simple as possible, we will always use parentheses when needed to avoid ambiguities. For example, without parentheses it is hard to say whether the λ-term

$$\lambda x.x\ \lambda x.y$$

represents

$$(\lambda x.x)\ (\lambda x.y)$$

or

$$\lambda x.(x\ (\lambda x.y)).$$

For newcomers, it is highly recommended to use explicit parentheses when needed.

9.2.2 Free and bound variables.

The set of *free* variables is computed using the function $\mathit{free} : t \to 2^X$ defined below:

- $\mathit{free}(x) = \{x\}$;
- $\mathit{free}(\lambda x.t) = \mathit{free}(t) \setminus \{x\}$;
- $\mathit{free}(t\ s) = \mathit{free}(t) \cup \mathit{free}(s)$.

The free variables of a term are the ones that are not captured by a binder. As an example, here is the set of free variables of $(\lambda x.x)\ (\lambda x.y)$:

$$\begin{aligned}
\mathit{free}\big((\lambda x.x)\ (\lambda x.y)\big) &= \mathit{free}\big((\lambda x.x)\big) \cup \mathit{free}\big((\lambda y.y)\big) \\
&= \Big(\mathit{free}\big(x\big) \setminus \{x\}\Big) \cup \Big(\mathit{free}\big(y\big) \setminus \{x\}\Big) \\
&= \big(\{x\} \setminus \{x\}\big) \cup \big(\{y\} \setminus \{x\}\big) \\
&= \emptyset \cup \{y\} \\
&= \{y\}.
\end{aligned}$$

Note that there is no binder for y in our example, so $\mathit{free}\big((\lambda x.x)\ (\lambda x.y)\big) = \{y\}$.

Conversely, the set of *bound* variables is computed using the $\mathit{bound} : t \to 2^X$ function:

- $\mathit{bound}(x) = \emptyset$;
- $\mathit{bound}(\lambda x.t) = \mathit{bound}(t) \cup \{x\}$;
- $\mathit{bound}(t\ s) = \mathit{bound}(t) \cup \mathit{bound}(s)$.

We recall the previous example $(\lambda x.x)\ (\lambda x.y)$, and we compute the set of bound variables for this λ-term:

$$\begin{aligned}
\mathit{bound}\big((\lambda x.x)\ (\lambda x.y)\big) &= \mathit{bound}\big((\lambda x.x)\big) \cup \mathit{bound}\big((\lambda x.y)\big) \\
&= \Big(\mathit{bound}\big(x\big) \cup \{x\}\Big) \cup \Big(\mathit{bound}\big(y\big) \cup \{x\}\Big) \\
&= \big(\emptyset \cup \{x\}\big) \cup \big(\emptyset \cup \{x\}\big) \\
&= \{x\} \cup \{x\} \\
&= \{x\}.
\end{aligned}$$

Since there is a binder for x in both $\lambda x.x$ and $\lambda x.y$, x is an element in the set of bound variables.

9.2.3 Capturing substitution

We now define capturing (or non-capture avoiding) substitution. The notation $t[t'/x]$ stands for the term obtained from t by substituting t' for all free occurrences of x. The formal definition is below:

1. $x[t'/x] = t'$;

2. $y[t'/x] = y$, if $y \neq x$;

3. $(\lambda x.t)[t'/x] = \lambda x.t$;

4. $(\lambda y.t)[t'/x] = \lambda y.(t[t'/x])$, if $y \neq x$;

5. $(t\ s)[t'/x] = (t[t'/x])\ (s[t'/x])$.

Note that $t[t'/x]$ simply replaces in t the free occurrences of x with t'. This is ensured by the third and fourth cases in the above definition: $(\lambda x.t)[t'/x]$ results in $\lambda x.t$ because x is bound in t, while $(\lambda y.t)[t'/x] = \lambda y.(t[t'/x])$ when $y \neq x$, that is, x is free in t. Let us take a look at the following example:

$$\begin{aligned} \big((\lambda x.x)\ (\lambda x.y)\big)[z/y] = & \quad \big((\lambda x.x)[z/y]\big)\ \big((\lambda x.y)[z/y]\big) \\ = & \quad \big(\lambda x.(x[z/y])\big)\ \big(\lambda x.(y[z/y])\big) \\ = & \quad \big(\lambda x.x\big)\ \big(\lambda x.z\big) \end{aligned}$$

It is worth noting that $\big(\lambda x.(x[z/y])\big)$ is indeed $\big(\lambda x.x\big)$ because $x[z/y] = x$, while $\big(\lambda x.(y[z/y])\big)$ is equal by definition to $\big(\lambda x.z\big)$ because $y[z/y]$.

