

Introduction to Functional Programming and the Structure of Programming Languages using OCaml

Lecture Notes
Version of April 5, 2022

Gert Smolka
Saarland University

Copyright © 2021 by Gert Smolka, all rights reserved

Contents

Preface	vii
1 Getting Started	1
1.1 Programs and Declarations	1
1.2 Functions and Let Expressions	3
1.3 Conditionals, Comparisons, and Booleans	4
1.4 Recursive Power Function	5
1.5 Integer Division	6
1.6 Mathematical Level versus Coding Level	9
1.7 More about Mathematical Functions	11
1.8 Linear Search and Higher-Order Functions	14
1.9 Partial Applications	15
1.10 Inversion of Strictly Increasing Functions	17
1.11 Tail Recursion	18
1.12 Tuples	20
1.13 Exceptions and Spurious Arguments	21
1.14 Summary	23
2 Syntax and Semantics	25
2.1 Lexical Syntax	26
2.2 Phrasal Syntax	27
2.2.1 Types	28
2.2.2 Expressions	30
2.2.3 Declarations and Programs	33
2.3 Derivation Systems	34
2.4 Abstract Expressions	35
2.5 Static Semantics	36
2.6 Type Checking Algorithm	40
2.7 Free and Local Variables	42
2.8 Dynamic Semantics	43
2.9 Derived Forms	47
2.10 Summary and Discussion	49
3 Polymorphic Functions and Iteration	53
3.1 Polymorphic Functions	53
3.2 Iteration	54
3.3 Iteration on Pairs	55

Contents

3.4	Computing Primes	57
3.5	Polymorphic Typing Rules	59
3.6	Polymorphic Exception Raising and Equality Testing	61
3.7	Summary	62
4	Lists	64
4.1	Nil and Cons	64
4.2	Basic List Functions	65
4.3	List Functions in OCaml	68
4.4	Fine Points About Lists	70
4.5	Membership and List Quantification	71
4.6	Head and Tail	72
4.7	Position Lookup	73
4.8	Option Types	74
4.9	Generalized Match Expressions	75
4.10	Sublists	77
4.11	Folding Lists	78
4.12	Insertion Sort	80
4.13	Generalized Insertion Sort	82
4.14	Lexical Order	83
4.15	Prime Factorization	84
4.16	Key-Value Maps	86
5	Constructor Types, Trees, and Linearization	89
5.1	Constructor Types	89
5.2	AB-Trees	90
5.3	Prefix, Postfix, and Infix Linearization	92
5.4	Infix Linearization with Dropped Parentheses	94
5.5	ABC-Trees	96
5.6	Mini-OCaml	97
5.7	Natural Numbers as Constructor Type	100
5.8	Summary	101
6	Parsing	102
6.1	Lexing	102
6.2	Recursive Descent Parsing	105
6.3	Infix with Associativity	106
6.4	Infix with Precedence and Juxtaposition	108
6.5	Postfix Parsing	109
6.6	Mini-OCaml	110

Contents

7	Mini-OCaml Interpreter	113
7.1	Expressions, Types, Environments	114
7.2	Type Checker	115
7.3	Evaluator	115
7.4	Lexer	116
7.5	Parser	118
7.6	The Project	119
8	Running Time	121
8.1	The Idea	121
8.2	List Reversal	123
8.3	Insertion Sort	124
8.4	Merge Sort	125
8.5	Binary Search	126
8.6	Trees	128
8.7	Summary and Background	130
8.8	Call Depth	131
9	Inductive Correctness Proofs	133
9.1	Propositions and Proofs	133
9.2	List induction	134
9.3	Properties of List Reversal	136
9.4	Natural Number Induction	138
9.5	Correctness of Tail-Recursive Formulations	140
9.6	Properties of Iteration	143
9.7	Induction for Terminating Functions	144
9.8	Euclid’s Algorithm	145
9.9	Complete Induction	148
9.10	Prime Factorization Revisited	151
10	Arrays	155
10.1	Basic Array Operations	155
10.2	Conversion between Arrays and Lists	157
10.3	Binary Search	158
10.4	Reversal	160
10.5	Sorting	161
10.6	Equality for Arrays	164
10.7	Execution Order	164
10.8	Cells	165
10.9	Mutable Objects Mathematically	167

Contents

11	Data Structures	168
11.1	Structures and Signatures	168
11.2	Stacks and Queues	169
11.3	Array-Based Stacks	171
11.4	Circular Queues	173
11.5	Block Representation of Lists in Heaps	174
11.6	Realization of a Heap	175
11.7	Block Representation of AB-Trees	177
11.8	Structure Sharing and Physical Equality	179
12	Appendix: Example Exams	181

Preface

This text teaches functional programming and the structure of programming languages to beginning students. It is written for the Programming 1 course for computer science students at Saarland University. We assume that incoming students are familiar with mathematical thinking, but we do not assume programming experience. The course is designed to take about one third of the study time of the first semester.

We have been teaching a course like this at Saarland University since 1998. Students perceive the course as challenging and exciting, whether they have programmed before or not. In 2021, we changed the teaching language to English and the programming language to OCaml.

As it comes to functional programming, we cover higher-order recursive functions, polymorphic typing, and constructor types for lists, trees, and abstract syntax. We emphasize the role of correctness statements and practice inductive correctness proofs. We also cover asymptotic running time considering binary search (logarithmic), insertion sort (quadratic), merge sort (linearithmic), and other algorithms.

As it comes to the structure of programming languages, we study the different layers of syntax and semantics at the example of the idealized functional programming language Mini-OCaml. We describe the syntactic layers with grammars and the semantic layers with inference rules. Based on these formal descriptions, we program recursive descent parsers, type checkers and evaluators.

We also cover stateful programming with arrays and cells (assignable variables). We explain how lists and trees can be stored as linked blocks in an array, thus explaining memory consumption for constructor types.

There is a textbook¹ written for the German iterations of the course (1998 to 2020). The new English text realizes some substantial changes: OCaml rather than Standard ML as programming language, less details about the concrete programming language being used, more emphasis on correctness arguments and algorithms, informal type-theoretic explanations rather than formal set-theoretic definitions.

The current version of the text leaves room for improvement. More basic explanations with more examples could be helpful in many places. An additional chapter on imperative programming with loops and the

¹Gert Smolka, *Programmierung — eine Einführung in die Informatik mit Standard ML*. Oldenbourg Verlag, 2008.

Preface

realization with stack machines with jumps (see the German textbook) would be interesting, but this extra material may not fit into the time-budget of a one-semester course.

At Saarland University, the course spans 15 weeks of teaching in the winter semester. Each week comes with two lectures (90 minutes each), an exercises assignment, office hours, tutorials, and a test (15 minutes). There is a midterm exam in December and a final exam (offered twice) after the lecture period has finished. The 2021 iteration of the course also came with a take home project over the holiday break asking the students to write an interpreter for Mini-OCaml. The take home project should be considered an important part of the course, given that it requires the students writing and debugging a larger program (about 250 lines), which is quite different from the small programs (up to 10 lines) the weekly assignments ask for.

1 Getting Started

In this chapter we start programming in OCaml. We use an interactive tool called an interpreter that checks and executes the declarations of a program one by one. We concentrate on recursive functions on integers computing things like powers, integer quotients, digit sums, and integer roots. We also formulate a general algorithm known as linear search as a higher-order function. We follow an approach known as functional programming where functions are designed at a mathematical level using types and equations before they are coded in a concrete programming language.

1.1 Programs and Declarations

An OCaml **program** is a sequence of **declarations**, which are executed in the order they are written. Our first program

```
let a = 2 * 3 + 2
let b = 5 * a
```

consists of two declarations. The first declaration **binds** the **identifier** a to the integer 8, and the second declaration binds the identifier b to the integer 40. This is clear from our understanding of the arithmetic operations “+” and “*”.

To learn programming in OCaml, you want to use an interactive tool called an **interpreter**.¹ The user feeds the interpreter with text. The interpreter checks that the given text can be interpreted as a program formulated correctly according to the rules of the programming language. If this is the case, the interpreter determines a **type** for every identifier and every expression of the program. Our example program is formulated correctly and the declared identifiers a and b both receive the type **int** (for integer). After a program has been checked successfully, the interpreter will execute it. In our case, execution will bind the identifier a to the integer 8 and the identifier b to the integer 40. After the program has been executed successfully, the interpreter will show the values it has computed for the declared identifiers. If a program is not formulated correctly, the interpreter will show an error message indicating which rule of the language is violated. Once the interpreter

¹A nice browser-based interpreter is <https://try.ocamlpro.com>.

1 Getting Started

has checked and executed a program, the user can extend it with further declarations. This way one can write the declarations of a program one by one.

At this point you want to start working with the interpreter. You will learn the exact rules of the language through experiments with the interpreter guided by the examples and explanations given in this text.

Here is a program redeclaring the identifier *a*:

```
let a = 2 * 3 + 2
let b = 5 * a
let a = 5
let c = a + b
```

The second declaration of the identifier *a* **shadows** the first declaration of *a*. Shadowing does not affect the binding of *b* since it is obtained before the second declaration of *a* is executed. After execution of the program, the identifiers *a*, *b*, and *c* are bound to the integers 5, 40, and 45, respectively.

The declarations we consider in this chapter all start with the **keyword** *let* and consist of a **head** and a **body** separated by the equality symbol “=”. Keywords cannot be used as identifiers. The bodies of declarations are **expressions**. Expressions can be obtained with identifiers, constants, and operators. The **nesting of expressions** can be arranged with **parentheses**. For instance, the expression $2 \cdot 3 + 2 - x$ may be written with **redundant parentheses** as $((2 \cdot 3) + 2) - x$. The parentheses in $2 \cdot (x + y) - 3$ are not redundant and are needed to make the expression $x + y$ the right argument of the product with the left argument 2.

Every expression has a **type**. So far we have only seen expressions of the type *int*. The values of the type *int* are integers (whole numbers $\dots, -2, -1, 0, 1, -2, \dots$). In contrast to the infinite mathematical type \mathbb{Z} , OCaml’s type *int* provides only a finite interval of **machine integers** realized efficiently by the hardware of the underlying computer. The endpoints of the interval can be obtained with the predefined identifiers *min_int* and *max_int*. All machine operations respect this interval. We have $max_int + 1 = min_int$, for instance.²

When we reason about programs, we will usually ignore the machine integers and just assume that all integers are available. As long as the

²It turns out that different interpreters realize different intervals for machine integers, even on the same computer. For instance, on the author’s computer, in October 2021, the browser-based Try OCaml interpreter realizes *max_int* as $2^{31} - 1 = 2147483647$, while the official OCaml interpreter realizes *max_int* as $2^{61} - 1 = 4611686018427387903$.

1 Getting Started

numbers in a concrete computation are small enough, this simplification does not lead to wrong conclusions.

Every programming language provides machine integers for efficiency. There are techniques for realizing much larger intervals of integers based on machine integers in programming languages.

Exercise 1.1.1 Give an expression and a declaration. Explain the structure of a declaration. Explain nesting of expressions. Give a type.

Exercise 1.1.2 To what value does the program

```
let a = 2 let a = a * a let a = a * a
```

bind the identifier a ?

Exercise 1.1.3 Give a machine integer x such that $x + 1 < x$.

1.2 Functions and Let Expressions

Things become interesting once we declare functions. The declaration

```
let square x = x * x
```

declares a function

$$\text{square} : \text{int} \rightarrow \text{int}$$

squaring its argument. The identifier square receives a **functional type** $\text{int} \rightarrow \text{int}$ describing functions that given an integer return an integer. Given the declaration of square , execution of the declaration

```
let a = square 5
```

binds the identifier a to the integer 25.

Can we compute a power x^8 with just three multiplications? Easy, we just square the integer x three times:

```
let pow8 x = square (square (square x))
```

This declaration gives us a function $\text{pow8} : \text{int} \rightarrow \text{int}$.

Another possibility is the declaration

```
let pow8' x =  
  let a = x * x in  
  let b = a * a in  
  b * b
```

1 Getting Started

declaring a function $pow8' : int \rightarrow int$ doing the three multiplications using two **local declarations**. Local declarations are obtained with **let expressions**

let d in e

combining a declaration d with an expression e using the keywords *let* and *in*. Note that the body of pow' nests two let expressions as *let d_1 in (let d_2 in e)*. OCaml lets you write redundant parentheses marking the nesting (if you want to). Let expressions must not be confused with top-level declarations (which don't use the keyword *in*).

Exercise 1.2.1 Write a function computing x^5 with just 3 multiplications. Write both a version with *square* and a version with local declarations. Write the version with local declarations with and without redundant parentheses marking the nesting.

1.3 Conditionals, Comparisons, and Booleans

The declaration

```
let abs x = if x < 0 then -x else x
```

declares a function $abs : int \rightarrow int$ returning the absolute value of an integer (e.g., $abs(-5) = 5$). The declaration of abs uses a **comparison** $x < 0$ and a **conditional** formed with the keywords **if**, **then**, and **else**.

The declaration

```
let max x y : int = if x <= y then y else x
```

declares a function $max : int \rightarrow int \rightarrow int$ computing the maximum of two numbers. This is the first function we see taking two arguments. There is an explicit **return type specification** in the declaration (appearing as “: int ”), which is needed so that max receives the correct type.³

Given max , we can declare a function

```
let max3 x y z = max (max x y) z
```

returning the maximum of three numbers. This time no return type specification is needed since max forces the correct type $max3 : int \rightarrow int \rightarrow int \rightarrow int$.

Next we declare a three-argument maximum function using a local declaration:

³The complication stems from the design that in OCaml comparisons like \leq also apply to types other than int .

1 Getting Started

```
let max3 x y z : int =
  let a = if x <= y then y else x in
  if a <= z then z else a
```

There is a type *bool* having two values *true* and *false* called **booleans**. The comparison operator “ \leq ” used for *max* (written “ \leq ” in OCaml) is used with the functional type $int \rightarrow int \rightarrow bool$ saying that an expression $e_1 \leq e_2$ where both subexpressions e_1 and e_2 have type *int* has type *bool*.

The declaration

```
let test (x : int) y z = if x <= y then y <= z else false
```

declares a function $test : int \rightarrow int \rightarrow int \rightarrow bool$ testing whether its three arguments appear in order. The **type specification** given for the first argument x is needed so that *test* receives the correct type. Alternatively, type specifications could be given for one or all of the other arguments.

Exercise 1.3.1 Declare minimum functions analogous to the maximum functions declared above. For each declared function give its type (before you check it with the interpreter). Also, write the declarations with and without redundant parentheses to understand the nesting.

Exercise 1.3.2 Declare functions $int \rightarrow int \rightarrow bool$ providing the comparisons $x = y$, $x \neq y$, $x < y$, $x \leq y$, $x > y$, and $x \geq y$. Do this by just using conditionals and comparisons $x \leq y$. Then realize $x \leq y$ with $x \leq 0$ and subtraction.

1.4 Recursive Power Function

Our next goal is a function computing powers x^n using multiplication. We recall that powers satisfy the equations

$$\begin{aligned}x^0 &= 1 \\ x^{n+1} &= x \cdot x^n\end{aligned}$$

Using the equations, we can compute every power x^n where $n \geq 0$. For instance,

$$\begin{aligned}2^3 &= 2 \cdot 2^2 && \text{2nd equation} \\ &= 2 \cdot 2 \cdot 2^1 && \text{2nd equation} \\ &= 2 \cdot 2 \cdot 2 \cdot 2^0 && \text{2nd equation} \\ &= 2 \cdot 2 \cdot 2 \cdot 1 && \text{1st equation} \\ &= 8\end{aligned}$$

1 Getting Started

The trick is that the 2nd equation reduces larger powers to smaller powers, so that after repeated application of the 2nd equation the power x^0 appears, which can be computed with the 1st equation. What we see here is a typical example of a recursive computation.

Recursive computations can be captured with **recursive functions**. To arrive at a function computing powers, we merge the two equations for powers into a single equation using a conditional:

$$x^n = \text{IF } n < 1 \text{ THEN } 1 \text{ ELSE } x \cdot x^{n-1}$$

We now declare a recursive function implementing this equation:

```
let rec pow x n =
  if n < 1 then 1
  else x * pow x (n - 1)
```

The identifier *pow* receives the type $\text{pow} : \text{int} \rightarrow \text{int} \rightarrow \text{int}$. We say that the function *pow* applies itself. Recursive function applications are admitted if the declaration of the function uses the keyword **rec**.

We have demonstrated an important point about programming at the example of the power function: Recursive functions are designed at a mathematical level using equations. Once we have the right equations, we can implement the function in any programming language.

1.5 Integer Division

Given two integers $x \geq 0$ and $y > 0$, there exist unique integers $k, r \geq 0$ such that

$$x = k \cdot y + r$$

and

$$r < y$$

We call k the **quotient** and r the **remainder** of x and y . For instance, given $x = 11$ and $y = 3$, we have the quotient 3 and the remainder 2 (since $11 = 3 \cdot 3 + 2$). We speak of *integer division*, or *division with remainder*, or *Euclidean division*.

We may also characterize the quotient as the largest number k such that $k \cdot y \leq x$, and define the remainder as $r = x - k \cdot y$.

There is a nice geometrical interpretation of integer division. The idea is to place boxes of the same length y next to each other into a shelf of length x . The maximal number of boxes that can be placed into

1 Getting Started

the shelf is the quotient k and the length of the space remaining is the remainder $r = x - k \cdot y < y$. For instance, if the shelf has length 11 and each box has length 3, we can place at most 3 boxes into the shelf, with 2 units of length remaining.⁴

Given that x and y uniquely determine k and r , we are justified in using the notations x/y and $x \% y$ for k and r . By definition of k and r , we have

$$x = (x/y) \cdot y + x \% y$$

and

$$x \% y < y$$

for all $x \geq 0$ and $y > 0$.

From our explanations it is clear that we can compute x/y and $x \% y$ given x and y . In fact, the resulting operations x/y and $x \% y$ are essential for programming and are realized efficiently for machine integers on computers. We refer to the operations as *division* and *modulo*, or just as “div” and “mod”. Accordingly, we read the applications x/y and $x \% y$ as “ x div y ” and “ x mod y ”. OCaml provides both operations as primitive operations using the notations x/y and $x \bmod y$.

Digit sum

With div and mod we can decompose the decimal representation of numbers. For instance, $367 \% 10 = 7$ and $367/10 = 36$. More generally, $x \% 10$ yields the last digit of the decimal representation of x , and $x/10$ cuts off the last digit of the decimal representation of x .

Knowing these facts, we can declare a recursive function computing the digit sum of a number:

```
let rec digit_sum x =  
  if x < 10 then x  
  else digit_sum (x / 10) + (x mod 10)
```

For instance, we have $digit_sum\ 367 = 16$. We note that $digit_sum$ terminates since the argument gets smaller upon recursion.

Exercise 1.5.1 (First digit) Declare a function that yields the first digit of the decimal representation of a number. For instance, the first digit of 367 is 3.

Exercise 1.5.2 (Maximal digit) Declare a function that yields the maximal digit of the decimal representation of a number. For instance, the maximal digit of 376 is 7.

⁴A maybe simpler geometrical interpretation of integer division asks how many boxes of height y can be stacked on each other without exceeding a given height x .

1 Getting Started

Digit reversal

We now write a function *rev* that given a number computes the number represented by the reversed digital representation of the number. For instance, we want $rev\ 76 = 67$, $rev\ 67 = 76$, and $rev\ 7600 = 67$. To write *rev*, we use an important algorithmic idea. The trick is to have an additional **accumulator argument** that is initially 0 and that collects the digits we cut off at the right of the main argument. For instance, we want the **trace**

$$rev' 456\ 0 = rev' 45\ 6 = rev' 4\ 65 = rev' 0\ 654 = 654$$

for the helper function *rev'* with the accumulator argument.

We declare *rev* and the helper function *rev'* as follows:

```
let rec rev' x a =
  if x <= 0 then a
  else rev' (x / 10) (10 * a + x mod 10)
let rev x = rev' x 0
```

We refer to *rev'* as the **worker function** for *rev* and to the argument *a* of *rev'* as the **accumulator argument** of *rev'*. We note that *rev'* terminates since the first argument gets smaller upon recursion.

Greatest common divisors

Recall the notion of greatest common divisors. For instance, the greatest common divisor of 34 and 85 is the number 17. In general, two numbers $x, y \geq 0$ such that $x + y > 0$ always have a unique greatest common divisor. We assume the following rules for greatest common divisors (gcds for short):⁵

1. The gcd of x and 0 is x .
2. If $y > 0$, the gcd of x and y is the gcd of y and $x \% y$.

The two rules suffice to declare a function $gcd : int \rightarrow int \rightarrow int$ computing the gcd of two numbers $x, y \geq 0$ such that $x + y > 0$:

```
let rec gcd x y =
  if y < 1 then x
  else gcd y (x mod y)
```

The function terminates for valid arguments since $(x \% y) < y$ for $x \geq 0$ and $y \geq 1$.

⁵We will prove the correctness of the rules in a later chapter.

1 Getting Started

Computing div and mod with repeated subtraction

We can compute x/y and $x\%y$ using repeated subtraction. To do so, we simply subtract y from x as long as we do not obtain a negative number. Then x/y is the number of successful subtractions and b is the remaining number.

```
let rec my_div x y = if x < y then 0 else 1 + my_div (x - y) y
let rec my_mod x y = if x < y then x else my_mod (x - y) y
```

We remark that both functions terminate for $x \geq 0$ and $y > 0$ since the first argument gets smaller upon recursion.

Exercise 1.5.3 (Traces) Give traces for the following applications:

```
rev' 678 0    rev' 6780 0    gcd 90 120    gcd 153 33    my_mod 17 5
```

We remark that the functions rev' , gcd , and my_mod employ a special form of recursion known as *tail recursion*. We will discuss tail recursion in §1.11.

1.6 Mathematical Level versus Coding Level

When we design a function for OCaml or another programming language, we do this at the mathematical level. The same is true when we reason about functions and their correctness. Designing and reasoning at the mathematical level has the advantage that it serves all programming languages, not just the concrete programming language we have chosen to work with (OCaml in our case). Given the design of a function at the mathematical level, we refer to the realization of the function in a concrete programming language as *coding*. Programming as we understand it emphasizes design and reasoning over coding.

It is important to distinguish between the mathematical level and the coding level. At the mathematical level, we ignore the type *int* of machine integers and instead work with infinite *mathematical types*. In particular, we will use the types

\mathbb{N}	: 0, 1, 2, 3, ...	natural numbers
\mathbb{N}^+	: 1, 2, 3, 4, ...	positive integers
\mathbb{Z}	: ..., -2, -1, 0, 1, 2, ...	integers
\mathbb{B}	: false, true	booleans

When start with the design of a function, it is helpful to fix a mathematical type for the function. Once the type is settled, we can collect

1 Getting Started

equations the function should satisfy. The goal here is to come up with a collection of equations that is sufficient for computing the function.

For instance, when we design a power function, we may start with the mathematical type

$$pow : \mathbb{Z} \rightarrow \mathbb{N} \rightarrow \mathbb{Z}$$

and the equation

$$pow\ x\ n = x^n$$

Together, the type and the equation *specify* the function we want to define. Next we need equations that can serve as defining equations for *pow*. The specifying equation is not good enough since we assume, for the purpose of the example, that the programming language we want to code in doesn't have a power operator. We now recall that powers x^n satisfy the equations

$$\begin{aligned}x^0 &= 1 \\x^{n+1} &= x \cdot x^n\end{aligned}$$

In §1.4 we have already argued that rewriting with the two equations suffices to compute all powers we can express with the type given for *pow*. Next we adapt the equations to the function *pow* we are designing:

$$\begin{aligned}pow\ x\ 0 &= 1 \\pow\ x\ n &= x \cdot pow\ x\ (n - 1) \quad \text{if } n > 0\end{aligned}$$

The second equation now comes with an **application condition** replacing the pattern $n + 1$ in the equation $x^{n+1} = x \cdot x^n$.

We observe that the equations are **exhaustive** and **disjoint**, that is, for all x and n respecting the type of *pow*, the left side of one and only one of the equations applies to the **application** $pow\ x\ n$. We choose the equations as **defining equations** for *pow* and summarize our design with the mathematical definition

$$\begin{aligned}pow : \mathbb{Z} \rightarrow \mathbb{N} \rightarrow \mathbb{Z} \\pow\ x\ 0 &:= 1 \\pow\ x\ n &:= x \cdot pow\ x\ (n - 1) \quad \text{if } n > 0\end{aligned}$$

Note that we write defining equations with the symbol “:=” to mark them as defining.

We observe that the defining equations for *pow* are **terminating**. The **termination argument** is straightforward: Each **recursion step**

1 Getting Started

issued by the second equation decreases the second argument n by 1. This ensures termination since a chain $x_1 > x_2 > x_3 > \dots$ of natural numbers cannot be infinite.

Next we code the mathematical definition as a function declaration in OCaml:

```
let rec pow x n =
  if n < 1 then 1
  else x * pow x (n - 1)
```

The switch to OCaml involves several significant issues:

1. The type of pow changes to $int \rightarrow int \rightarrow int$ since OCaml has no special type for \mathbb{N} (as is typical for execution-oriented programming languages). Thus the OCaml function admits arguments that are not admissible for the mathematical function. We speak of **spurious arguments**.
2. To make pow terminating for negative n , we return 1 for all $n < 1$. We can also use the equivalent comparison $n \leq 0$. If we don't scare away from nontermination for spurious arguments, we can also use the equality test $n = 0$.
3. The OCaml type int doesn't give us the full mathematical type \mathbb{Z} of integers but just a finite interval of machine integers.

When we design a function, there is always a mathematical level governing the coding level for OCaml. One important point about the mathematical level is that it doesn't change when we switch to another programming language. In this text, we will concentrate on the mathematical level.

When we argue the *correctness* of the function pow , we do this at the mathematical level using the infinite types \mathbb{Z} and \mathbb{N} . As it comes to the realization in OCaml, we just hope that the numbers involved for particular examples are small enough so that the difference between mathematical arithmetic and *machine arithmetic* doesn't show. The reason we ignore machine integers at the mathematical level is simplicity.

1.7 More about Mathematical Functions

We use the opportunity and give mathematical definitions for some functions we already discussed. A mathematical function definition consists of a type and a system of defining equations.

1 Getting Started

Remainder

$$\begin{aligned} \% : \mathbb{N} \rightarrow \mathbb{N}^+ \rightarrow \mathbb{N} \\ x \% y &:= x && \text{if } x < y \\ x \% y &:= (x - y) \% y && \text{if } x \geq y \end{aligned}$$

Recall that \mathbb{N}^+ is the type of positive integers $1, 2, 3, \dots$. By using the type \mathbb{N}^+ for the divisor we avoid a division by zero.

Digit sum

$$\begin{aligned} D : \mathbb{N} \rightarrow \mathbb{N} \\ D(x) &:= x && \text{if } x < 10 \\ D(x) &:= D(x/10) + (x \% 10) && \text{if } x \geq 10 \end{aligned}$$

Digit Reversal

$$\begin{aligned} R : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ R \ 0 \ a &:= a \\ R \ x \ a &:= R(x/10) (10 \cdot a + (x \% 10)) && \text{if } x > 0 \end{aligned}$$

A system of defining equations for a function must respect the type specified for the function. In particular, the defining equations must be *exhaustive* and *disjoint* for the type specified for the function.

Often, the defining equations of a function will be *terminating* for all arguments. This is the case for each of the three function defined above. In each case, the same termination argument applies: The first argument, which is a natural number, is decreased by each recursion step.

Curly braces

We can use curly braces to write several defining equations with the same left-hand side with just one left-hand side. For instance, we may write the defining equations of the digit sum function as follows:

$$D \ x \ := \ \begin{cases} x & \text{if } x < 10 \\ D(x/10) + (x \% 10) & \text{if } x \geq 10 \end{cases}$$

Total and partial functions

Functions where the defining equations terminate for all arguments are called **total**, and function where this is not the case are called **partial**. Most mathematical functions we will look at are total, but there are a few partial functions that matter. If there is a way to revise the

1 Getting Started

mathematical definition of a function such that the function becomes total, we will usually do so. The reason we prefer total functions over partial functions is that equational reasoning for total functions is much easier than it is for partial functions. For correct equational reasoning about partial functions we need side conditions making sure that all function applications occurring in an equation used for reasoning do terminate.

We say that a function **diverges** for an argument if it does not terminate for this argument. Here is a function that diverges for all numbers smaller than 10 and terminates for all other numbers:

$$\begin{aligned} f &: \mathbb{N} \rightarrow \mathbb{N} \\ f(n) &:= f(n) && \text{if } n < 10 \\ f(n) &:= n && \text{if } n \geq 10 \end{aligned}$$

Exercise 1.7.1 Define a function $\mathbb{N} \rightarrow \mathbb{N}$ that terminates for even numbers and diverges for odd numbers.

Graph of a function

Abstractly, we may see a function $f : X \rightarrow Y$ as the set of all argument-result pairs $(x, f(x))$ where x is taken from the argument type X . We speak of the **graph** of a function. The graph of a function completely forgets about the definition of the function.

When we speak of a mathematical function in this text, we always include the definition of the function with a type and a system of defining equations. This information is needed so that we can compute with the function.

In noncomputational mathematics, one usually means by a function just a set of argument-result pairs.

GCD function

It is interesting to look at the mathematical version of the gcd function from §1.5:

$$\begin{aligned} G &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ G \ x \ 0 &:= x \\ G \ x \ y &:= G \ y \ (x \% y) && \text{if } y > 0 \end{aligned}$$

The defining equations are terminating since the second argument is decreased upon recursion (since $x \% y < y$ if $y > 0$). Note that the type of G admits $x = y = 0$, a case where no greatest common divisor exists.

1.8 Linear Search and Higher-Order Functions

A *boolean test for numbers* is a function $f : \text{int} \rightarrow \text{bool}$ expressing a condition for numbers. If $f(k) = \text{true}$, we say that k *satisfies* f .

Linear search is an algorithm that given a boolean test $f : \text{int} \rightarrow \text{bool}$ and a number n computes the first number $k \geq n$ satisfying f by checking f for

$$k = n, n + 1, n + 2, \dots$$

until f is satisfied for the first time. We realize linear search with an OCaml function

$$\text{first} : (\text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{int}$$

taking the test as first argument:

```
let rec first f k =
  if f k then k
  else first f (k + 1)
```

Functions taking functions as arguments are called **higher-order functions**. Higher-order functions are a key feature of functional programming.

Recall that we have characterized in §1.5 the integer quotient x/y as the maximal number k such that $k \cdot y \leq x$. Equivalently, we may characterize x/y as the first number $k \geq 0$ such that $(k + 1) \cdot y > x$ (recall the shelf interpretation). This gives us the equation

$$x/y = \text{first} (\lambda k. (k + 1) \cdot y > x) 0 \quad \text{if } x \geq 0 \text{ and } y > 0 \quad (1.1)$$

The functional argument of *first* is described with a **lambda expression**

$$\lambda k. (k + 1) \cdot y > x$$

Lambda expressions are a common mathematical notation for describing functions without giving them a name.⁶

We use Equation 1.1 to declare a function

$$\text{div} : \text{int} \rightarrow \text{int} \rightarrow \text{int}$$

computing quotients x/y :

```
let div x y = first (fun k -> (k + 1) * y > x) 0
```

⁶The greek letter “ λ ” is pronounced “lambda”. Sometimes lambda expressions $\lambda x.e$ are written with the more suggestive notation $x \mapsto e$.

1 Getting Started

From the declaration we learn that OCaml writes lambda expressions with the words “**fun**” and “**->**”. Here is a trace showing how quotients x/y are computed with *first*:

$$\begin{aligned} \text{div } 11 \ 3 &= \text{first } (\lambda k. (k + 1) \cdot 3 > 11) \ 0 && 1 \cdot 3 \leq 11 \\ &= \text{first } (\lambda k. (k + 1) \cdot 3 > 11) \ 1 && 2 \cdot 3 \leq 11 \\ &= \text{first } (\lambda k. (k + 1) \cdot 3 > 11) \ 2 && 3 \cdot 3 \leq 11 \\ &= \text{first } (\lambda k. (k + 1) \cdot 3 > 11) \ 3 && 4 \cdot 3 > 11 \\ &= 3 \end{aligned}$$

We remark that *first* is our first inherently partial function. For instance, the function

$$\text{first } (\lambda k. \text{false}) : \mathbb{N} \rightarrow \mathbb{N}$$

diverges for all arguments. More generally, the application $\text{first } f \ n$ diverges whenever there is no $k \geq n$ satisfying f .

Exercise 1.8.1 Declare a function div' such that $\text{div } x \ y = \text{div}' \ x \ y \ 0$ by specializing *first* to the test $\lambda k. (k + 1) \cdot y > x$.

Exercise 1.8.2 Declare a function $\text{sqrt} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{sqrt}(n^2) = n$ for all n . Hint: Use *first*.

Exercise 1.8.3 Declare a terminating function bounded_first such that $\text{bounded_first } f \ n$ yields the first $k \geq 0$ such that $k \leq n$ and k satisfies f .

1.9 Partial Applications

Functions described with lambda expressions can also be expressed with declared functions. To have an example, we declare *div* with *first* and a helper function *test* replacing the lambda expression:

```
let test x y k = (k + 1) * y > x
let div x y = first (test x y) 0
```

The type of the helper function *test* is $\text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{bool}$. Applying *test* to x and y yields a function of type $\text{int} \rightarrow \text{bool}$ as required by *first*. We speak of a **partial application**. Here is a trace showing how quotients x/y are computed with *first* and *test*:

$$\begin{aligned} \text{div } 11 \ 3 &= \text{first } (\text{test } 11 \ 3) \ 0 && 1 \cdot 3 \leq 11 \\ &= \text{first } (\text{test } 11 \ 3) \ 1 && 2 \cdot 3 \leq 11 \\ &= \text{first } (\text{test } 11 \ 3) \ 2 && 3 \cdot 3 \leq 11 \\ &= \text{first } (\text{test } 11 \ 3) \ 3 && 4 \cdot 3 > 11 \\ &= 3 \end{aligned}$$

1 Getting Started

We may describe the partial applications of *test* with equivalent lambda expressions:

$$\begin{aligned} \text{test } x \ y &= \lambda k. (k + 1) \cdot y > x \\ \text{test } 11 \ 3 &= \lambda k. (k + 1) \cdot 3 > 11 \end{aligned}$$

We can also describe partial applications of *test* to a single argument with equivalent lambda expressions:

$$\begin{aligned} \text{test } x &= \lambda y k. (k + 1) \cdot y > x = \lambda y. \lambda k. (k + 1) \cdot y > x \\ \text{test } 11 &= \lambda y k. (k + 1) \cdot y > 11 = \lambda y. \lambda k. (k + 1) \cdot y > 11 \end{aligned}$$

Note that lambda expressions with two argument variables are notation for nested lambda expressions with single arguments. We can also describe the function *test* with a nested lambda expression:

$$\text{test} = \lambda x y k. (k + 1) \cdot y > x = \lambda x. \lambda y. \lambda k. (k + 1) \cdot y > x$$

Following the nesting of lambda expressions, we may see applications and function types with several arguments as nestings of applications and function types with single arguments:

$$\begin{aligned} e_1 \ e_2 \ e_3 &= (e_1 \ e_2) \ e_3 \\ t_1 \rightarrow t_2 \rightarrow t_3 &= t_1 \rightarrow (t_2 \rightarrow t_3) \end{aligned}$$

Note that applications group to the left and function types group to the right.

We have considered *equations between functions* in the discussion of partial applications of *test*. We consider two functions as equal if they agree on all arguments. So functions with very different definitions may be equal. In fact, two functions are equal if and only if they have the same graph. We remark that there is no algorithm deciding equality of functions in general.

Exercise 1.9.1

- Write $\lambda x y k. (k + 1) \cdot y > x$ as a nested lambda expression.
- Write $\text{test } 11 \ 3 \ 10$ as a nested application.
- Write $\text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{bool}$ as a nested function type.

Exercise 1.9.2 Express the one-argument functions described by the expressions x^2 , x^3 and $(x+1)^2$ with lambda expressions in mathematical notation. Translate the lambda expressions to expressions in OCaml and have them type checked. Do the same for the two-argument function described by the expression $x < k^2$.

1 Getting Started

Exercise 1.9.3 (Sum functions)

- Define a function $\mathbb{N} \rightarrow \mathbb{N}$ computing the sum $0 + 1 + 2 + \dots + n$ of the first n numbers.
- Define a function $\mathbb{N} \rightarrow \mathbb{N}$ computing the sum $0 + 1^2 + 2^2 + \dots + n^2$ of the first n square numbers.
- Define a function $sum : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ computing for a given function f the sum $f(0) + f(1) + f(2) + \dots + f(n)$.
- Give partial applications of the function sum from (c) providing specialized sum functions as asked for by (a) and (b).