However, there is something odd about the capturing substitution. For instance, $(\lambda y.x)[y/x]$ is (by definition) equal to $\lambda y.(x[y/x])$, which in turn, equals to $\lambda y.y$:

$$\begin{aligned} (\lambda y.x)[z/y] = & \quad \lambda y.(x[y/x]) \\ = & \quad \lambda y.y \end{aligned}$$

So, we started with a term which ignores its input, that is $(\lambda y.x)$ and by applying a substitution we obtain a term which does not ignore its input anymore, namely, $\lambda y.y$. Basically, we replace x with y, but in $(\lambda y.x)$, y is not free! The y which replaces x is *captured* by the binder λy, and thus, we obtain a completely different term. The trouble is that substitutions are syntactical operations, and they should not change the behaviour of λ-terms.

A possible fix to this problem is to use an α-equivalent version of $(\lambda y.x)$, say $(\lambda y'.x)$. Now, $(\lambda y'.x)[y/x] = \lambda y'.(x[y/x]) = \lambda y'.y$. We obtained a similar λ-term which ignores the input. Section 9.2.4 provides the appropriate definition for capture-avoiding substitutions.

9.2.4 Capture-avoiding substitution

We are now ready to define a capture-avoiding substitution, which fixes the problem described above. We denote by $t[\![t'/x]\!]$ the substitution that avoids capturing variables:

1. $x[\![t'/x]\!] = t'$;
2. $y[\![t'/x]\!] = y$, if $y \neq x$;
3. $(\lambda x.t)[\![t'/x]\!] = \lambda x.t$;
4. $(\lambda y.t)[\![t'/x]\!] = \lambda y'.\big((t[y'/y])[\![t'/x]\!]\big)$, if $y \neq x$ and y' is a fresh variable;
5. $(t\ s)[\![t'/x]\!] = (t[\![t'/x]\!])\ (s[\![t'/x]\!])$.

The major difference between the definition of capture-avoiding substitution and the capturing substitution is in the fourth case: the capture-avoiding substitution performs a renaming of the bound variable y with a *fresh* variable y'. Here, *fresh* means that y' does not occur anywhere else in the involved terms. If we go back to our tricky example we can observe the differences:

- Capturing substitution:

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

- Capture-avoiding substitution:

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

Here is yet another example, where the substitution with a fresh variable makes more sense:

- Capturing substitution:

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

- Capture-avoiding substitution:

$$\begin{aligned}
\big(\lambda y.(x\ y)\big)[\![y/x]\!] &= \lambda y'.\Big(\big((x\ y)[y'/y]\big)[\![y/x]\!]\Big) \\
&= \lambda y'.\Big(\big((x[y'/y])\ (y[y'/y])\big)[\![y/x]\!]\Big) \\
&= \lambda y'.\big((x\ y')[\![y/x]\!]\big) \\
&= \lambda y'.\big((x[\![y/x]\!])\ (y'[\![y/x]\!])\big) \\
&= \lambda y'.(y\ y').
\end{aligned}$$

Note that the obtained term $\lambda y'.(y\ y')$ contains the fresh variable y' which was introduced by the capture-avoiding substitution.

9.2.5 Alpha equivalence

As you may have noticed, the identity function can be represented in many ways as an abstraction. For instance, $\lambda x.x$ is one possibility, $\lambda y.y$ is another possibility. Both λ-terms represent the identity function: they take an input and return that input. We say that these two λ-terms, $\lambda x.x$ and $\lambda y.y$, are *alpha equivalent* or *α-equivalent*.

In general, in an abstraction $\lambda x.t$, if the bound variable x is replaced everywhere by a variable y (that does not occur in t) we obtain an α-equivalent form $\lambda y.t[y/x]$ of $\lambda x.t$. Typically, we use the notation $t_1 \equiv_\alpha t_2$ to denote the fact that t_1 and t_2 are α-equivalent.