1.10 Inversion of Strictly Increasing Functions

A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is **strictly increasing** if

$$f(0) < f(1) < f(2) < \dots$$

Strictly increasing functions can be inverted using linear search. That is, given a strictly increasing function $f : \mathbb{N} \rightarrow \mathbb{N}$, we can construct a function $g : \mathbb{N} \rightarrow \mathbb{N}$ such that $g(f(n)) = n$ for all n . The construction is explained by the equation

$$first (\lambda k. f(k + 1) > f(n)) 0 = n \quad (1.2)$$

which in turn gives us the equation

$$(\lambda x. first (\lambda k. f(k + 1) > x) 0) (f(n)) = n \quad (1.3)$$

For a concrete example, let $f(n) := n^2$. Equation 1.3 tells us that

$$g(x) := first (\lambda k. (k + 1)^2 > x) 0$$

is a function $\mathbb{N} \rightarrow \mathbb{N}$ such that $g(n^2) = n$ for all n . Thus we know that

$$sqrt\ x := first (\lambda k. (k + 1)^2 > x) 0$$

computes integer square roots $\lfloor \sqrt{x} \rfloor$. For instance, we have $sqrt(1) = 1$, $sqrt(4) = 2$, and $sqrt(9) = 3$. The **floor operator** $\lfloor x \rfloor$ converts a real number x into the greatest integer $y \leq x$.

Exercise 1.10.1 Give a trace for $sqrt\ 10$.

Exercise 1.10.2 Declare a function $sqrt'$ such that $sqrt\ x = sqrt'\ x 0$ by specializing $first$ to the test $\lambda k. (k + 1)^2 > x$.

1 Getting Started

Exercise 1.10.3 The **ceiling operator** $\lceil x \rceil$ converts a real number into the least integer y such that $x \leq y$.

- Declare a function computing rounded down cube roots $\lfloor \sqrt[3]{x} \rfloor$.
- Declare a function computing rounded up cube roots $\lceil \sqrt[3]{x} \rceil$.

Exercise 1.10.4 Let $y > 0$. Convince yourself that $\lambda x. x/y$ inverts the strictly increasing function $\lambda n. n \cdot y$.

Exercise 1.10.5 Declare inverse functions for the following functions:

- $\lambda n. n^3$
- $\lambda n. n^k$ for $k \geq 2$
- $\lambda n. k^n$ for $k \geq 2$

Exercise 1.10.6 Declare a function $inv : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ that given a strictly increasing function f yields a function inverting f . Then express the functions from Exercise 1.10.5 using inv .

Exercise 1.10.7 Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be strictly increasing. Convince yourself that the functions

$$\begin{aligned} &\lambda x. first (\lambda k. f(k) = x) 0 \\ &\lambda x. first (\lambda k. f(k) \geq x) 0 \\ &\lambda x. first (\lambda k. f(k + 1) > x) 0 \end{aligned}$$

all invert f and find out how they differ.

1.11 Tail Recursion

A special form of functional recursion is **tail recursion**. Tail recursion matters since it can be executed more efficiently than general recursion. Tail recursion imposes the restriction that recursive function applications can only appear in tail positions where they directly yield the result of the function. Hence recursive applications appearing as part of another application (operator or function) are not tail recursive. Typical examples of tail recursive functions are the functions rev' , gcd , my_mod , and $first$ we have seen before. Counterexamples for tail recursive functions are the recursive functions pow (the recursive application is nested into a product) and my_div (the recursive application is nested into a sum).

1 Getting Started

Tail recursive functions have the property that their execution can be traced in a simple way. For instance, we have the tail recursive trace

$$\begin{aligned} \text{gcd } 36 \ 132 &= \text{gcd } 132 \ 36 \\ &= \text{gcd } 36 \ 24 \\ &= \text{gcd } 24 \ 12 \\ &= \text{gcd } 12 \ 0 \\ &= 12 \end{aligned}$$

For functions where the recursion is not tail recursive, traces look more complicated, for instance

$$\begin{aligned} \text{pow } 2 \ 3 &= 2 \cdot \text{pow } 2 \ 2 \\ &= 2 \cdot (2 \cdot \text{pow } 2 \ 1) \\ &= 2 \cdot (2 \cdot (2 \cdot \text{pow } 2 \ 0)) \\ &= 2 \cdot (2 \cdot (2 \cdot 1)) \\ &= 8 \end{aligned}$$

In imperative programming languages tail recursive functions can be expressed with loops. While imperative languages are designed such that loops should be used whenever possible, functional programming languages are designed such that tail recursive functions are preferable over loops.

Often recursive functions that are not tail recursive can be reformulated as tail recursive functions by introducing an extra argument serving as accumulator argument. Here is a tail recursive version of *pow*:

```
let rec pow' x n a =  
  if n < 1 then a  
  else pow' x (n - 1) (x * a)
```

We explain the role of the accumulator argument with a trace:

$$\begin{aligned} \text{pow}' \ 2 \ 3 \ 1 &= \text{pow}' \ 2 \ 2 \ 2 \\ &= \text{pow}' \ 2 \ 1 \ 4 \\ &= \text{pow}' \ 2 \ 0 \ 8 \\ &= 8 \end{aligned}$$

Exercise 1.11.1 (Factorials) In mathematics, the **factorial** of a positive integer n , denoted by $n!$, is the product of all positive integers less than or equal to n :

$$n! = n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1$$

1 Getting Started

For instance,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

In addition, $0!$ is defined as 1. We capture this specification with a recursive function defined as follows:

$$\begin{aligned} ! : \mathbb{N} &\rightarrow \mathbb{N} \\ 0! &:= 1 \\ (n + 1)! &:= (n + 1) \cdot n! \end{aligned}$$

- Declare a function $fac : int \rightarrow int$ computing factorials.
- Define a tail recursion function $f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ such that $n! = f \ 1 \ n$.
- Declare a tail recursive function $fac' : int \rightarrow int \rightarrow int$ such that $fac' \ 1$ computes factorials.

1.12 Tuples

Sometimes we want a function that returns more than one value. For instance, the time for a marathon may be given with three numbers in the format $h : m : s$, where h is the number of hours, m is the number of minutes, and s is the number of seconds the runner needed. The time of Eliud Kipchoge's world record in 2018 in Berlin was $2 : 01 : 39$. There is the constraint that $m < 60$ and $s < 60$.

OCaml has **tuples** to represent collections of values as single values. To represent the marathon time of Kipchoge in Berlin, we can use the tuple

$$(2, 1, 39) : int \times int \times int$$

consisting of three integers. The product symbol “ \times ” in the **tuple type** is written as “ $*$ ” in OCaml. The **component types** of a tuple are not restricted to int , and there can be $n \geq 2$ **positions**, where n is called the **length** of the tuple. We may have tuples as follows:

$$\begin{aligned} (2, 3) &: int \times int \\ (7, true) &: int \times bool \\ ((2, 3), (7, true)) &: (int \times int) \times (int \times bool) \end{aligned}$$

Note that the last example nests tuples into tuples. We mention that tuples of length 2 are called **pairs**, and that tuples of length 3 are called **triples**.

1 Getting Started

We can now write two functions

$$\begin{aligned} \text{sec} &: \text{int} \times \text{int} \times \text{int} \rightarrow \text{int} \\ \text{hms} &: \text{int} \rightarrow \text{int} \times \text{int} \times \text{int} \end{aligned}$$

translating between times given in total seconds and times given as (h, m, s) tuples:

```
let sec (h,m,s) = 3600 * h + 60 * m + s
let hms x =
  let h = x / 3600 in
  let m = (x mod 3600) / 60 in
  let s = x mod 60 in
  (h,m,s)
```

Exercise 1.12.1

- Give a tuple of length 5 where the components are the values 2 and 3.
- Give a tuple of type $\text{int} \times (\text{int} \times (\text{bool} \times \text{bool}))$.
- Give a pair whose first component is a pair and whose second component is a triple.

Exercise 1.12.2 (Sorting triples) Declare a function *sort* sorting triples. For instance, we want $\text{sort}(3, 2, 1) = (1, 2, 3)$. Designing such a function is interesting. Given a triple (x, y, z) , the best solution we know of first ensures $y \leq z$ and then inserts x at the correct position. Start from the code snippet

```
let sort (x,y,z) =
  let (y,z) = if y <= z then (y,z) else (z, y) in
  if x <= y then ...
  else ...
```

where the local declaration ensures $y \leq z$ using shadowing.

Exercise 1.12.3 (Medians) The median of three numbers is the number in the middle. For instance, the median of 5, 0, 1 is 1. Declare a function that takes three numbers and yields the median of the numbers.

1.13 Exceptions and Spurious Arguments

What happens when we execute the native operation $5/0$ in OCaml? Execution is aborted and an **exception** is reported:

Exception: `Division_by_zero`

1 Getting Started

Exceptions can be useful when debugging erroneous programs. We will say more about strings and exceptions in later chapters.

There is no equivalent to exceptions at the mathematical level. At the mathematical level we use types like \mathbb{N}^+ or side conditions like $y \neq 0$ to exclude undefined applications like $x/0$.

When coding a mathematical function in OCaml, we need to replace mathematical types like \mathbb{N} with the OCaml type *int*. This introduces **spurious arguments** not anticipated by the mathematical function. There are different ways to cope with spurious arguments:

1. Ignore the presence of spurious arguments. This is the best strategy when you solve exercises in this text.
2. Use a wrapper function raising exceptions when spurious arguments show up. The wrapper function facilitates the discovery of situations where functions are accidentally applied to spurious arguments.

As an example, we consider the coding of the mathematical remainder function

$$\begin{aligned} \text{rem} : \mathbb{N} \rightarrow \mathbb{N}^+ \rightarrow \mathbb{N} \\ \text{rem } x \ y &:= x && \text{if } x < y \\ \text{rem } x \ y &:= \text{rem } (x - y) \ y && \text{if } x \geq y \end{aligned}$$

as the OCaml function

```
let rec rem x y = if x < y then x else rem (x - y) y
```

receiving the type $\text{int} \rightarrow \text{int} \rightarrow \text{int}$. In OCaml we now have the spurious situation that $\text{rem } x \ 0$ diverges for all $x \geq 0$. There other spurious situations whose analysis is tedious since machine arithmetic needs to be taken into account. Using the wrapper function

```
let rem_checked x y =  
  if x >= 0 && y > 0 then rem x y  
  else invalid_arg "rem_checked"
```

all spurious situations are uniformly handled by throwing the exception

```
Invalid_argument "rem_checked"
```

There are several new features here:

- The lazy boolean and connective `x >= 0 && y > 0` tests two conditions and is equivalent to `if x >= 0 then y > 0 else false`.
- There is the string `"rem_checked"`. Strings are values like integers and booleans and have type *string*.

1 Getting Started

- The predefined function *invalid_arg* raises an exception saying that *rem_checked* was called with spurious arguments.⁷

When an exception is raised, execution of a program is aborted and the exception raised is reported.

We use the opportunity and introduce the **lazy boolean connectives** as abbreviations for conditionals:

$e_1 \ \&\& \ e_2$	\rightsquigarrow	IF e_1 THEN e_2 ELSE false	lazy and
$e_1 \ \ e_2$	\rightsquigarrow	IF e_1 THEN true ELSE e_2	lazy or

Exercise 1.13.1 Consider the declaration

```
let eager_or x y = x || y
```

Find expressions e_1 and e_2 such that the expressions $e_1 || e_2$ and *eager_or* e_1 e_2 behave differently. Hint: Choose a diverging expression for e_2 and keep in mind that execution of a function application executes all argument expressions. In contrast, execution of a conditional IF e_1 THEN e_2 ELSE e_3 executes e_1 and then either e_2 or e_3 , but not both.

Exercise 1.13.2 (Sorting triples) Recall Exercise 1.12.2. With lazy boolean connectives a function sorting triples can be written without much thinking by doing a naive case analysis considering the alternatives x is in the middle or y is in the middle or z is in the middle.

1.14 Summary

After working through this chapter you should be able to design and code functions computing powers, integer quotients and remainders, digit sums, digit reversals, and integer roots. You should understand that the design of functions happens at a mathematical level using mathematical types and equations. A given design can then be refined into a program in a given programming language. In this text we are using OCaml as programming language, assuming that this is the first programming language you see.⁸

⁷OCaml says “invalid argument” for “spurious argument”.

⁸Most readers will have done some programming in some programming language before starting with this text. Readers of this group often face the difficulty that they invest too much energy on mapping back the new things they see here to the form of programming they already understand. Since functional programming is rather different from other forms of programming, it is essential that you open yourself to the new ideas presented here. Keep in mind that a good programmer quickly adapts to new ways of thinking and to new programming languages.

1 *Getting Started*

You also saw a first higher-order function *first*, which can be used to obtain integer quotients and integer roots with a general scheme known as linear search. You will see many more examples of higher-order functions expressing basic computational schemes.

The model of computation we have assumed in this chapter is rewriting with defining equations. In this model, recursion appears in a natural way. You will have noticed that recursion is the feature where things get really interesting. We also have discussed tail recursion, a restricted form of recursion with nice properties we will study more carefully as we go on. All recursive functions we have seen in this chapter have equivalent tail recursive formulations (often using an accumulator argument).

Finally, we have seen tuples, which are compound values combining several values into a single value.

2 Syntax and Semantics

Programming languages are structured into 4 layers:

- **Lexical Syntax** This layer concerns the translation of sequences of characters into a sequences of words.
- **Phrasal Syntax** This layer concerns the translation of sequences of words into syntax trees.
- **Static Semantics** This layer concerns the conditions syntax trees must satisfy to be well-formed.
- **Dynamic Semantics** This layer concerns the evaluation of syntax trees. In contrast to the processing of the other layers, which either succeeds or fails, evaluation of a syntax tree may diverge (i.e., not terminate),

The first three layers are called **static layers**. Every program is processed following the four layers. First, the given sequence of characters is translated into a sequence of words, then the sequence of words is translated into a syntax tree, which is then elaborated and checked. An interpreter will then evaluate the syntax tree representing the program. There are also compilers compiling programs via syntax trees into low-level programs for **machines**. Compiled programs may be stored and executed on suitable machines. An interpreter for a language L may be seen as a machine for L realized with software.

When a program is processed, errors can occur at each of the four layers. Usually, a program will only be executed if there are no **static errors** (i.e., errors concerning a static layer). When a program is executed, **dynamic errors** like “division by zero” may still occur.

The components of an interpreter or compiler taking care of the static layers are known as **lexer** (lexical syntax), **parser** (phrasal syntax), and **type checker** (static semantics).

In this chapter we consider sublanguages of OCaml covering many of the features we have seen in the first chapter.

This chapter is a demanding discussion of the structure of programming languages introducing many new notions beginners will not have seen before. To gain a better idea of where the journey goes, the reader may first skim the last section of this chapter giving a high-level review of the material of this chapter.

2.1 Lexical Syntax

The lexical syntax of a programming languages fixes the characters a program can be written with. There are **white space characters** including space, horizontal tabulation, carriage return, and line feed. The lexical syntax also fixes different classes of words. We have seen the following classes of words in OCaml:

- **Keywords** We have seen the keywords

```
let rec in if then else fun
```

The symbols

```
( ) = : -> * ,
```

also count as keywords.

- **Operators** We have seen the operators

```
+ - / mod <> < <= > >=
```

There are many more.

- **Boolean literals** `true` and `false`.
- **Integer literals** Examples are `0` and `2456`.
- **String literals** For instance, `"Saarbrücken"`. String literals can contain special letters like the German umlauts.
- **Identifiers** Examples are `pow`, `pow'`, and `div_checked`. Identifiers start with a lower case letter and can contain letters, digits and the characters “`_`” (underline) and “`'`” (prime). Identifiers exclude words that appear as keywords, operators, or boolean literals. Moreover, identifiers must not contain special letters like the German umlauts. The identifiers `int`, `bool`, and `string` are used as names for the respective types.

Words are usually separated by white space characters. In some cases, white space characters are not needed, as for instance in the character sequence `3*(x/y)` where every character is a word.

The lexical word classes are disjoint. This shows in the symbols “`=`” and “`*`”, which are classified as keywords but not as operators. This fits their use as keywords in declarations and types. However, the symbols “`=`” and “`*`” may also serve as infix operators in expressions. The necessary disambiguation will happen at the level of the phrasal syntax.

There are also **comments**, which are character sequences

```
(* ... *)
```

2 Syntax and Semantics

counting as white space. Comments are only visible at the lexical level. Comments are useful for remarks the programmer wants to write into a program. Here is a text where the words identified by the lexical syntax are marked:

```
(* Linear Search *)
let rec first f k =
  if f k then k      (* base case *)
  else first f ( k + 1 )  (* recursive case *)
```

Note that the characters of the three comments in the text count as white space and don't contribute to the words.

Consult the OCaml manual if you want to know more about the lexical syntax of OCaml.

Exercise 2.1.1 Mark the words in the following character sequences. For each word give the lexical class.

- a) `if x <= 1 then true else f (x+1)`
- b) `let rec f x : int = (*TODO*) +x`
- c) `let city = "Saarbrücken" in`
- d) `int * int -> bool`
- e) `if rec then <3`

Why does the character sequence `Saarbrücken` not represent a valid word?

2.2 Phrasal Syntax

The semantic layers of a language don't see programs as sequences of characters or words. Instead, the semantic layers of a language see abstract syntactic objects called **phrases**. Phrases can be described by sequences of words but are best understood as **syntax trees**. It is the job of the phrasal syntax to say which phrases there are and how they are represented as syntax trees and as sequences of words. While every phrase can be represented with a sequence of words, not every sequence of words represents a phrase. The phrasal syntax organizes the phrases of the language into different **syntactic classes**. For OCaml, the syntactic classes include types, expressions, declarations, and programs. Every phrase of the language belongs to exactly one of the syntactic classes of the language and has a unique representation as syntax tree.

At the phrasal level we are using mathematical symbols for the symbolic words of the lexical syntax. Here is the translation table:

2 Syntax and Semantics

+	-	/	mod	+	-	/	%
*				·	or	×	
=	<>			=	≠		
<	<=	>	>=	<	≤	>	≥
->				->			

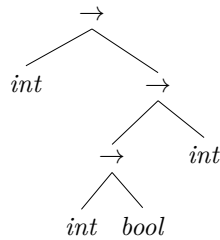
The keyword “*” is special in that it translates into the symbol \times if used to form tuple types, and into the symbol \cdot if used as multiplication operator.

2.2.1 Types

We have seen the **base types** *int*, *bool*, and *string*. Types can be combined into **function types** and **tuple types** using the symbols \rightarrow and \times . As example we consider the **nested function type**

$$int \rightarrow (int \rightarrow bool) \rightarrow int$$

The type is given in lexical representation. As a syntax tree, the type takes the form



making explicit the nesting of the outer function type. The nesting can also be indicated with **redundant parentheses** in the lexical representation:

$$int \rightarrow ((int \rightarrow bool) \rightarrow int)$$

Redundant parentheses are always possible. If you like, you can write the base type *int* as $((int))$.

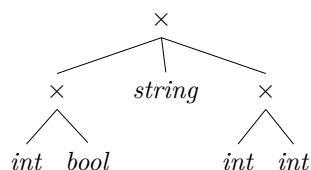
The binary structure of function types explains functions with several arguments as functions taking single arguments and returning functions. Partial applications are a necessary feature of the binary structure of function types.

Next we look at a **nested tuple type**

$$(int \times bool) \times string \times (int \times int)$$

whose tree representation looks as follows:

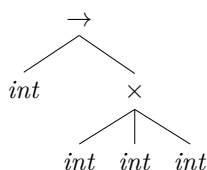
2 Syntax and Semantics



Another example is the function type

$$int \rightarrow int \times int \times int$$

whose result type is a tuple type. The tree representation of this type is



In contrast to function types, there is no implicit nesting of tuple types. Thus the types $int \times int \times int$ and $int \times (int \times int)$ are different: The first tuple type has three components while the second tuple type has only two components.

It now should be clear that syntax trees are a device making explicit the nesting of phrases and eliminating lexical details that are semantically irrelevant.

The syntactic structure of types described so far with examples can be expressed succinctly with a **grammar**:

$$\begin{aligned}
 \langle type \rangle &::= \langle base\ type \rangle \\
 &| \langle type \rangle \rightarrow \langle type \rangle \\
 &| \langle type \rangle_1 \times \cdots \times \langle type \rangle_n \quad (n \geq 2) \\
 \langle base\ type \rangle &::= int \mid bool \mid string
 \end{aligned}$$

The grammar defines the syntactic classes $\langle type \rangle$ and $\langle base\ type \rangle$ by means of **grammar rules**, where each rule offers three alternatives separated by a vertical bar “|”. Function types and tuple types are **compound types** coming with **constituents** that are types again. A **function type** $\langle type \rangle \rightarrow \langle type \rangle$, for instance, has two constituents, the **argument type** and the **result type**. A **tuple type** $\langle type \rangle_1 \times \cdots \times \langle type \rangle_n$ has $n \geq 2$ constituents called **component types**. Tuple types are also known as *product types*.

Grammars like the above grammar for types are often called BNFs. Google BNF (Backus-Naur form) to learn more about their origin.

2 Syntax and Semantics

Exercise 2.2.1 Draw syntax trees for the following phrases in lexical representation.

- $int \times int \rightarrow bool$
- $int \times (int \times int) \times int \rightarrow bool$

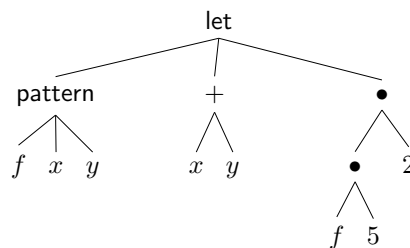
2.2.2 Expressions

Following the ideas we have seen for types, we now describe the phrasal syntax of a class of expressions with the grammar shown in Figure 2.1. To keep things manageable, we only consider a small sublanguage of OCaml. In particular, we do not consider type specifications in lambda expressions and let expressions.

Note that there are **binary operator applications** (e.g., $e_1 + e_2$) and **unary operator applications** (e.g., $-e$ and $+e$).

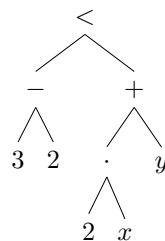
We illustrate the syntactic structure described by the grammar with a let expression represented both with a sequence of words and a syntax tree:

let $f\ x\ y = x + y$ in $f\ 5\ 2$



Note that the let expression shown contains patterns, a binary operator application, and two function applications.

One thing we need to explain about the syntactic structure of expressions in addition to what is explained by the grammar is how parentheses can be omitted in lexical representations. For instance, the expression



2 Syntax and Semantics

$$\begin{aligned}
 \langle \textit{expression} \rangle & ::= \\
 & | \langle \textit{atomic expression} \rangle \\
 & | \langle \textit{operator application} \rangle \\
 & | \langle \textit{function application} \rangle \\
 & | \langle \textit{conditional} \rangle \\
 & | \langle \textit{tuple expression} \rangle \\
 & | \langle \textit{lambda expression} \rangle \\
 & | \langle \textit{let expression} \rangle \\
 \\
 \langle \textit{atomic expression} \rangle & ::= \langle \textit{literal} \rangle | \langle \textit{identifier} \rangle \\
 \\
 \langle \textit{operator application} \rangle & ::= \\
 & | \langle \textit{expression} \rangle \langle \textit{operator} \rangle \langle \textit{expression} \rangle \\
 & | \langle \textit{operator} \rangle \langle \textit{expression} \rangle \\
 \\
 \langle \textit{operator} \rangle & ::= + | - | \cdot | / | \% | = | \neq | < | \leq | > | \geq \\
 \\
 \langle \textit{function application} \rangle & ::= \langle \textit{expression} \rangle \langle \textit{expression} \rangle \\
 \\
 \langle \textit{conditional} \rangle & ::= \textit{if} \langle \textit{expression} \rangle \textit{then} \langle \textit{expression} \rangle \textit{else} \langle \textit{expression} \rangle \\
 \\
 \langle \textit{tuple expression} \rangle & ::= (\langle \textit{expression} \rangle_1, \dots, \langle \textit{expression} \rangle_n) \quad (n \geq 2) \\
 \\
 \langle \textit{lambda expression} \rangle & ::= \textit{fun} \langle \textit{pattern} \rangle_1 \dots \langle \textit{pattern} \rangle_n \rightarrow \langle \textit{expression} \rangle \quad (n \geq 1) \\
 \\
 \langle \textit{let expression} \rangle & ::= \textit{let} \langle \textit{let pattern} \rangle = \langle \textit{expression} \rangle \textit{in} \langle \textit{expression} \rangle \\
 \\
 \langle \textit{let pattern} \rangle & ::= \\
 & | \langle \textit{pattern} \rangle \\
 & | \langle \textit{identifier} \rangle \langle \textit{pattern} \rangle_1 \dots \langle \textit{pattern} \rangle_n \quad (n \geq 1) \\
 & | \textit{rec} \langle \textit{identifier} \rangle \langle \textit{pattern} \rangle_1 \dots \langle \textit{pattern} \rangle_n \quad (n \geq 0) \\
 \\
 \langle \textit{pattern} \rangle & ::= \\
 & | \langle \textit{identifier} \rangle \\
 & | (\langle \textit{pattern} \rangle_1, \dots, \langle \textit{pattern} \rangle_n) \quad (n \geq 2)
 \end{aligned}$$

Figure 2.1: Grammar for expressions

2 Syntax and Semantics

may be described with the sequence of words

$$(3 - 2) < ((2 \cdot x) + y)$$

making explicit all nestings through parentheses, but can also be described with the sequence

$$3 - 2 < 2 \cdot x + y$$

omitting all parentheses following common mathematical conventions.

We start with the **parentheses rule for function applications**:

$$e_1 e_2 e_3 \rightsquigarrow (e_1 e_2) e_3$$

The rule says that function applications group to the left, which ensures that arguments are taken from left to right. The lexical representation of the let expression shown above already made use of this rule.

The **parentheses rules for binary operators** distinguish 3 levels:

$$\begin{array}{ccccccc} = & \neq & < & \leq & > & \geq & \\ & & + & - & & & \\ & & \cdot & / & \% & & \end{array}$$

The operators at the lowest level take their arguments first, followed by the operators at the mid-level, followed by the comparison operators at the top level. This rule explains why we have

$$3 - 2 < 2 \cdot x + y \rightsquigarrow (3 - 2) < ((2 \cdot x) + y)$$

If we have several operators at the same level, they take their arguments from left to right. For instance,

$$e_1 + e_2 + e_3 - e_4 - e_5 \rightsquigarrow (((e_1 + e_2) + e_3) - e_4) - e_5$$

The operators “+” and “-” can also be used as unary operators. When used as unary operators, “+” and “-” take their arguments before all binary operators. For instance,

$$1 + + + 5 - - + - 7 \rightsquigarrow (1 + (+(+5))) - (-(+(-7)))$$

Note that the grammar in Figure 2.1 allows for free nesting of expressions. If you like, you can nest a parenthesized conditional into a function application, or a let expression into the first constituent of a conditional.

2 Syntax and Semantics

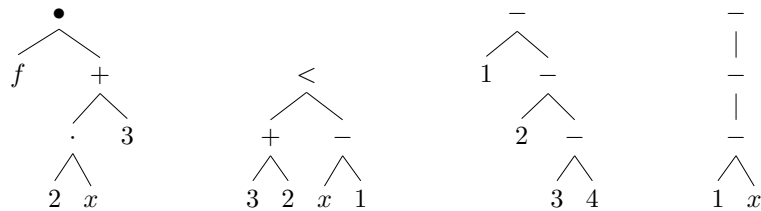
There is the rule that function applications and operator applications group before other syntactic forms. The rule has the effect that the scope of lambda expressions is taken as large as possible:

$$\begin{aligned} \lambda x. e_1 e_2 &\rightsquigarrow \lambda x. (e_1 e_2) \\ \lambda x. e_1 o e_2 &\rightsquigarrow \lambda x. (e_1 o e_2) \end{aligned}$$

Exercise 2.2.2 Draw syntax trees for the following phrases:

- | | |
|---|--|
| a) $x + 3 \cdot f(x) - 4$ | e) $1 + 2 \cdot x - y \cdot 3 + 4$ |
| b) $1 + (2 + (3 + 4))$ | f) $(1 + 2) \cdot x - (y \cdot 3) + 4$ |
| c) $1 + 2 + 3 + 4$ | g) IF $x < 3$ THEN 3 ELSE $p(3)$ |
| d) $1 + (2 + 3) + 4$ | h) LET $x = 2 + y$ IN $x - y$ |
| i) LET $p \ x \ n =$ IF $n > 0$ THEN $x \cdot p \ x \ (n - 1)$ ELSE 1 IN $p \ 2 \ 10$ | |

Exercise 2.2.3 (Redundant parentheses) Give lexical representations for the following syntax trees not using redundant parentheses:



2.2.3 Declarations and Programs

There is no need to say much about declarations and programs: Declarations already appear as prefixes of let expressions (the part before the keyword `in`), and programs are simply sequences of declarations. We may describe declarations and programs with the following grammar:

$$\begin{aligned} \langle \text{program} \rangle &::= \langle \text{declaration} \rangle_1 \cdots \langle \text{declaration} \rangle_n \quad (n \geq 0) \\ \langle \text{declaration} \rangle &::= \text{let } \langle \text{let pattern} \rangle = \langle \text{expression} \rangle \end{aligned}$$

Once we have the syntactic class $\langle \text{declaration} \rangle$, we could model let expressions as trees with only two subtrees

$$\langle \text{let expression} \rangle ::= \langle \text{declaration} \rangle \text{ in } \langle \text{expression} \rangle$$

What we see here is that there are many design choices for the grammar of a language, even after the lexical representation of phrases is fixed. Once the grammar is fixed, the structure of syntax trees is fixed, up to the exact choices of labels.

2.3 Derivation Systems

The semantic layers of a programming language are described with derivation systems. **Derivation systems** are systems of **inference rules** deriving statements called **judgments**. Judgments may be seen as syntactic objects. We explain derivation systems with a system deriving judgments $x \dot{<} y$ where x and y are integers. The system consists of two inferences rules:

$$\frac{}{x \dot{<} x + 1} \qquad \frac{x \dot{<} y \quad y \dot{<} z}{x \dot{<} z}$$

We may verbalize the rules as saying:

1. Every number is smaller than its successor.
2. If x is smaller than y and y is smaller than z , then x is smaller than z .

Derivations of judgments are obtained by combining rules. Here are two different derivations of the judgment $3 \dot{<} 6$:

$$\frac{\frac{3 \dot{<} 4}{\quad} \quad \frac{\frac{4 \dot{<} 5}{\quad} \quad \frac{5 \dot{<} 6}{\quad}}{4 \dot{<} 6}}{3 \dot{<} 6} \qquad \frac{\frac{3 \dot{<} 4}{\quad} \quad \frac{4 \dot{<} 5}{\quad}}{3 \dot{<} 5} \quad \frac{5 \dot{<} 6}{\quad}}{3 \dot{<} 6}$$

Every line in the derivations represents the application of one of the two inferences rules. Note that the initial judgments of the derivation are justified by the first rule, and that the non-initial judgments of the derivation are justified by the second rule.

In general, an **inference rule** has the format

$$\frac{P_1 \quad \dots \quad P_n}{C}$$

where the **conclusion** C is a judgment and the **premises** P_1, \dots, P_n are either judgments or side conditions. A rule says that a derivation of the conclusion judgment can be obtained from the derivations of the premise judgments provided the side conditions appearing as premises are satisfied. The inference rules our example system don't have side conditions.

Returning to our example derivation system, it is clear that whenever a judgment $x \dot{<} y$ is derivable, the comparison $x < y$ is true. This follows from the fact that both inference rules express valid properties of comparisons $x < y$. Moreover, given two numbers x and y such $x < y$, we can always construct a derivation of the judgment $x \dot{<} y$ following a recursive algorithm:

2 Syntax and Semantics

1. If $y = x + 1$, the first rule yields a derivation of $x \dot{<} y$.
2. Otherwise, obtain a derivation of $x \dot{<} y - 1$ by recursion. Moreover, obtain a derivation of $y - 1 \dot{<} y$ by the first rule. Now obtain a derivation of $x \dot{<} y$ using the second rule.

The example derivation system considered here was chosen for the purpose of explanation. In practice, derivation systems are used for describing computational relations that cannot be described otherwise. Typing and evaluation of expressions are examples for such relations, and we are going to use derivation systems for this purpose.

Exercise 2.3.1 Give all derivations for $5 \dot{<} 9$.

Exercise 2.3.2 Give two inference rules for judgments $x \dot{<} y$ such that your rules derive the same judgments the rules given above derive, but have the additional property that no judgments has more than one derivation.

Exercise 2.3.3 Give two inference rules for judgments $x \dot{<} y$ such that a judgment $x \dot{<} y$ is derivable if and only if $x + k \cdot 5 = y$ for some $k \geq 1$.

2.4 Abstract Expressions

The semantic layers of a language see the phrases of the language as abstract syntactic objects where lexical details don't play a role. This brings us back to the philosophy of Chapter 1 where we explained OCaml with abstract syntactic notions such as expressions, types, and declarations and lexical details were conveyed tacitly through the code for mathematically motivated examples.

For the discussion of the semantic layers, we switch to a language of abstract expressions differing from the sublanguage considered for the syntactic layers. The reason for the switch is ease of explanation. The new language has types as considered before

$$t ::= \text{int} \mid \text{bool} \mid \text{string} \mid t_1 \rightarrow t_2 \mid t_1 \times \cdots \times t_n \quad n \geq 2$$

2 Syntax and Semantics

and **abstract expressions** as described by the following **grammar**:

$$\begin{aligned}
 e ::= & c \mid x \mid e_1 o e_2 \mid e_1 e_2 \\
 & \mid \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \\
 & \mid (e_1, \dots, e_n) && n \geq 2 \\
 & \mid \pi_n e && n \geq 1 \\
 & \mid \text{LET } x = e_1 \text{ IN } e_2 \\
 & \mid \lambda x. e \\
 & \mid \text{LET REC } f x = e_1 \text{ IN } e_2
 \end{aligned}$$

The letter c ranges over **constants** (lexically represented as literals), the letters x and f range over **variables** (lexically represented as identifiers), and the letter o ranges over **operators**. Constants are values of the types *int*, *bool*, and *string*. For simplicity, we just have binary operator applications. The variables introduced by lambda expressions and let expressions are called **local variables**. Lambda expressions are written in mathematical notation and take only one argument variable. Similarly, let expressions bind only one or two variables. Tuple patterns are replaced by **projections** π_n , yielding the n th component of a tuple.

We will refer to the expressions identified by the phrasal syntax as **concrete expressions**. It is possible to translate concrete expressions into equivalent abstract expressions. The reverse translation is trivial as long as no projections are used. The translation of projections poses a problem since projections don't commit to a particular tuple length. The problem can be eliminated by working with **graded projections** π_i^n that yield the i th component of tuples of length n .

Exercise 2.4.1 Translate the following concrete expressions into equivalent abstract expressions. Use graded projections.

- a) LET $(x, y) = z$ IN $x + y$
- b) $\lambda fxy. fxy$
- c) LET REC $f x y = \text{IF } x < 1 \text{ THEN } y \text{ ELSE } f(x - 1)(y + 1)$ IN $f \ 5 \ 6$

2.5 Static Semantics

We formulate the static semantics of abstract expressions using **typing judgments**

$$E \vdash e : t$$

saying that in the environment E the expression e has type t . An **environment** is a finite collection of **bindings** $x : t$ mapping variables

2 Syntax and Semantics

to types. We assume that environments are **functional**, that is, have at most one binding per variable. Here is an example for a typing judgment:

$$x : int, y : bool \vdash \text{IF } y \text{ THEN } x \text{ ELSE } 0 : int$$

The inference rules in Figure 2.2 establish a derivation system for typing judgments. We will refer to the rules as **typing rules**. If the judgment $E \vdash e : t$ is derivable, we say that expression e has type t in environment E . The environment is needed so that e can contain variables that are not bound by surrounding lambda expressions or let expressions. The typing rules for constants and for operator applications assume that the types for constants and operators are given in tables (one type per constant or operator).

Note that the rules for constants, variables, and operator applications have **tacit premises** ($c : t, (x : t) \in E, o : t_1 \rightarrow t_2 \rightarrow t$) that will not show up in derivations but need to be checked for a derivation to be valid. Tacit premises are better known as **side conditions**. Note that the rule for projections also has a side condition ($1 \leq i \leq n$).

Here is a derivation for the above example judgement (we write E for the environment $[x : int, y : bool]$):

$$\frac{\overline{E \vdash y : bool} \quad \overline{E \vdash x : int} \quad \overline{E \vdash 0 : int}}{E \vdash \text{IF } y \text{ THEN } x \text{ ELSE } 0 : int}$$

Note that the subderivations are established by the rules for variables and constants.

The rules for lambda expressions and let expressions **update** the existing environment E with a binding $x : t$. If E doesn't contain a binding for x , the binding $x : t$ is added. Otherwise, the existing binding for x is replaced with $x : t$. For instance,

$$\begin{aligned} [x : t_1, y : t_2], z : t_3 &= [x : t_1, y : t_2, z : t_3] \\ [x : t_1, y : t_2], x : t_3 &= [y : t_2, x : t_3] \end{aligned}$$

We say that an expression e is **well typed** in an environment E if the judgment $E \vdash e : t$ is derivable for some type t . Correspondingly, we say that an expressions e is **ill-typed** in an environment E if there is no type t such that the judgment $E \vdash e : t$ is derivable.

The notion of well-typedness generalizes to declarations and programs. OCaml will only execute well-typed programs.