Here are some examples of α-equivalences:

1. $\lambda x.x \equiv_\alpha \lambda y.y$;
2. $(\lambda x.x)\ (\lambda y.y) \equiv_\alpha (\lambda z.z)\ (\lambda y.y)$;
3. $(\lambda x.x)\ (\lambda y.y) \equiv_\alpha (\lambda z.z)\ (\lambda z.z)$;
4. $(\lambda x.x)\ (\lambda y.y) \equiv_\alpha (\lambda z.z)\ (\lambda z'.z')$;
5. $(\lambda x.x)\ x \equiv_\alpha (\lambda y.y)\ x$;

9.2.6 Beta-reduction

At the beginning of this chapter we mentioned that in λ-calculus we can express computations. In fact, in λ-calculus there is only one computation rule, called β-reduction:

$$(\lambda x.t)\ t' \to_\beta t[\![t'/x]\!].$$

The β-reduction rule is the rule that applies an abstraction (a function) to an argument. The expression $(\lambda x.t)\ t'$ is called *redex* (i.e., the place where the β-reduction happens) and $t[\![t'/x]\!]$ is called *reductum* (i.e., the reduced expression). We discuss below some examples.

First, we discuss the identity function that we have seen before. Normally, when applied to an argument it returns, as expected, the given argument:

$$(\lambda x.x)\ y \to_\beta x[\![y/x]\!] = y.$$

Here, $(\lambda x.x)\ y$ is the redex, while y is the reductum. Also, note that $\to_\beta$ represents a β-reduction step, while the result of the reduction $x[\![y/x]\!]$ is then computed using the definition of the capture-avoiding substitution.

Function application can be easily expressed as $\lambda f.\big(\lambda x.(f\ x)\big)$. When applied to a function f and an argument x, $\lambda f.\big(\lambda x.(f\ x)\big)$ should apply f to x. Here is an example:

$$
\begin{aligned}
\big(\lambda f.(\lambda x.(f\ x))\big)\ (\lambda x.x)\ z \to_\beta\ & \big((\lambda x.(f\ x))[\![(\lambda x.x)/f]\!]\big)\ z \\
=\ & \big(\lambda x'.(((f\ x)[x'/x])[\![(\lambda x.x)/f]\!])\big)\ z \\
=\ & \big(\lambda x'.(((f[x'/x])\ (x[x'/x]))[\![(\lambda x.x)/f]\!])\big)\ z \\
=\ & \big(\lambda x'.((f\ x')[\![(\lambda x.x)/f]\!])\big)\ z \\
=\ & (\lambda x'.(f[\![(\lambda x.x)/f]\!]\ x'[\![(\lambda x.x)/f]\!]))\ z \\
=\ & \big(\lambda x'.((\lambda x.x)\ x')\big)\ z \\
\to_\beta\ & ((\lambda x.x)\ x')[\![z/x']\!] \\
=\ & ((\lambda x.x)[\![z/x']\!])\ (x'[\![z/x']\!]) \\
=\ & (\lambda x''.(x''[x''/x])[\![z/x']\!])\ z \\
=\ & (\lambda x''.x''[\![z/x']\!])\ z \\
=\ & (\lambda x''.x'')\ z \\
\to_\beta\ & x''[\![z/x'']\!] \\
=\ & z.
\end{aligned}
$$

Note that the first input f of $\lambda f.(\lambda x.(f\ x))$ is applied to the second input x. That is, $\lambda f.(\lambda x.(f\ x))$ can take a function as the first parameter (e.g., $\lambda x.x$ as above). In fact, in lambda calculus, passing functions to other functions and returning functions is very natural. This is why the programming languages based on lambda calculus (e.g., functional languages like OCaml, Haskell, Coq, etc.) can handle functions very easily. In contrast, in other programming languages passing functions as inputs or returning functions require a lot of complicated code (e.g., pointers to functions in C). Recently, popular languages (C++, Java, C#) have been enriched with lambda-functions which is regarded as a very cool feature that allows them to write lambda terms.