The rules for lambda expressions and recursive let expressions are special in that they *guess* types for the local variables they introduce.

2 Syntax and Semantics

$$\begin{array}{c}
 \frac{c : t}{E \vdash c : t} \qquad \frac{(x : t) \in E}{E \vdash x : t} \\
 \\
 \frac{o : t_1 \rightarrow t_2 \rightarrow t \quad E \vdash e_1 : t_1 \quad E \vdash e_2 : t_2}{E \vdash e_1 o e_2 : t} \\
 \\
 \frac{E \vdash e_1 : t_2 \rightarrow t \quad E \vdash e_2 : t_2}{E \vdash e_1 e_2 : t} \\
 \\
 \frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : t \quad E \vdash e_3 : t}{E \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 : t} \\
 \\
 \frac{E \vdash e_1 : t_1 \quad \dots \quad E \vdash e_n : t_n}{E \vdash (e_1, \dots, e_n) : t_1 \times \dots \times t_n} \\
 \\
 \frac{E \vdash e : t_1 \times \dots \times t_n \quad 1 \leq i \leq n}{E \vdash \pi_i e : t_i} \\
 \\
 \frac{E \vdash e_1 : t_1 \quad E, x : t_1 \vdash e_2 : t}{E \vdash \text{LET } x = e_1 \text{ IN } e_2 : t} \\
 \\
 \frac{E, x : t_1 \vdash e : t_2}{E \vdash \lambda x. e : t_1 \rightarrow t_2} \\
 \\
 \frac{E, f : t_1 \rightarrow t_2, x : t_1 \vdash e_1 : t_2 \quad E, f : t_1 \rightarrow t_2 \vdash e_2 : t}{E \vdash \text{LET REC } f x = e_1 \text{ IN } e_2 : t}
 \end{array}$$

Figure 2.2: Typing rules

2 Syntax and Semantics

The guessing can be eliminated by adding type specifications for the local variables:

$$\begin{aligned} \lambda x.e &\rightsquigarrow \lambda x:t.e \\ \text{LET REC } f \ x = e_1 \text{ IN } e_2 &\rightsquigarrow \text{LET REC } f \ (x : t_1) : t_2 = e_1 \text{ IN } e_2 \end{aligned}$$

The typing rule for non-recursive let expressions

$$\frac{E \vdash e_1 : t_1 \quad E, x : t_1 \vdash e_2 : t}{E \vdash \text{LET } x = e_1 \text{ IN } e_2 : t}$$

accommodates the local variable x without type guessing since the premise $E \vdash e_1 : t_1$ determines the type for x . This argument assumes that a judgment $E \vdash e : t$ determines t given E and e . We will expand on this argument in the next section when we discuss a type checking algorithm.

From the typing rules for let expressions one can obtain typing rules for declarations and programs.

A system of typing rules is commonly referred to as a **type system**. One says that the typing rules formulate a **type discipline** for a language, and that type checking ensures that programs obey the type discipline.

Here is an example of a type derivation involving tuples and projections:

$$\frac{\frac{\frac{\overline{\square \vdash 1 : int} \quad \overline{\square \vdash \text{true} : bool}}{\square \vdash (1, \text{true}) : int \times bool} \quad \overline{\square \vdash 3 : int}}{\square \vdash ((1, \text{true}), 3) : (int \times bool) \times int}}{\square \vdash \pi_1((1, \text{true}), 3) : int \times bool}$$

The derivation uses the typing rules for constants, tuple expressions, and projections.

Here is an example of a type derivation where t_1 and t_2 are placeholders for arbitrary types:

$$\frac{\frac{\overline{x : t_2 \vdash x : t_2}}{x : t_1 \vdash \lambda x.x : t_2 \rightarrow t_2}}{\square \vdash \lambda x.\lambda x.x : t_1 \rightarrow t_2 \rightarrow t_2}$$

The derivation uses the typing rules for variables and lambda expressions. Note how elegantly the typing rules handle the shadowing of the first argument variable in $\lambda x.\lambda x.x$.

2 Syntax and Semantics

Exercise 2.5.1 For each of the following expressions e construct a derivation $\square \vdash e : t$. Use the placeholders t_1, t_2, t_3 for types to avoid unnecessary commitments to concrete types. Try to give most general types.

- a) $\lambda x.x$
- b) $\lambda xy.x$
- c) $\lambda xx.x$
- d) $\lambda fgx.fx(gx)$
- e) LET REC $fx = fx$ IN f
- f) $\lambda xyz.IF x THEN y ELSE $z$$

Exercise 2.5.2 Extend the grammar for abstract expressions with rules for declarations and programs and design corresponding typing rules. Hint: Use judgments $E \vdash D \Rightarrow E'$ and $E \vdash P \Rightarrow E'$ and model programs with the grammar rule $P ::= \emptyset \mid D P$ (\emptyset is the empty program).

2.6 Type Checking Algorithm

We call an expression **simple** if it carries type specifications for the local variables introduced by lambda expression and recursive let expressions. By this definition every expression containing neither lambda expressions nor recursive let expressions is simple.

Given an environment E and a simple expression e , there is at most one type t such that $E \vdash e : t$. Moreover, there is a **type checking algorithm** that given E and e yields the corresponding type if there is one, and otherwise isolates a subexpression where the type checking fails. We may also say that a type checking algorithm constructs a derivation $E \vdash e : t$ given E and e .

The algorithm exploits that there is exactly one typing rule for every syntactic form. Moreover, the typing rules have the important property that the judgments appearing as premises of a rule contain only subexpressions of the expression appearing in the conclusion of the rule. Consequently, construction of a derivation for a given expression by backward application of the typing rules will always terminate. Moreover, when constructing a derivation for a simple expression, only simple expressions can be reached and hence only typing rules for simple expressions have to be considered.

The typing rules for simple expressions describe a **syntax-directed** type checking algorithm that given an environment E and a simple expressions e checks whether there is a type t such that the judgment $E \vdash e : t$ is derivable. Given a simple expression e , the algorithm considers the uniquely determined typing rule for e and recurses on the instantiated premises of the rule. For instance, given an environment E

2 Syntax and Semantics

and an expression `IF e_1 THEN e_2 ELSE e_3` , the algorithm follows the typing rule for conditionals

$$\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : t \quad E \vdash e_3 : t}{E \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 : t}$$

and recurses on the subproblems (E, e_1) , (E, e_2) , and (E, e_3) . If the algorithm succeeds for the subproblems with the types t_1 , t_2 , and t_3 , it checks the equations $t_1 = \text{bool}$ and $t_2 = t_3$. If the equations are satisfied, the algorithm has constructed a derivation for $E \vdash e : t_2$. If the equations are not satisfied, no such derivation exists and the algorithm fails with one of the following error messages:

1. Expression e_1 does not have type *bool*.
2. Expression e_2 and e_3 have different types.

We now have a type checking algorithm for simple expressions. The algorithm is given by the inference rules for simple expressions. We will eventually program the algorithm in OCaml. An important observation is that the typing rules give us a concise description of the algorithm. If we start to formulate the algorithmic reading of the rules with words, we end up with a lengthy text that is hard to understand, as we have seen at the example of the conditional rule.

Exercise 2.6.1 Give the typing rule for lambda expressions $\lambda x:t. e$ augmented with type specifications. Formulate the algorithmic reading of the rule with words.

Exercise 2.6.2 Give the typing rule for recursive let expressions

$$\text{LET REC } f(x:t_1) : t_2 = e_1 \text{ IN } e_2$$

augmented with type specifications. Formulate the algorithmic reading of the rule with words.

Exercise 2.6.3 For each of the following simple expressions e construct a derivation $E \vdash e : t$. Use the placeholders t_1, t_2, t_3, t_4 for types to avoid unnecessary commitments to concrete types. Try to give most general types.

- | | |
|--------------------------------------|---|
| a) fx | e) $\lambda x:t. fx(gx)y$ |
| b) $fx(gx)y$ | f) <code>IF x THEN y ELSE z</code> |
| c) $\lambda x:t_1. \lambda y:t_2. x$ | g) <code>IF x THEN y ELSE x</code> |
| d) $\lambda x:t_1. \lambda x:t_2. x$ | h) <code>IF x THEN $y + 0$ ELSE z</code> |

Exercise 2.6.4 Test your understanding by giving typing rules for judgments $E \vdash e : t$ and expressions $e ::= x \mid \lambda x.e \mid ee$ without looking at Figure 2.2.

2.7 Free and Local Variables

Given an expression e , we call a variable x **free in e** if x has an occurrence in e that is not in the **scope** of a **variable binder**. For instance, the variable x is free in the expression $(\lambda x.x)x$ since the 3rd occurrence of x is not in the scope of the lambda expression. We call an expression e **open** if some variable occurs **free** in e , and **closed otherwise**. For instance, the expression

$$\text{LET } f = \lambda x.f x \text{ IN } f$$

is open since f occurs free in it (2nd occurrence of f). On the other hand, the expression

$$\text{LET REC } f x = f x \text{ IN } f$$

is closed since the 2nd occurrence of f is in the scope of the recursive let expression. When we speak of the **free variables** of an expressions, we mean that variables that are free in the expression.

We distinguish between **binding occurrences** and **using occurrences** of a variable in an expression. For instance, in the expression

$$\text{LET } f = \lambda x.f x \text{ IN } f$$

the first occurrences of f and x are binding, and all other occurrences of f and x are using.

The typing rules for expressions observe the binding rules for variables. If $E \vdash e : t$ is derivable, we know that at most the variables that are assigned a type by the environment E are free in e . Hence we know that e is **closed** if $\square \vdash e : t$ is derivable for some type t .

Let the letter X denote finite sets of variables. We say that an expression e is **closed in X** if every variable that is free in e is an element of X . It is helpful to consider a derivation system for judgments $X \vdash e$ such that $X \vdash e$ is derivable if and only if e is closed in X . Figure 2.3 gives such a derivation system. We refer to the rules of the system as **binding rules**. The notation X, x stands for the set obtained from X by adding x (i.e., $X \cup \{x\}$). Note that the binding rules can be seen as simplifications of the typing rules, and that the algorithmic reading of the rules is obvious (check whether e is closed in X). Only the rule for variables has a side condition.

Exercise 2.7.1 Give derivations for the judgments $\{y, z\} \vdash (\lambda x.x)y$ and $\emptyset \vdash \lambda x.x$. Explain why the judgment $\{x, y\} \vdash z$ is not be derivable.

2 Syntax and Semantics

$$\begin{array}{c}
 \frac{}{X \vdash c} \quad \frac{x \in X}{X \vdash x} \quad \frac{X \vdash e_1 \quad X \vdash e_2}{X \vdash e_1 \circ e_2} \quad \frac{X \vdash e_1 \quad X \vdash e_2}{X \vdash e_1 e_2} \\
 \\
 \frac{X \vdash e_1 \quad X \vdash e_2 \quad X \vdash e_3}{X \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3} \quad \frac{X \vdash e_1 \quad \cdots \quad X \vdash e_n}{X \vdash (e_1, \dots, e_n)} \quad \frac{X \vdash e}{X \vdash \pi_i e} \\
 \\
 \frac{X \vdash e_1 \quad X, x \vdash e_2}{X \vdash \text{LET } x = e_1 \text{ IN } e_2} \quad \frac{X, x \vdash e}{X \vdash \lambda x. e} \quad \frac{X, f, x \vdash e_1 \quad X, f \vdash e_2}{X \vdash \text{LET REC } f \ x = e_1 \text{ IN } e_2}
 \end{array}$$

Figure 2.3: Binding rules

Exercise 2.7.2 Two expressions are **alpha equivalent** if they are equal up to consistent renaming of local variables. For instance, $\lambda x.x$ and $\lambda y.y$ are alpha equivalent. Consistent renaming of local variables is also known as **alpha renaming**. For each of the following expressions give an alpha equivalent expression such that no variable has more than one binding occurrence.

- a) $(\lambda x.x)(\lambda x.x)$ b) $\lambda x.\lambda x.\lambda x.x$ c) $\text{LET REC } xx = x \text{ IN } x$

2.8 Dynamic Semantics

The dynamic semantics of a programming language defines what effect the execution of a phrase should have. For our language of abstract expressions, the successful execution of an expression should produce a value. One usually refers to the execution of an expression as **evaluation** to emphasize that it will produce a value if it doesn't diverge or lead to an erroneous situation (e.g., division by zero).

In Chapter 1 we explained the evaluation of expressions with a **rewriting model** simplifying expressions by rewriting with defining equations. We now switch to an **environment model** where expressions are evaluated in an environment binding variables to values. The environment model is the standard model for the dynamic semantics of programming languages.

One characteristic aspect of the environment model is the fact that functions appear as values called **closures**. For instance, if we evaluate a lambda expression $\lambda x.e$ in a value environment V , we obtain the closure

$$(x, e, V)$$

consisting of the argument variable x , the expression e , and the value environment V providing the values for the free variables of the abstract-

2 Syntax and Semantics

tion $\lambda x.e$. The name closure is motivated by the fact that a lambda expression becomes a self-contained function once we add an environment providing values for the free variables. Closures don't appear in the rewriting model since there the values for free variables are substituted in as part of rewriting.

Closures for recursive functions take the form

$$(f, x, e, V)$$

where f is the name of the function, x is the argument variable, e is the body of the function, and V is a value environment providing values for the free variables of e other than f and x . Only when the recursive closure is applied to an argument, the bindings for f and x will be added to the environment V .

We describe the evaluation of expressions with a derivation system for **evaluation judgments** of the form

$$V \vdash e \triangleright v$$

saying that in the value environment V the expression e evaluates to the value v . A **value environment** is a finite collection of bindings $x \triangleright v$ mapping variables to values. As for type environments, we assume that value environments are functional.

Values are either values of the base types (numbers, booleans, strings), or tuples of values, or closures as described above.

The inference rules for evaluation judgments, commonly called **evaluation rules**, appear in Figure 2.4. The structure of the evaluation rules is similar to the structure of the typing rules. As one would expect, every evaluation rule has an algorithmic reading. For conditionals we have two rules accommodating the two possible branchings. For function applications we also have two rules accommodating separately plain closures and recursive closures. Both rules evaluate the expression of the closure in the environment of the closure updated with a binding of the argument variable to the value serving as actual argument. In the recursive case also a binding of the function variable to the closure is added to enable recursion.

The rule for constants assumes that constants are values, which is fine at the level of abstract syntax. Recall that constants range over numbers, booleans, and strings. For operators we assume that we have a table

$$v_1 \circ v_2 = v$$

fixing the results of successful operator applications.

2 Syntax and Semantics

$$\begin{array}{c}
 \frac{}{V \vdash c \triangleright c} \qquad \frac{(x \triangleright v) \in V}{V \vdash x \triangleright v} \\
 \\
 \frac{V \vdash e_1 \triangleright v_1 \quad V \vdash e_2 \triangleright v_2 \quad v_1 \circ v_2 = v}{V \vdash e_1 \circ e_2 \triangleright v} \\
 \\
 \frac{V \vdash e_1 \triangleright \text{true} \quad V \vdash e_2 \triangleright v}{V \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \triangleright v} \qquad \frac{V \vdash e_1 \triangleright \text{false} \quad V \vdash e_3 \triangleright v}{V \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \triangleright v} \\
 \\
 \frac{V \vdash e_1 \triangleright v_1 \quad \cdots \quad V \vdash e_n \triangleright v_n}{V \vdash (e_1, \dots, e_n) \triangleright (v_1, \dots, v_n)} \\
 \\
 \frac{V \vdash e \triangleright (v_1, \dots, v_n) \quad 1 \leq i \leq n}{V \vdash \pi_i e \triangleright v_i} \\
 \\
 \frac{V \vdash e_1 \triangleright v_1 \quad V, x \triangleright v_1 \vdash e_2 \triangleright v}{V \vdash \text{LET } x = e_1 \text{ IN } e_2 \triangleright v} \\
 \\
 \frac{}{V \vdash \lambda x. e \triangleright (x, e, V)} \\
 \\
 \frac{V \vdash e_1 \triangleright (x, e, V') \quad V \vdash e_2 \triangleright v_2 \quad V', x \triangleright v_2 \vdash e \triangleright v}{V \vdash e_1 e_2 \triangleright v} \\
 \\
 \frac{V, f \triangleright (f, x, e_1, V) \vdash e_2 \triangleright v}{V \vdash \text{LET REC } f x = e_1 \text{ IN } e_2 \triangleright v} \\
 \\
 \frac{V \vdash e_1 \triangleright v_1 \quad V \vdash e_2 \triangleright v_2 \quad v_1 = (f, x, e, V') \quad V', f \triangleright v_1, x \triangleright v_2 \vdash e \triangleright v}{V \vdash e_1 e_2 \triangleright v}
 \end{array}$$

Figure 2.4: Evaluation Rules

2 Syntax and Semantics

Note the side condition $v_1 = (f, x, e, V')$ appearing in the application rule for recursive closures. We have this side condition for better readability. It can be eliminated by replacing both occurrences of v_1 with the closure (f, x, e, V') .

The evaluation rules describe the successful evaluation of expressions at an abstract level omitting details that will be added by an implementation. In particular, the rules don't say what happens when an operator application fails (e.g., division by zero).

We say that an expression e is **evaluable** in an environment V if there is a value v such that the judgment $V \vdash e \triangleright v$ is derivable with the evaluation rules.

Here is an example of a derivation of an evaluation judgment:

$$\frac{\overline{\boxed{\vdash} \lambda x. \lambda y. x \triangleright (x, \lambda y. x, \boxed{\vdash})} \quad \overline{\boxed{\vdash} 1 \triangleright 1} \quad \overline{x \triangleright 1 \vdash \lambda y. x \triangleright (y, x, [x \triangleright 1])}}{\boxed{\vdash} (\lambda x. \lambda y. x) 1 \triangleright (y, x, [x \triangleright 1])}$$

The above derivation can be written more concisely by omitting material copied from the conclusion to the premises:

$$\frac{\overline{\dots \triangleright (x, \lambda y. x, \boxed{\vdash})} \quad \overline{\dots \triangleright 1} \quad \overline{x \triangleright 1 \vdash \lambda y. x \triangleright (y, x, [x \triangleright 1])}}{\boxed{\vdash} (\lambda x. \lambda y. x) 1 \triangleright (y, x, [x \triangleright 1])}$$

Note that the evaluation rules for function applications can yield a value for an application $e_1 e_2$ only if both subexpressions e_1 and e_2 are evaluable. This is in contrast to conditionals IF e_1 THEN e_2 ELSE e_3 , which may be evaluable although one of the constituents e_2 and e_3 is not evaluable. This relates to the problem underlying Exercise 1.13.1.