Below is another example of the same function $\lambda f.(\lambda x.(f\ x))$ applied to two functions. Note that the resulted term is a term which represents a function too:

$$
\begin{aligned}
\Big(\lambda f.\big(\lambda x.(f\ x)\big)\Big)\ (\lambda x.x)\ (\lambda y.y) \to_\beta &\quad \Big(\big(\lambda x.(f\ x)\big)[\![(\lambda x.x)/f]\!]\Big)\ (\lambda y.y)\\
= &\quad \Big(\lambda x'.\big(\big((f\ x)[x'/x]\big)[\![(\lambda x.x)/f]\!]\big)\Big)\ (\lambda y.y)\\
= &\quad \Big(\lambda x'.\big(\big((f[x'/x])\ (x[x'/x])\big)[\![(\lambda x.x)/f]\!]\big)\Big)\ (\lambda y.y)\\
= &\quad \Big(\lambda x'.\big((f\ x')[\![(\lambda x.x)/f]\!]\big)\Big)\ (\lambda y.y)\\
= &\quad (\lambda x'.(f[\![(\lambda x.x)/f]\!]\ x'[\![(\lambda x.x)/f]\!]))\ (\lambda y.y)\\
= &\quad \Big(\lambda x'.\big((\lambda x.x)\ x'\big)\Big)\ (\lambda y.y)\\
\to_\beta &\quad \big((\lambda x.x)\ x'\big)[\![(\lambda y.y)/x']\!]\\
= &\quad \big((\lambda x.x)[\![(\lambda y.y)/x']\!]\big)\ \big(x'[\![(\lambda y.y)/x']\!]\big)\\
= &\quad \big(\lambda x''.(x''[x''/x])[\![(\lambda y.y)/x']\!]\big)\ (\lambda y.y)\\
= &\quad \big(\lambda x''.x''[\![(\lambda y.y)/x']\!]\big)\ (\lambda y.y)\\
= &\quad (\lambda x''.x'')\ (\lambda y.y)\\
\to_\beta &\quad x''[\![(\lambda y.y)/x'']\!]\\
= &\quad (\lambda y.y).
\end{aligned}
$$

The inputs of $\lambda f.\big(\lambda x.(f\ x)\big)$ are the identity functions $\lambda x.x$ and $\lambda y.y$, while the output is $\lambda y.y$, that is, a function.

Convention The above examples are very detailed, especially when computing substitutions. Due to the high level of detail, it becomes harder to follow such long examples. As a convention, we will often omit the part where substitutions are computed and we will focus only on the β-reductions. As always, there is a catch: do not forget to rename the bound variables when needed! Here is the same example as above, written using our new convention:

$$
\begin{aligned}
\Big(\lambda f.\big(\lambda x.(f\ x)\big)\Big)\ (\lambda x.x)\ (\lambda y.y) \to_\beta &\quad \Big(\lambda x'.\big((\lambda x.x)\ x'\big)\Big)\ (\lambda y.y)\\
\to_\beta &\quad (\lambda x''.x'')\ (\lambda y.y)\\
\to_\beta &\quad (\lambda y.y).
\end{aligned}
$$

It is now easier to see that f has been replaced by $\lambda x.x$. Also, note that in the second step, x in $\lambda x.x$ was replaced by x'' - this is the definition of the capture-avoiding substitution. However, for this particular case, this renaming is not necessary because the variables in $\lambda y.y$ do not interact with the variables in $\lambda x.x$. We can actually rewrite the example as follows:

$$
\begin{aligned}
\Big(\lambda f.\big(\lambda x.(f\ x)\big)\Big)\ (\lambda x.x)\ (\lambda y.y) \to_\beta &\quad \Big(\lambda x'.\big((\lambda x.x)\ x'\big)\Big)\ (\lambda y.y)\\
\to_\beta &\quad (\lambda x.x)\ (\lambda y.y)\\
\to_\beta &\quad (\lambda y.y).
\end{aligned}
$$

9.2.7 Normal forms

When a λ-term t cannot be reduced anymore, that is, the β-reduction rule cannot be applied, we say that t is in *normal form.* For instance, $\lambda x.(\lambda y.x)$ is in normal form, while $\lambda x.\big((\lambda y.x)\ z\big)$ is not in normal form. This term can be further reduced:

$$\lambda x.\big((\lambda y.x)\ z\big) \rightarrow_\beta \lambda x.\big(x[\![z/y]\!]\big) = \lambda x.x.$$

Now, $\lambda x.x$ is a normal form for $\lambda x.\big((\lambda y.x)\ z\big)$.

Not all lambda terms have a normal form. For example, the sequence

$$\big(\lambda x.(x\ x)\big)\ \big(\lambda x.(x\ x)\big) \rightarrow_\beta \big(\lambda x.(x\ x)\big)\ \big(\lambda x.(x\ x)\big) \rightarrow_\beta \cdots$$

is infinite.