Type Safety

The evaluation rules do not mention types at all. Thus we can execute ill-typed expressions. This may lead to erroneous situations that cannot appear with well-typed expressions, for instance a function application $e_1 e_2$ where e_1 evaluates to a number, or an addition of a closure and a string, the application of a projection to a value that is not a tuple. In contrast, if we evaluate a well-typed expression, this kind of erroneous situations are impossible. This fundamental property is known as **type safety**. Type safety provides for more efficient execution of programming languages, and also makes it easier for the programmer to write correct programs. OCaml is designed as a type-safe language.

Exercise 2.8.1 For the following expressions e find values v such that $\boxed{\vdash} e \triangleright v$ is derivable. Draw the derivations.

2 Syntax and Semantics

$\lambda x_1 \dots x_n. e$	cascaded lambda
$\rightsquigarrow \lambda x_1. \dots . \lambda x_n. e$	
LET $f x_1 \dots x_n = e_1$ IN e_2	lifted lambda
\rightsquigarrow LET $f = \lambda x_1 \dots x_n. e_1$ IN e_2	
LET REC $f x_1 \dots x_n = e_1$ IN e_2	lifted lambda
\rightsquigarrow LET REC $f x = \lambda x_1 \dots x_n. e_1$ IN e_2	
LET $(x, y) = e_1$ IN e_2	tuple pattern
\rightsquigarrow LET $a = e_1$ IN LET $x = \pi_1 a$ IN LET $y = \pi_2 a$ IN e_2	<i>a fresh</i>
$e_1 \ \&\& \ e_2$	lazy and
\rightsquigarrow IF e_1 THEN e_2 ELSE false	
$e_1 \ \parallel \ e_2$	lazy or
\rightsquigarrow IF e_1 THEN true ELSE e_2	
(<i>o</i>)	lifted operator
$\rightsquigarrow \lambda x y. x \ o \ y$	

Figure 2.5: Derived forms with translation rules

- | | |
|---|---|
| a) $(\lambda x. x)1$ | d) LET REC $f x = x$ IN f |
| b) $(\lambda x. \lambda y. x)1$ | e) $(\lambda y. \text{LET REC } f x = x \text{ IN } f) 5$ |
| c) $(\lambda f. \lambda x. f x) (\lambda x. x)$ | |

Exercise 2.8.2 Give evaluation rules for declarations and programs. Hint: Use judgments $V \vdash D \Rightarrow V'$ and $V \vdash P \Rightarrow V'$ and model programs with the grammar $P ::= \emptyset \mid D P$ (\emptyset is the empty program).

2.9 Derived Forms

When describing a programming language, it is useful to distinguish between a **kernel language** and **derived forms**. Derived forms are constructs that are explained by translation to the kernel language. The advantage of this approach is that derived forms need neither typing nor evaluation rules. Figure 2.5 shows several derived forms together with their translations into abstract expressions. Derived forms like these do exist in OCaml.

The translation rule for tuple patterns needs the side condition that

2 Syntax and Semantics

<i>lazy OR</i>		right
<i>lazy AND</i>	&&	right
<i>comparisons</i>	= ≠ < ≤ > ≥	none
<i>arithmetic operations</i>	+ -	left
	· / %	left
<i>unary minus and plus</i>	- +	right
<i>function application</i>	$e_1 e_2$	left

Figure 2.6: Operator precedences and associativities

the transport variable a must be *fresh* (i.e., does not occur otherwise). The freshness condition is needed so that a does not shadow an outer variable used in e_1 . Moreover, the freshness condition disallows $a = x$.

Syntactically, the lazy boolean operations `&&` (and) and `||` (or) act as operators that take their arguments after comparisons. Moreover, `||` takes its arguments after `&&`. Both lazy boolean operators group to the right. Figure 2.6 shows a table showing the **precedence** (in taking arguments) and **associativity** (i.e., grouping direction) of all operators we have seen so far. In OCaml operators are realized following the table, except that comparisons group to the left.

The grouping rules for the typ-forming operators are as follows:

<i>function type former</i>	→	right
<i>tuple type former</i>	×	none

Abstract expressions can express a form of programs that comes close to programs as sequences of declarations. A program now takes the form

$$D_1 \dots D_n \text{ IN } e$$

and is translated into the let expression

$$D_1 \text{ IN } \dots \text{ IN } D_n \text{ IN } e$$

For e we may choose a tuple expression listing all variables declared by the declarations $D_1 \dots D_n$.

Exercise 2.9.1 Give typing and evaluation rules for the lazy boolean operations assuming they are accommodated as abstract expressions.

Exercise 2.9.2 (Tuple patterns) The translation rule for tuple patterns in Figure 2.5 is just given for pairs so that the translation fits into one line. Give the translation rule for triples. Try to give the translation rule for general patterns using the dots notation (\dots).

Exercise 2.9.3 (Recursive lambda expressions) We may extend our language with recursive lambda expressions $\rho f x.e$ describing a recursive function with a local variable f serving as function name. Note that the greek letter ρ takes the place of the greek letter λ .

- a) Explain recursive lambda expressions $\rho f x.e$ as derived form.
- b) Give a typing rule for recursive lambda expressions $\rho f x.e$ assuming they are abstract expressions.
- c) Give an evaluation rule for recursive lambda expressions $\rho f x.e$ assuming they are abstract expressions.

2.10 Summary and Discussion

In this chapter we discussed the structure of programming languages taking a small sublanguage of OCaml as example. A good starting point into this venture is an abstract grammar

$$e ::= c \mid x \mid e_1 \circ e_2 \mid e_1 e_2 \mid \lambda x.e \mid \dots$$

describing expressions as abstract syntactic objects. The grammar identifies atomic expressions (constants, variables) and composite expressions (applications, lambda expressions, etc). Composite expressions come with constituents, which may be expressions again (note the recursive structure of expressions). We speak of nesting of expressions. Composite expressions can be drawn as syntax trees representing nesting and constituents graphically.

When we program, expressions are written as sequences of characters, which are converted into sequences of words, which are then parsed into expressions. The rules for the conversion of character sequences into abstract expressions are fixed by the *concrete syntax* of a language, which refines the *abstract syntax* of the language. The part of the concrete syntax concerned with characters and words is known as lexical syntax. In the lexical representation of an expression, subexpressions can be indicated with parentheses, following a convention familiar from mathematical notation. To reduce the number of parentheses needed for the textual representation of expressions, the concrete syntax of a language fixes operator precedences and grouping rules.

We also have types as abstract syntactic objects:

$$t ::= int \mid t_1 \rightarrow t_2 \mid \dots$$

Given the abstract syntax of a language, we can define the static semantics and the dynamic semantics of the language. The dynamic semantics is concerned with the evaluation of expressions and does not

2 Syntax and Semantics

make use of types. The static semantics defines a type discipline such that the evaluation of well-typed expressions doesn't lead to obviously erroneous situations. The static semantics determines well-typed expressions and their types, and the dynamic semantics determines evaluable expressions and their values. Both semantics employ environments mapping free variables to either types or values. The dynamic semantics represents functional values as so-called closures combining the syntactic description of a function with a value environment providing values for the free variables.

The semantic systems of a language are described as so-called derivation systems consisting of inference rules for syntactic objects called judgments. We speak of typing judgments and typing rules for the static semantics, and of evaluation judgments and evaluation rules for the dynamic semantics. Typing rules and evaluation rules have a *declarative reading* (concerning the naive construction of derivations) and an *algorithmic reading*. The algorithmic reading of the inference rules gives us abstract algorithms that given an expression and an environment compute either the type or the value of the expression provided it exists.

The type checking algorithm given by the typing rules needs to guess the types of the argument variables of lambda expressions. The guessing can be eliminated by using lambda expressions carrying the type of the argument variable:

$$e ::= \dots \mid \lambda x:t.e$$

For recursive function declarations we need an argument type and a result type to avoid guessing:

$$e ::= \dots \mid \text{LET REC } f(x:t_1) : t_2 = e_1 \text{ IN } e_2$$

A good way to think about type checking sees type checking as abstract evaluation of an expression. While evaluation computes values, type checking computes types. There is the connection that an expression of a type t evaluate to a value of type t (so-called type safety). As a consequence, well-typed expressions have the property that there evaluation cannot lead to type conflicts (e.g., addition of a function and a number, or application of a number to a number).

The type checking algorithm given by the typing rules always terminates. This is in contrast to the evaluation algorithm given by the evaluation rules, which may diverge on expressions not identified as evaluable by the evaluation rules. Note however that the evaluation algorithm will always terminate on expressions identified as evaluable by the evaluation

2 Syntax and Semantics

rules. The reason type checking always terminates is that in contrast to evaluation type checking does not unfold function applications.

In later chapters we will implement in OCaml a type checker and an evaluator for the language considered here following the algorithmic reading of the typing and evaluation rules. The main thing we are currently missing for doing this in OCaml are data structures for sequences and syntax trees.

One speaks of *static typing* to emphasize the fact that evaluation does not rely on type information. In fact, evaluation is defined for all expressions, not just well-typed expressions.

There is the possibility to extend an existing syntax with so-called derived forms whose static and dynamic semantics is obtained with a translation to the existing syntax. Examples for derived forms we discussed include lazy boolean operations and cascaded lambda expressions.

Types give us an organizing principle for the language constructs:

- The base type *bool* comes with the boolean constants and the conditional. In the language we consider, conditionals cannot be expressed otherwise.
- Function types come with function applications, lambda expression, and recursive let expressions.
- Tuple types come with tuple expressions and projections. Graded projections π_i^n state the length of the tuple they are operating on and can be expressed in OCaml. Ungraded projections π_i only state the minimal length of the tuple they are operating on and cannot be expressed in OCaml.

Syntax and semantics of programming languages are inherently recursive notions (even in the absence of recursive functions). This also shows in that recursion is a key aspect of grammars and derivation systems, two mathematical tools we were using for describing syntax and semantics.

There are programming language that come with no or just rudimentary type checking (e.g., Scheme and JavaScript). In practice, type checking is of great importance since it identifies erroneous constructions in programs before programs are executed. Identifying erroneous constructions in programs at execution time running them on test data (so-called debugging) requires understanding and cleverness and is very time consuming (i.e., expensive in human resources) in practice.

One important technique the reader learns in this chapter is high-level programming of type checkers and evaluators using judgments and inference rules.

2 Syntax and Semantics

This chapter attempts to discuss the structure of programming languages with sufficient depth. The key notions are abstract syntax, typing, and evaluation, accompanied by grammars and derivation system as descriptive tools. A weakness of the current presentation of the chapter is that we start with a detailed account of concrete syntax, which, it turns out, doesn't matter for the key issues we are interested in. With hindsight we now recommend a more gentle path through the material of this chapter:

- Abstract expressions (grammars, constituents, syntax trees)
- Parentheses (application chaining, operator precedences)
- Derived forms (cascaded lambdas)
- Derivation systems $x \dot{<} y$
- Binding $X \vdash e$
- Typing $E \vdash e : t$
- Evaluation $V \vdash e : v$
- Lexical syntax

The material on lexical syntax can be shortened.

No doubt, this is a demanding chapter for newcomers who have not seen grammars and derivation systems before. In contrast to Chapter 1, where we can rely on established mathematical intuitions for numbers, the discussion of programming language structure in this chapter is a venture into new territory.

3 Polymorphic Functions and Iteration

So far, some functions don't have unique types, as for instance the projection functions for pairs. The problem is solved by giving such functions schematic types called polymorphic types.

Using polymorphic types, we can declare a tail-recursive higher-order function iterating a function on a given value. It turns out that many functions can be obtained as instances of iteration. Our key example will be a function computing the n th prime number.

3.1 Polymorphic Functions

Consider the declaration of a projection function for pairs:

```
let fst (x,y) = x
```

What type does *fst* have? Clearly, $int \times int \rightarrow int$ and $bool \times int \rightarrow bool$ are both types admissible for *fst*. In fact, every type $t_1 \times t_2 \rightarrow t_1$ is admissible for *fst*. Thus there are infinitely many types admissible for *fst*.

OCaml solves the situation by typing *fst* with the polymorphic type

$$\forall \alpha \beta. \alpha \times \beta \rightarrow \alpha$$

A **polymorphic type** is a type scheme whose quantified variables (α and β in the example) can be instantiated with all types. The **instances** of the polymorphic type above are all types $t_1 \times t_2 \rightarrow t_1$ where the types t_1 and t_2 can be freely chosen. When a polymorphically typed identifier is used in an expression, it can be used with any instance of its polymorphic type. Thus *fst*(1, 2) and *fst*(true, 5) are both well-typed expressions.

Here is a polymorphic swap function for pairs:

```
let swap (x,y) = (y,x)
```

OCaml will type *swap* with the polymorphic type

$$\forall \alpha \beta. \alpha \times \beta \rightarrow \beta \times \alpha$$

This is in fact the **most general polymorphic type** that is admissible for *swap*. Similarly, the polymorphic type given for *fst* is the most general polymorphic type admissible for *fst*. OCaml will always derive most general types for function declarations.

3 Polymorphic Functions and Iteration

Exercise 3.1.1 Declare functions admitting the following polymorphic types:

- a) $\forall\alpha. \alpha \rightarrow \alpha$
- b) $\forall\alpha\beta. \alpha \rightarrow \beta \rightarrow \alpha$
- c) $\forall\alpha\beta\gamma. (\alpha \times \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$
- d) $\forall\alpha\beta\gamma. (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \times \beta \rightarrow \gamma$
- e) $\forall\alpha\beta. \alpha \rightarrow \beta$

3.2 Iteration

Given a function $f : t \rightarrow t$, we write $f^n(x)$ for the n -fold application of f to x . For instance, $f^0(x) = x$, $f^1(x) = f(x)$, $f^2(x) = f(f(x))$, and $f^3(x) = f(f(f(x)))$. More generally, we have

$$f^{n+1}(x) = f^n(fx)$$

Given an **iteration** $f^n(x)$, we call f the **step function** and x the **start value** of the iteration.

With iteration we can compute sums, products, and powers of non-negative integers just using additions $x + 1$:

$$\begin{aligned}x + n &= x + 1 \cdots + 1 = (\lambda a. a + 1)^n(x) \\n \cdot x &= 0 + x \cdots + x = (\lambda a. a + x)^n(0) \\x^n &= 1 \cdot x \cdots \cdot x = (\lambda a. a \cdot x)^n(1)\end{aligned}$$

Exploiting commutativity of addition and multiplication, we arrive at the defining equations

$$\begin{aligned}\text{succ } x &:= x + 1 \\ \text{add } x \ n &:= \text{succ}^n(x) \\ \text{mul } n \ x &:= (\text{add } x)^n(0) \\ \text{pow } x \ n &:= (\text{mul } x)^n(1)\end{aligned}$$

We define a polymorphic **iteration operator**

$$\begin{aligned}\text{iter} &: \forall\alpha. (\alpha \rightarrow \alpha) \rightarrow \mathbb{N} \rightarrow \alpha \rightarrow \alpha \\ \text{iter } f \ 0 \ x &:= x \\ \text{iter } f \ (n + 1) \ x &:= \text{iter } f \ n \ (f x)\end{aligned}$$

so that we can obtain iterations $f^n(x)$ as applications $\text{iter } f \ n \ x$ of the operator. Note that the function iter is polymorphic, higher-order, and tail-recursive. In OCaml, we will use the declaration

3 Polymorphic Functions and Iteration

```
let rec iter f n x =  
  if n < 1 then x  
  else iter f (n - 1) (f x)
```

Functions for addition, multiplication, and exponentiation can now be declared as follows:

```
let succ x = x + 1  
let add x y = iter succ y x  
let mul x y = iter (add y) x 0  
let pow x y = iter (mul x) y 1
```

Note that these declarations are non-recursive. Thus termination needs only be checked for *iter*, where it is obvious (2nd argument is decreased).

Exercise 3.2.1 Declare a function testing evenness of numbers by iterating on booleans. What do you have to change to obtain a function checking oddness?

Exercise 3.2.2 We have the equation

$$f^{n+1}(x) = f(f^n(x))$$

providing for an alternative, non-tail-recursive definition of an iteration operator. Give the mathematical definition and the declaration in OCaml of an iteration operator using the above equation.

3.3 Iteration on Pairs

Using iteration and successor as basic operations on numbers, we have defined functions computing sums, products, and powers of nonnegative numbers. We can also define a **predecessor function**¹

$$\begin{aligned} pred &: \mathbb{N}^+ \rightarrow \mathbb{N} \\ pred(n+1) &:= n \end{aligned}$$

just using iteration and successor (the successor of an integer x is $x+1$). The trick is to iterate on pairs. We start with the pair $(0, 0)$ and iterate with a step function f such that $n+1$ iterations yield the pair $(n, n+1)$. For instance, the iteration

$$f^5(0, 0) = f^4(0, 1) = f^3(1, 2) = f^2(2, 3) = f(3, 4) = (4, 5)$$

using the **step function**

$$f(a, k) = (k, k+1)$$

¹the predecessor of an integer x is $x-1$.

3 Polymorphic Functions and Iteration

yields the predecessor of 5 as the first component of the computed pair. More generally, we have

$$(n, n + 1) = f^{n+1}(0, 0)$$

We can now declare a predecessor function as follows:

```
let pred n = fst (iter (fun (a,k) -> (k, succ k)) n (0,0))
```

Iteration on pairs is a powerful computation scheme. Our second example concerns the sequence of **Fibonacci numbers**

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

which is well known in mathematics. The sequence is obtained by starting with 0, 1 and then adding new elements as the sum of the two preceding elements. We can formulate this method as a recursive function

$$\begin{aligned} fib : \mathbb{N} &\rightarrow \mathbb{N} \\ fib(0) &:= 0 \\ fib(1) &:= 1 \\ fib(n + 2) &:= fib(n) + fib(n + 1) \end{aligned}$$

Incidentally, this is the first recursive function we see where the recursion is **binary** (two recursive applications) rather than **linear** (one recursive application). Termination follows with the usual argument that each recursion step decreases the argument.

If we look again at the rule generating the Fibonacci sequence, we see that we can compute the sequence by starting with the pair (0, 1) and iterating with the step function $f(a, b) = (b, a + b)$. For instance,

$$f^5(0, 1) = f^4(1, 1) = f^3(1, 2) = f^2(2, 3) = f(3, 5) = (5, 8)$$

yields the pair $(fib(5), fib(6))$. More generally, we have

$$(fib(n), fib(n + 1)) = (\lambda(a, b).(b, a + b))^n(0, 1)$$

Thus we can declare an iterative Fibonacci function as follows:

```
let fibi n = fst (iter (fun (a,b) -> (b, a + b)) n (0,1))
```

In contrast to the previously defined function fib , function $fibi$ requires only tail recursion as provided by $iter$.

Exercise 3.3.1 Declare a function computing the sum $0 + 1 + 2 + \dots + n$ by iteration starting from the pair (0, 1).

3 Polymorphic Functions and Iteration

Exercise 3.3.2 Declare a function $f : \mathbb{N} \rightarrow \mathbb{N}$ computing the sequence

$$0, 1, 1, 2, 4, 7, 13, \dots$$

obtained by starting with 0, 1, 1 and then adding new elements as the sum of the three preceding elements. For instance, $f(3) = 2$, $f(4) = 4$, and $f(5) = 7$.

Exercise 3.3.3 Functions defined with iteration can always be elaborated into tail-recursive functions not using iteration. If the iteration is on pairs, one can use separate accumulator arguments for the components of the pairs. Follow this recipe and declare a tail-recursive function fib' such that $fib' \ n \ 0 \ 1 = fib(n)$.

Exercise 3.3.4 Recall the definition of factorials $n!$ from Exercise 1.11.1.

- Give a step function f such that $(n!, n) = f^n(1, 0)$.
- Declare a function $faci$ computing factorials with iteration.
- Declare a tail-recursive function fac' such that $fac' \ n \ 1 \ 0 = n!$. Follow the recipe from Exercise 3.3.3.

3.4 Computing Primes

A **prime number** is an integer greater 1 that cannot be obtained as the product of two integers greater 1. There are infinitely many prime numbers. The sequence of prime numbers starts with

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, \dots$$

We want to declare a function $nth_prime : \mathbb{N} \rightarrow \mathbb{N}$ that yields the elements of the sequence starting with $nth_prime \ 0 = 2$. Assuming a primality test $prime : int \rightarrow bool$, we can declare nth_prime as follows:

```
let next_prime x = first prime (x + 1)
let nth_prime n = iter next_prime n 2
```

Note that $next_prime \ x$ yields the first prime greater than x . Also note that the elegant declarations of $next_prime$ and nth_prime are made possible by the higher-order functions $first$ and $iter$.

It remains to come up with a **primality test** (a test checking whether an integer is a prime number). Here are 4 equivalent characterizations of primality of x assuming that x , k , and n are natural numbers:

3 Polymorphic Functions and Iteration

1. $x \geq 2 \wedge \forall k \geq 2. \forall n \geq 2. x \neq k \cdot n$
2. $x \geq 2 \wedge \forall k, 2 \leq k < x. x \% k > 0$
3. $x \geq 2 \wedge \forall k > 1, k^2 \leq x. x \% k > 0$
4. $x \geq 2 \wedge \forall k, 1 < k \leq \sqrt[3]{x}. x \% k > 0$

Characterization (1) is close to the informal definition of prime numbers. Characterizations (2), (3), and (4) are useful for our purposes since they can be realized algorithmically. Starting from (1), we see that $x - 1$ can be used as an upper bound for k and n . Thus we have to test $x = k \cdot n$ only for finitely many k and n , which can be done algorithmically. The next thing we see is that it suffices to have k because n can be kept implicit using the remainder operation. Finally, we see that it suffices to test for k such that $k^2 \leq x$ since we can assume $n \geq k$. Thus we can sharpen the upper bound from $x - 1$ to \sqrt{x} .

Here we choose the second characterization to declare a primality test in OCaml and leave the realization of the computationally faster fourth characterization as an exercise.

To test the bounded universal quantification in the second characterization, we declare a higher-order function

$$\text{forall} : \text{int} \rightarrow \text{int} \rightarrow (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool}$$

such that

$$\text{forall } m \ n \ f = \text{true} \iff \forall k, m \leq k \leq n. f \ k = \text{true}$$

using the defining equations²

$$\text{forall } m \ n \ f := \begin{cases} \text{true} & m > n \\ f \ m \ \&\& \ \text{forall } (m + 1) \ n \ f & m \leq n \end{cases}$$

The function *forall* terminates since $|n + 1 - m| \geq 0$ decreases with every recursion step.

We now define a primality test based on the second characterization:

$$\text{prime } x := x \geq 2 \ \&\& \ \text{forall } 2 \ (x - 1) \ (\lambda k. x \% k > 0)$$

The discussion of primality tests makes it very clear that programming involves mathematical reasoning.

Efficient primality tests are important in cryptography and other areas of computer science. The naive versions we have discussed here are known as trial division algorithms and are too slow for practical purposes.

²We use the curly brace as an abbreviation to write two equations as a single equation.

3 Polymorphic Functions and Iteration

Exercise 3.4.1 Declare a test $exists : int \rightarrow int \rightarrow (int \rightarrow bool) \rightarrow bool$ such that $exists\ m\ n\ f = true \iff \exists k, m \leq k \leq n. fk = true$ in two ways:

- Directly following the design of *forall*.
- Using *forall* and boolean negation $not : bool \rightarrow bool$.

Exercise 3.4.2 Declare a primality test based on the fourth characterization (i.e., upper bound $\sqrt[2]{x}$). Convince yourself with an OCaml interpreter that testing with upper bound $\sqrt[2]{x}$ is much faster on large primes (check 479,001,599 and 87,178,291,199).

Exercise 3.4.3 Explain why the following functions are primality tests:

- $\lambda x. x \geq 2 \ \&\& \ first (\lambda k. x \% k = 0) \ 2 = x$
- $\lambda x. x \geq 2 \ \&\& \ (first (\lambda k. k^2 \geq x \ || \ x \% k = 0) \ 2)^2 > x$

Hint for (b): Let k be the number the application of *first* yields. Distinguish three cases: $k^2 < x$, $k^2 = x$, and $k^2 > x$.

Exercise 3.4.4 Convince yourself that the four characterizations of primality given above are equivalent.

3.5 Polymorphic Typing Rules

We distinguish between **monomorphic** and **polymorphic types**. Polymorphic types (e.g., $\forall \alpha. \alpha \rightarrow \alpha$) are type schemes where one or several type variables are universally quantified. Monomorphic types (e.g., $t \rightarrow t$) may contain type variables but must not contain quantifiers. Polymorphic types are obtained from monomorphic types by quantifying one or several variables. This disallows nesting of polymorphic types into function types or tuple types.

The typing rules from §2.5 can be extended to polymorphic types. We will write t for monomorphic types and T for polymorphic types. A type environment E can now contain both monomorphic bindings $x : t$ and polymorphic bindings $x : T$. The rules from §2.5 stay unchanged except for the now more general E and t . Three new **polymorphic typing rules** are added to handle polymorphic types.

The first polymorphic typing rule provides for the use of polymorphic variables in expressions:

$$\frac{(x : T) \in E \quad T \succ t}{E \vdash x : t}$$

3 Polymorphic Functions and Iteration

The notation $T \succ t$ says that the monomorphic type t is an instance of the polymorphic type T . We can now derive the typing

$$f : \forall\alpha. \alpha \rightarrow \alpha \vdash f : int \rightarrow int$$

The second polymorphic typing rule accommodates let expressions declaring polymorphic functions:

$$\frac{E \vdash e_1 : t_1 \quad E, x : T \vdash e_2 : t \quad T \succ^E t_1 \quad e_1 = \lambda \dots}{E \vdash \text{LET } x = e_1 \text{ IN } e_2 : t}$$

The side condition $T \succ^E t$ says that the monomorphic type t is a most general instance of the polymorphic type T obtained by erasing the quantifier prefix of T after the quantified type variables have been renamed so that they don't occur in E . The side condition $e_1 = \lambda \dots$ says that e_1 must be a lambda expression. We can now derive the typing

$$\square \vdash \text{LET } f = \lambda x. x \text{ IN } (f\ 5, f\ \text{true}) : int \times bool$$

where f is assigned the polymorphic type $\forall\alpha. \alpha \rightarrow \alpha$ and the most general instance $\alpha \rightarrow \alpha$ by the rule for let expressions. The two using occurrences of f are typed with the instances $int \rightarrow int$ and $bool \rightarrow bool$ of $\forall\alpha. \alpha \rightarrow \alpha$.

Note that the two rules for polymorphic types are not algorithmic since they involve guessing of types. It turns out that OCaml comes with a smart type inference algorithm doing all type guessing in an algorithmic way. In fact you can use OCaml to find out for given E and e the most general type t such that $E \vdash e : t$.

Example: Polymorphic self application

As an example we consider the expression

$$\text{LET } f = \lambda x. x \text{ IN } ff$$

which defines f as an identity function. This expression can only be typed if f is given a polymorphic type since $f : t \rightarrow t$ cannot be applied to itself (since $t \neq t \rightarrow t$). However, the polymorphic typing rule for let expressions makes it possible to assign to f the polymorphic type $\forall\alpha. \alpha \rightarrow \alpha$, which makes it possible to type the self application ff with the instances $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ and $\alpha \rightarrow \alpha$. Here is a derivation establishing a most general type for ff (the derivation is given in two parts to fit on the page):

$$\frac{\frac{E, f : \forall\alpha. \alpha \rightarrow \alpha \vdash f : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)}{E, f : \forall\alpha. \alpha \rightarrow \alpha \vdash f : \alpha \rightarrow \alpha} \quad \frac{E, f : \forall\alpha. \alpha \rightarrow \alpha \vdash f : \alpha \rightarrow \alpha}{E, f : \forall\alpha. \alpha \rightarrow \alpha \vdash ff : \alpha \rightarrow \alpha}}$$

3 Polymorphic Functions and Iteration

Like functions, operations must be accommodated with types. So what is the type of the equality test? OCaml follows common mathematical practice and admits the equality test for all types. To this purpose, the operator testing equality is accommodated with a polymorphic type:

$$= : \forall\alpha. \alpha \rightarrow \alpha \rightarrow \text{bool}$$

The polymorphic type promises more than the equality test delivers. There is the general problem that a meaningful equality test for functions cannot be realized computationally. OCaml bypasses the problem by the crude provision that an equality test on functions raises an invalid argument exception.

We assume that functions at the mathematical level always return values and do not raise exceptions. We handle division by zero by assuming that it returns some value, say 0. Similarly, we handle an equality test for functions by assuming that it always returns `true`. When reasoning at the mathematical level, we will avoid situations where the ad hoc definitions come into play, following common mathematical practice.

3.7 Summary

Functions can receive polymorphic types. Polymorphic types are type schemes whose variables serve as placeholders for arbitrary types. A polymorphic function can be used with every instance of its type scheme. Typical examples for polymorphic functions (i.e., functions that are assigned a polymorphic type) are functions for tuples and the iteration operator.

OCaml will assign polymorphic types to declared functions whenever this is possible. This way OCaml's type inference algorithm can infer unique polymorphic types for functions that don't have most general monomorphic types.

Type checking as explained in §2.5 extends to polymorphic types by adding three polymorphic typing rules. Only variables declared with `let` expressions can receive polymorphic types. In particular, lambda expressions cannot receive polymorphic types.

We remark that the dynamic semantics of programs is not affected by polymorphic types since it doesn't know about static types anyway. We may see polymorphic types as a means for making programs well typed that otherwise would not be well typed. An example is the iteration function which can iterate on different types (we have seen iterations on numbers, booleans, and pairs). Another example are graded projections for tuples, which can be used for different component types.

3 Polymorphic Functions and Iteration

Iteration is a tail-recursive computation scheme that can be formulated as a polymorphic higher-order function. Without functional arguments it is impossible to formulate iteration as a function, and without polymorphism we cannot formulate iteration for all iteration types.

4 Lists

Lists are a basic mathematical data structure providing a recursive representation for finite sequences. Lists are essential for programming. OCaml, and functional programming languages in general, accommodate lists in a mathematically clean way. Three interesting problems we will attack with lists are decimal representation, sorting, and prime factorization:

$$\begin{aligned} dec\ 735 &= [7, 3, 5] \\ sort\ [7, 2, 7, 6, 3, 4, 5, 3] &= [2, 3, 3, 4, 5, 6, 7, 7] \\ prime_fac\ 735 &= [3, 5, 7, 7] \end{aligned}$$

4.1 Nil and Cons

A list represents a finite sequence $[x_1, \dots, x_n]$ of values. All **elements** of a list must have the same type. A **list type** $\mathcal{L}(t)$ contains all lists whose elements are of type t ; we speak of **lists over** t . For instance,

$$\begin{aligned} [1, 2, 3] &: \mathcal{L}(\mathbb{Z}) \\ [\text{true}, \text{true}, \text{false}] &: \mathcal{L}(\mathbb{B}) \\ [(\text{true}, 1), (\text{true}, 2), (\text{false}, 3)] &: \mathcal{L}(\mathbb{B} \times \mathbb{Z}) \\ [[1, 2], [3], []] &: \mathcal{L}(\mathcal{L}(\mathbb{Z})) \end{aligned}$$

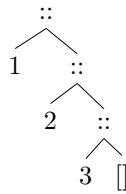
All lists are obtained from the **empty list** $[]$ using the binary constructor **cons** written as “ $::$ ”:

$$\begin{aligned} [1] &= 1 :: [] \\ [1, 2] &= 1 :: (2 :: []) \\ [1, 2, 3] &= 1 :: (2 :: (3 :: [])) \end{aligned}$$

The empty list $[]$ also counts as a constructor and is called **nil**. Given a nonempty list $x :: l$ (i.e., a list obtained with cons), we call x the **head** and l the **tail** of the list.

It is important to see lists as trees. For instance, the list $[1, 2, 3]$ may be depicted as the tree

4 Lists



The tree representation shows how lists are obtained with the constructors `nil` and `cons`. It is important to keep in mind that the bracket notation $[x_1, \dots, x_n]$ is just notation for a list obtained with n applications of `cons` from `nil`. Also keep in mind that every list is obtained with either the constructor `nil` or the constructor `cons`.

Notationally, `cons` acts as an infix operator grouping to the right. Thus we can omit the parentheses in $1 :: (2 :: (3 :: []))$. Moreover, we have $[x_1, \dots, x_n] = x_1 :: \dots :: x_n :: []$.

Given a list $[x_1, \dots, x_n]$, we call the values x_1, \dots, x_n the **elements** or the **members** of the list.

Despite the fact that tuples and lists both represent sequences, tuple types and list types are quite different:

- A tuple type $t_1 \times \dots \times t_n$ admits only tuples of length n , but may fix different types for different components.
- A list type $\mathcal{L}(t)$ admits lists $[x_1, \dots, x_n]$ of any length but fixes a single type t for all elements.

Exercise 4.1.1 Give the types of the following lists and tuples.

- a) $[1, 2, 3]$ c) $[(1, 2), (2, 3)]$ e) $[[1, 2], [2, 3]]$
 b) $(1, 2, 3)$ d) $((1, 2), (2, 3))$

4.2 Basic List Functions

The fact that all lists are obtained with `nil` and `cons` facilitates the definition of basic operations on lists. We start with the definition of a polymorphic function

$$\begin{aligned}
 \mathit{length} &: \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathbb{N} \\
 \mathit{length} \ [] &:= 0 \\
 \mathit{length} \ (x :: l) &:= 1 + \mathit{length} \ l
 \end{aligned}$$

4 Lists

that yields the **length** of a list. We have $length[x_1, \dots, x_n] = n$. Another prominent list operation is **concatenation**:

$$\begin{aligned} @ : \forall \alpha. \mathcal{L}(\alpha) &\rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \\ [] @ l_2 &:= l_2 \\ (x :: l_1) @ l_2 &:= x :: (l_1 @ l_2) \end{aligned}$$

We have $[x_1, \dots, x_m] @ [y_1, \dots, y_n] = [x_1, \dots, x_m, y_1, \dots, y_n]$. We call $l_1 @ l_2$ the **concatenation** of l_1 and l_2 .

How can we define an operation **reversing** lists:

$$rev[x_1, \dots, x_n] = [x_n, \dots, x_1]$$

For instance, $rev[1, 2, 3] = [3, 2, 1]$. To define rev , we need defining equations for `nil` and `cons`. The defining equation for `nil` is obvious, since the reversal of the empty list is the empty list. For `cons` we use the equation $rev(x :: l) = rev(l) @ [x]$ which expresses a basic fact about reversal. This brings us to the following definition of **list reversal**:

$$\begin{aligned} rev : \forall \alpha. \mathcal{L}(\alpha) &\rightarrow \mathcal{L}(\alpha) \\ rev [] &:= [] \\ rev (x :: l) &:= rev(l) @ [x] \end{aligned}$$

We can also define a tail recursive list reversal function. As usual we need an accumulator argument. The resulting function combines reversal and concatenation:

$$\begin{aligned} rev_append : \forall \alpha. \mathcal{L}(\alpha) &\rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \\ rev_append [] l_2 &:= l_2 \\ rev_append (x :: l_1) l_2 &:= rev_append l_1 (x :: l_2) \end{aligned}$$

We have $rev(l) = rev_append l []$. The following trace shows the defining equations of rev_append at work:

$$\begin{aligned} rev_append [1, 2, 3] [] &= rev_append [2, 3] [1] \\ &= rev_append [3] [2, 1] \\ &= rev_append [] [3, 2, 1] \\ &= [3, 2, 1] \end{aligned}$$

The functions defined so far are all defined by **list recursion**.¹ List recursion means that there is no recursion for the empty list, and that

¹List recursion is better known as *structural recursion on lists*.

4 Lists

for every nonempty list $x :: l$ the recursion is on the tail l of the list. List recursion always terminates since lists are obtained from `nil` with finitely many applications of `cons`.

Another prominent list operation is

$$\begin{aligned} \text{map} &: \forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\beta) \\ \text{map } f \ [] &:= [] \\ \text{map } f (x :: l) &:= f x :: \text{map } f l \end{aligned}$$

We have $\text{map } f [x_1, \dots, x_n] = [f x_1, \dots, f x_n]$. We say that map applies the function f **pointwise** to a list. Note that map is defined once more with list recursion.

Finally, we define a function that yields the list of all integers between two numbers m and n (including m and n):

$$\begin{aligned} \text{seq} &: \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathcal{L}(\mathbb{Z}) \\ \text{seq } m \ n &:= \text{IF } m > n \ \text{THEN } [] \ \text{ELSE } m :: \text{seq } (m + 1) \ n \end{aligned}$$

For instance, $\text{seq } -1 \ 5 = [-1, 0, 1, 2, 4, 5]$. This time the recursion is not on a list but on the number m . The recursion terminates since every recursion step decreases $n - m$ and recursion stops once $n - m < 0$.

Exercise 4.2.1 Define a function $\text{null} : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathbb{B}$ testing whether a list is the empty list. Do not use the equality test.

Exercise 4.2.2 Consider the expression $1 :: 2 :: [] @ 3 :: 4 :: []$.

1. Put in the redundant parentheses.
2. Give the list the expression evaluates to in bracket notation.
3. Give the tree representation of the list the expression evaluates to.

Exercise 4.2.3 Decide for each of the following equations whether it is well-typed, and, in case it is well-typed, whether it is true. Assume $l_1, l_2 : \text{List}(\mathbb{N})$.