Exercise 9.2.1 *Let Y be a shorthand for $\lambda f.\Big(\big(\lambda x.\big(f\ (x\ x)\big)\big)\ \lambda x.\big(\big(f\ (x\ x)\big)\big)\Big)$. What is the normal form of $\big(Y\ (\lambda x.x)\big)$?*

9.2.8 Evaluation strategies

It is often the case that the β-reduction rule can be applied in several locatios under a lambda term. For instance, when computing the normal form of $(\lambda x.x)\ \big((\lambda y.y)\ z\big)$ we can have two possible ways of finding it:

1. $(\lambda x.x)\ \big((\lambda y.y)\ z\big) \rightarrow_\beta (\lambda y.y)\ z \rightarrow z$, or
2. $(\lambda x.x)\ \big((\lambda y.y)\ z\big) \rightarrow_\beta (\lambda x.x)\ z \rightarrow z$.

This happens because $(\lambda x.x)\ \big((\lambda y.y)\ z\big)$ has more than one redex: we can either apply the reduction rule at $(\lambda x.x)\ \big(\dots\big)$ or we can apply it for $\big((\lambda y.y)\ z\big)$. In both cases we obtain the same result. Since we obtain the same result, why should we care about how β-reductions are applied? It turns out that the evaluation strategies that are used to apply the β-reduction rule may impact the performance. We discuss different evaluation strategies below.

Full-beta reduction

This strategy says that any redex can be reduced at any time. This strategy is more like a *no strategy* to apply β-reductions. Any of the previous reductions

1. $(\lambda x.x)\ \big((\lambda y.y)\ z\big) \rightarrow_\beta (\lambda y.y)\ z \rightarrow z$, or
2. $(\lambda x.x)\ \big((\lambda y.y)\ z\big) \rightarrow_\beta (\lambda x.x)\ z \rightarrow z$.

are allowed by the full-beta reduction strategy.

Exercise 9.2.2 *What are the possible successors of $(\lambda x.x)\ \Big((\lambda z.z)\ \big((\lambda y.y)\ x'\big)\Big)$ using the full beta-reduction strategy?*

Normal-order

This strategy says that the leftmost, outermost redex is reduced first. Back to our example, only

$$(\lambda x.x)\ \big((\lambda y.y)\ z\big) \to_\beta (\lambda y.y)\ z \to z$$

is allowed (which is different from full-beta reduction, see above). This strategy allows exactly one successor (up to α-equivalence) for any given lambda term. Also, normal-order is a *non-strict* evaluation strategy because it does not require that the arguments of an abstraction (function) are evaluated before β-reduction.

Exercise 9.2.3 *Find the normal form of* $(\lambda x.x)\ \Big((\lambda z.z)\ \big((\lambda y.y)\ x'\big)\Big)$ *using the normal-order strategy.*

Call-by-Name (CBN)

This strategy is a restricted version of the normal-order strategy where applications of the β-reduction rule inside a lambda abstraction is forbidden. Note the difference between this strategy and the previous strategy below:

- Normal-order:

$$\begin{aligned}(\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big) &\to_\beta\\ \lambda x.\big((\lambda y.y)\ x\big) &\to_\beta\\ \lambda x.x.\end{aligned}$$

  Here, the β-reduction rule inside a lambda abstraction is allowed.

- Call-by-name:

$$\begin{aligned}(\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big) &\to_\beta\\ \lambda x.\big((\lambda y.y)\ x\big) &\not\to_\beta .\end{aligned}$$

  Here, the β-reduction rule inside a lambda abstraction is forbidden. The $\not\to_\beta$ suggests that $\big((\lambda y.y)\ z\big)$ cannot be reduced using the call-by-name strategy.

The Call-by-Name strategy has advantages and disadvantages. Sometimes, the evaluation can be drastically shortened because it is a non-strict strategy (as normal-order). For instance,

$$\big(\lambda x.(\lambda y.y)\big)\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) \to_\beta \lambda y.y,$$

It is easy to notice that CBN has a clear advantage here: $\lambda y.y$ is the only successor here, and there is no need to evaluate the argument $\Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big)$.

On the other hand, CBN can create more complexity than needed. In the example below, the argument needs to evaluated twice:

$$
\begin{aligned}
(\lambda x.(x\ x))\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) &\rightarrow_\beta \\
\Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big)\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) &\rightarrow_\beta \\
\Big(\lambda x.\big((\lambda y.y)\ x\big)\Big)\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) &\rightarrow_\beta \\
(\lambda y.y)\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) &\rightarrow_\beta \\
(\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big) &\rightarrow_\beta \\
\lambda x.\big((\lambda y.y)\ x\big) &\not\rightarrow_\beta .
\end{aligned}
$$

Because in $\lambda x.(x\ x)$ the input x is used twice, whatever input is passed to this function will be evaluated twice. In order to avoid such issues, a slight variation of CBN called *call-by-need*, evaluates each term only once and then reuses the previously computed results.

Call-by-value (CBV)

The *call-by-value* strategy is probably the most common among the mainstream languages. The idea behind CBV is simple: the arguments of a function are completely evaluated to *values* first, and then the function is called on those values. In lambda calculus, CBV means to reduce the leftmost, innermost redex which is not inside a lambda abstraction. The only values are lambda abstractions.

Here is an example that highlights the difference between CBN and CBV:

- Call-by-name:

$$
\begin{aligned}
(\lambda f.f)\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) &\rightarrow_\beta \\
(\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big) &\rightarrow_\beta \\
\lambda x.\big((\lambda y.y)\ x\big) &\not\rightarrow_\beta .
\end{aligned}
$$

  Recall that the β-reduction rule inside a lambda abstraction is forbidden. Moreover, note that the arguments are not evaluated before applying functions.

- Call-by-value:

$$\begin{aligned}(\lambda f.f)\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) &\to_\beta \\ (\lambda f.f)\ \Big(\lambda x.\big((\lambda y.y)\ x\big)\Big) &\to_\beta \\ \lambda x.\big((\lambda y.y)\ x\big) &\not\to_\beta .\end{aligned}$$

  In constrast, for CBV we first evaluate the argument $\Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big)$ and then we apply $\lambda f.f$. For this argument, we apply CBV as well: first we evaluate $\lambda x.\big((\lambda y.y)\ x\big)$ - which is already a value and then we apply $(\lambda z.z)$ to it. Now that we obtained the result $\lambda x.\big((\lambda y.y)\ x\big)$ when evaluating $\Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big)$, the application of $\lambda f.f$ to this result produces $\lambda x.\big((\lambda y.y)\ x\big)$.

In some cases CBV performs a smaller number of steps than CBN.

$$\begin{aligned}\big(\lambda x.(x\ x)\big)\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) &\to_\beta \\ \big(\lambda x.(x\ x)\big)\ \Big(\lambda x.\big((\lambda y.y)\ x\big)\Big) &\to_\beta \\ \lambda x.\big((\lambda y.y)\ x\big) &\not\to_\beta .\end{aligned}$$

But in the same time, CBV can determine useless evaluations. In the following example, CBN terminates in just a single step, while for CBV we have some additional (unnecessary) steps:

$$\begin{aligned}\big(\lambda x.(\lambda y.y)\big)\ \Big((\lambda z.z)\ \lambda x.\big((\lambda y.y)\ x\big)\Big) &\to_\beta \\ \big(\lambda x.(\lambda y.y)\big)\ \Big(\lambda x.\big((\lambda y.y)\ x\big)\Big) &\to_\beta \\ (\lambda y.y).&\end{aligned}$$

Other strategies

We have discussed only a few strategies here (full beta-reduction, normal-order, call-by-name (+ call-by-need), call-by-value), but these are not the only evaluation strategies. We chose to discuss only these because they are the most common ones. In the literature you will find that there are other strategies for lambda calculus like *applicative-order*, *optimal reduction*, and *lazy evaluation*.

Bibliography

[1] Team Coq. Coq reference manual.

[2] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[3] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 2nd edition, 2016.

[4] Tobias Nipkow. Teaching semantics with a proof assistant: No more lsd trip proofs. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 24–38, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[5] Benjamin C. Pierce. Proof assistants as teaching assistants: A view from the trenches. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 8–8, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018. Version 5.5.

[7] Benjamin C. Pierce, Arthur Azevedo de Amorim 0;136;0c, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. E0;136;0clectronic textbook, May 2018.

[8] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.