- a) $1 :: 2 :: 3 = 1 :: [2, 3]$
- b) $1 :: 2 :: 3 :: [] = 1 :: (2 :: [3])$
- c) $l_1 :: [2] = l_1 @ [2]$
- d) $(l_1 @ [2]) @ l_2 = l_1 @ (2 :: l_2)$
- e) $(l_1 :: 2) @ l_2 = l_1 @ (2 :: l_2)$
- f) $\text{map } (\lambda x. x^2) [1, 2, 3] = [1, 4, 9]$
- g) $\text{rev } (l_1 @ l_2) = \text{rev } l_2 @ \text{rev } l_1$

4.3 List Functions in OCaml

Given a mathematical definition of a list function, it is straightforward to declare the function in OCaml. The essential new construct are so-called **match expressions** making it possible to discriminate between empty and nonempty lists. Here is a declaration of a length function:

```
let rec length l =
  match l with
  | [] -> 0
  | x :: l -> 1 + length l
```

The match expression realizes a case analysis on lists using separate **rules** for the empty list and for nonempty lists. The left hand sides of the rules are called **patterns**. The cons pattern applies to nonempty lists and binds the **local variables** x and l to the head and the tail of the list.

Note that the pattern variable l introduced by the second rule of the match shadows the argument variable l in the declaration of *length*. The shadowing could be avoided by using a different variable name for the pattern variable (for instance, l').

Below are declarations of OCaml functions realizing the functions *append*, *rev_append*, and *map* defined before. Note how the defining equations translate into rules of match expressions.

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | x :: l1 -> x :: append l1 l2

let rec rev_append l1 l2 =
  match l1 with
  | [] -> l2
  | x :: l1 -> rev_append l1 (x :: l2)

let rec map f l =
  match l with
  | [] -> []
  | x :: l -> f x :: map f l
```

For each of the function declarations, OCaml infers the polymorphic type we have specified with the mathematical definition.

We remark that OCaml realizes the bracket notation for lists using semicolons to separate elements. For instance, the list $[1, 2, 3]$ is written as `[1; 2; 3]` in OCaml.

OCaml provides predefined functions for lists as **fields** of a predefined **standard module** *List*. For instance:

4 Lists

```
List.length      : 'a list -> int
List.append      : 'a list -> 'a list -> 'a list
List.rev_append  : 'a list -> 'a list -> 'a list
List.rev         : 'a list -> 'a list
List.map         : ('a -> 'b) -> 'a list -> 'b list
```

The above listing uses OCaml notation:

- **Dot notation** is used to name the fields of modules; for instance, *List.append* denotes the field *append* of the module *List*.
- List types $\mathcal{L}(t)$ are written in reverse order as “*t list*”.
- Type variables are written with a leading quote, for instance, “*'a*”
- The quantification prefix of polymorphic types is suppressed, relying on the assumption that all occurring type variables are quantified.

Here are further notational details concerning lists in OCaml:

- List concatenation *List.append* is also available through the infix operator “@”.
- The infix operators “::” and “@” both group to the right, and “::” takes its arguments before “@”. For instance,

$$1 :: 2 :: [3; 4]@[5] \rightsquigarrow (1 :: (2 :: [3; 4]))@[5]$$

- The operators “::” and “@” take their arguments before comparisons and after arithmetic operations.

Exercise 4.3.1 Declare a function *seq* following the mathematical definition in the previous section.

Exercise 4.3.2 (Init) Declare a polymorphic function *init* such that $init\ n\ f = [f(0), \dots, f(n-1)]$ for $n \geq 0$. Note that *n* is the length of the result list. Write your function with a tail-recursive helper function. Make sure your function agrees with OCaml’s predefined function *List.init*. Use *List.init* to declare polymorphic functions that yield lists $[f(m), f(m+1), \dots, f(m+n-1)]$ and $[f(m), f(m+1), \dots, f(n)]$.

Exercise 4.3.3 Declare a function *flatten* : $\forall \alpha. \mathcal{L}(\mathcal{L}(\alpha)) \rightarrow \mathcal{L}(\alpha)$ concatenating the lists appearing as elements of a given list:

$$flatten [l_1, \dots, l_n] = l_1 @ \dots @ l_n @ []$$

For instance, we want $flatten [[1, 2], [], [3], [4, 5]] = [1, 2, 3, 4, 5]$.

Exercise 4.3.4 (Decimal Numbers) With lists we have a mathematical representation for decimal numbers. For instance, the decimal representation for the natural number 1234 is the list [1,2,3,4].

4 Lists

- a) Declare a function $dec : \mathbb{N} \rightarrow \mathcal{L}(\mathbb{N})$ that yields the decimal number for a natural number. For instance, we want $dec\ 1324 = [1, 3, 2, 4]$.
- b) Declare a function $num : \mathcal{L}(\mathbb{N}) \rightarrow \mathbb{N}$ that converts decimal numbers into numbers: $num(dec\ n) = n$.

Hint: Declare num with a tail-recursive function num' such that, for instance,

$$\begin{aligned} num\ [1, 2, 3] &= num'\ [1, 2, 3]\ 0 \\ &= num'\ [2, 3]\ 1 \\ &= num'\ [3]\ 12 \\ &= num'\ []\ 123 = 123 \end{aligned}$$

Exercise 4.3.5 Declare functions

$$\begin{aligned} zip &: \forall \alpha \beta. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\beta) \rightarrow \mathcal{L}(\alpha \times \beta) \\ unzip &: \forall \alpha \beta. \mathcal{L}(\alpha \times \beta) \rightarrow \mathcal{L}(\alpha) \times \mathcal{L}(\beta) \end{aligned}$$

such that

$$\begin{aligned} zip\ [x_1, \dots, x_n]\ [y_1, \dots, y_n] &= [(x_1, y_1), \dots, (x_n, y_n)] \\ unzip\ [(x_1, y_1), \dots, (x_n, y_n)] &= ([x_1, \dots, x_n], [y_1, \dots, y_n]) \end{aligned}$$

4.4 Fine Points About Lists

We speak of the collection of list types $\mathcal{L}(t)$ as a **type family**. We may describe the family of list types with the grammar

$$\mathcal{L}(\alpha) ::= [] \mid \alpha :: \mathcal{L}(\alpha)$$

fixing the **constructors** `nil` and `cons`. Speaking semantically, every value of a list type is obtained with either the constructor `nil` or the constructor `cons`. Moreover, `[]` is a value that is a member of every list type $\mathcal{L}(t)$, and $v_1 :: v_2$ is a value that is a member of a list type $\mathcal{L}(t)$ if the value v_1 is a member of the type t and the value v_2 is a member of the type $\mathcal{L}(t)$.

As it comes to type checking, the constructors `nil` and `cons` are accommodated with polymorphic types:

$$\begin{aligned} [] &: \forall \alpha. \mathcal{L}(\alpha) \\ (::) &: \forall \alpha. \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \end{aligned}$$

OCaml comes with the peculiarity that constructors taking arguments (e.g., `cons`) can only be used when applied to all arguments. We can

4 Lists

bypass this restriction by declaring a polymorphic function applying the cons constructor:

$$\begin{aligned} \text{cons} &: \forall \alpha. \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \\ \text{cons } x \ l &:= x :: l \end{aligned}$$

OCaml provides this function as *List.cons*.

4.5 Membership and List Quantification

We define a polymorphic function that tests whether a value appears as element of a list:

$$\begin{aligned} \text{mem} &: \forall \alpha. \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{B} \\ \text{mem } _ \ [] &:= \text{false} \\ \text{mem } x \ (y :: l) &:= (x = y) \ \|\ \text{mem } x \ l \end{aligned}$$

Note that *mem* tail-recurses on the list argument. Also recall the discussion of OCaml's polymorphic equality test in §3.6. We will write $x \in l$ to say that x is an element of the list l .

The structure of the membership test can be generalized with a polymorphic function that for a test and a list checks whether some element of the list satisfies the test:

$$\begin{aligned} \text{exists} &: \forall \alpha. (\alpha \rightarrow \mathbb{B}) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{B} \\ \text{exists } p \ [] &:= \text{false} \\ \text{exists } p \ (x :: l) &:= p \ x \ \|\ \text{exists } p \ l \end{aligned}$$

The expression

$$\text{exists } (\lambda x. x = 5) : \mathcal{L}(\mathbb{Z}) \rightarrow \mathbb{B}$$

now gives us a test that checks whether a lists of numbers contains the number 5. More generally, we have

$$\text{mem } x \ l = \text{exists } ((=)x) \ l$$

Exercise 4.5.1 Declare *mem* and *exists* in OCaml. For *mem* consider two possibilities, one with *exists* and one without helper function.

Exercise 4.5.2 Convince yourself that *exists* is tail-recursive (by eliminating the derived form `||`).

4 Lists

Exercise 4.5.3 Declare a tail-recursive function

$$\text{forall} : \forall \alpha. (\alpha \rightarrow \mathbb{B}) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{B}$$

testing whether all elements of a list satisfy a given test. Consider two possibilities, one without a helper function, and one with *exists* exploiting the equivalence $(\forall x \in l. p(x)) \iff (\neg \exists x \in l. \neg p(x))$.

Exercise 4.5.4 Declare a function $\text{count} : \forall \alpha. \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{N}$ that counts how often a value appears in a list. For instance, we want $\text{count } 5 [2, 5, 3, 5] = 2$.

Exercise 4.5.5 (Inclusion) Declare a function

$$\text{incl} : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{B}$$

which tests whether all elements of the first list are elements of the second list.

Exercise 4.5.6 (Repeating lists) A list is **repeating** if it has an element appearing at two different positions. For instance, $[2, 5, 3, 5]$ is repeating and $[2, 5, 3]$ is not repeating.

- Declare a function testing whether a list is repeating.
- Declare a function testing whether a list is non-repeating.
- Declare a function that given a list l yields a non-repeating list containing the same elements as l .

4.6 Head and Tail

In OCaml we can declare polymorphic functions

$$\text{hd} : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \alpha$$

$$\text{tl} : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$$

that yield the head and the tail of nonempty lists:

```
let hd l =
  match l with
  | [] -> failwith "hd"
  | x :: _ -> x

let tl l =
  match l with
  | [] -> failwith "tl"
  | _ :: l -> l
```


4 Lists

Both functions raise exceptions when applied to the empty list. Note the use of the underline symbol “_” for pattern variables that are not used in a rule. Also note the use of the predefined function

$$failwith : \forall\alpha. string \rightarrow \alpha$$

raising an exception *Failure s* carrying the string *s* given as argument. Interesting is the polymorphic type of *failwith* making it possible to type an application of *failwith* with whatever type is required by the context of the application.

At the mathematical level we don't admit exceptions. Thus we cannot define a polymorphic function

$$hd : \forall\alpha. \mathcal{L}(\alpha) \rightarrow \alpha$$

since we don't have a value we can return for the empty list over α .

Exercise 4.6.1 Define a polymorphic function *tl* : $\forall\alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ returning the tail of nonempty lists. Do not use exceptions.

Exercise 4.6.2 Declare a polymorphic function *last* : $\forall\alpha. \mathcal{L}(\alpha) \rightarrow \alpha$ returning the last element of a nonempty lists.

4.7 Position Lookup

The **positions** of a list are counted from left to right starting with the number 0. For instance, the list $[5, 6, 5]$ has the positions 0, 1, 2; moreover, the element at position 1 is 6, and the value 5 appears at the positions 0 and 2. The empty list has no position. More generally, a list of length n has the positions $0, \dots, n - 1$.

We declare a tail-recursive *lookup function*

$$nth : \forall\alpha. \mathcal{L}(\alpha) \rightarrow int \rightarrow \alpha$$

that given a list and a position returns the element at the position:

```
let rec nth l n =
  match l with
  | [] -> failwith "nth"
  | x :: l -> if n < 1 then x else nth l (n-1)
```

The function raises an exception if $l = []$ or $n \geq \text{length } l$. Note that *nth l 0* yields the head of *l* if *l* is nonempty. Also note that *nth* terminates since it recurses on the list argument. Here is a trace:

$$nth [0, 2, 3, 5] 2 = nth [2, 3, 5] 1 = nth [3, 5] 0 = 3$$

Exercise 4.7.1 What is the result of $nth\ [0, 2, 3, 5]\ (-2)$?

Exercise 4.7.2 Declare a function $nth_checked$ that raises an invalid-argument exception (see §1.13) if $n < 0$ and otherwise agrees with nth . Check that your function behaves the same as the predefined function $List.nth$.

Exercise 4.7.3 Define a function $pos : \forall\alpha. \mathcal{L}(\alpha) \rightarrow \mathbb{Z} \rightarrow \mathbb{B}$ testing whether a number is a position of a list.

Exercise 4.7.4 Declare a function $find : \forall\alpha. \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{N}$ that returns the first position of a list a given value appears at. For instance, we want $find\ 1\ [3, 1, 1] = 1$. If the value doesn't appear in the list, a failure exception should be raised.

4.8 Option Types

The lookup function for lists

$$nth : \forall\alpha. \mathcal{L}(\alpha) \rightarrow int \rightarrow \alpha$$

raises an exception if the position argument is not valid for the given list. There is the possibility to avoid the exception by changing the result type of nth to an option type

$$nth_opt : \forall\alpha. \mathcal{L}(\alpha) \rightarrow int \rightarrow \mathcal{O}(\alpha)$$

that in addition to the values of α has an extra value **None** that can be used to signal that the given position is not valid. In fact, OCaml comes with a type family

$$\mathcal{O}(\alpha) ::= \text{None} \mid \text{Some } \alpha$$

whose values are obtained with two polymorphic constructors

$$\text{Some} : \forall\alpha. \alpha \rightarrow \mathcal{O}(\alpha)$$

$$\text{None} : \forall\alpha. \mathcal{O}(\alpha)$$

such that **Some** injects the values of a type t into $\mathcal{O}(t)$ and **None** represents the extra value. We can declare a lookup function returning options as follows:

```
let rec nth_opt l n =
  match l with
  | [] -> None
  | x :: l -> if n < 1 then Some x else nth_opt l (n-1)
```

4 Lists

An option may be seen as a list that is either empty or a singleton list $[x]$. In fact, it is possible to replace option types with list types. Doing this gives away information as it comes to type checking.

Exercise 4.8.1 Declare a function *nth_opt_checked* that raises an invalid-argument exception if $n < 0$ and otherwise agrees with *nth_opt*.

Exercise 4.8.2 Declare a function

$$\text{nth_list} : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \text{int} \rightarrow \mathcal{L}(\alpha)$$

that agrees with *nth_opt* but returns a list with at most one element.

Exercise 4.8.3 Declare a function *find_opt* : $\forall \alpha. \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{O}(\mathbb{N})$ that returns the first position of a list a given value appears at. For instance, we want *find_opt* 7 [3, 7, 7] = *Some* 1 and *find_opt* 2 [3, 7, 7] = *None*.

4.9 Generalized Match Expressions

OCaml provides pattern matching in more general form than the basic list matches we have seen so far. A good example for explaining **generalized match expressions** is a function

$$\text{eq} : \forall \alpha. (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{B}$$

testing equality of lists using an equality test for the base type given as argument:

```
let rec eq (p: 'a -> 'a -> bool) l1 l2 =
  match l1, l2 with
  | [], [] -> true
  | x::l1, y::l2 -> p x y && eq p l1 l2
  | _, _ -> false
```

The generalized match expression in the declaration matches on two values and uses a final **catch-all rule**. Evaluation of a generalized match tries the patterns of the rules in the order they are given and commits to the first rule whose pattern matches. The pattern of the final rule will always match and will thus be used if no other rule applies. One speaks of a *catch all rule*.

Note that the above declaration specifies the type of the argument p using a type variable α . Without this specification OCaml would infer the more general type $\forall \alpha \beta. (\alpha \rightarrow \beta \rightarrow \mathbb{B}) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\beta) \rightarrow \mathbb{B}$ for *eq*.

The generalized match in the above declaration translates to a simple match with nested simple matches:

4 Lists

```
match l1 with
| [] ->
  begin match l2 with
  | [] -> true
  | _ :: _ -> false
  end
| x::l1 ->
  begin match l2 with
  | [] -> false
  | y::l2 -> p x y && eq p l1 l2
  end
```

The keywords `begin` and `end` provide a notational variant for a pair (\dots) of parentheses.

The notions of *disjointness* and *exhaustiveness* established for defining equations in §1.6 carry over to generalized match expressions. We will only use exhaustive match expressions but occasionally use non-disjoint match expressions where the order of the rules matters (e.g., catch-all rules). We remark that simple match expressions are always disjoint and exhaustive.

We see generalized match expressions as derived forms that compile into simple match expressions.

OCaml also has match expressions for tuples. For instance,

```
let fst a =
  match a with
  | (x, _) -> x
```

declares a projection function $fst : \forall\alpha\beta. \alpha \times \beta \rightarrow \alpha$ for pairs. In fact, match expressions for tuples are native in OCaml and the uses of tuple patterns we have seen in `let` expressions, lambda abstractions, and declarations all compile into match expressions for tuples.

Patterns in OCaml may also contain numbers and other constants. For instance, we may declare a function that tests whether a list starts with the numbers 1 and 2 as follows:

```
let test l =
  match l with
  | 1 :: 2 :: _ -> true
  | _ -> false
```

Exercise 4.9.1 Declare a function testing whether a list starts with the numbers 1 and 2 just using simple match expressions for lists.

Exercise 4.9.2 Declare a function $swap : \forall\alpha\beta. \alpha \times \beta \rightarrow \beta \times \alpha$ swapping the components of a pair using a simple match expression for tuples.

Exercise 4.9.3 (Maximal element) Declare a function that yields the maximal element of a list of numbers. If the list is empty, a failure exception should be raised.

Exercise 4.9.4 Translate the expression

```
fun l -> match l with
| 0::x::_-> Some x
| x::1::_-> Some x
| _-> None
```

into an expression only using simple matches.

4.10 Sublists

A **sublist** of a list l is obtained by deleting $n \geq 0$ positions of l . For instance, the sublists of $[1, 2]$ are the lists

$$[1, 2], [2], [1], []$$

We observe that the empty list $[]$ has only itself as a sublist, and that a sublist of a nonempty list $x :: l$ is either a sublist of l , or a list $x :: l'$ where l' is a sublist of l . Using this observation, it is straightforward to define a function that yields a list of all sublists of a list:

$$\begin{aligned} pow &: \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\mathcal{L}(\alpha)) \\ pow [] &:= [[]] \\ pow (x :: l) &:= pow l @ map (\lambda l. x :: l) (pow l) \end{aligned}$$

We call $pow l$ the **power list** of l .

A sublist test “ l_1 is sublist of l_2 ” needs a more involved case analysis:

$$\begin{aligned} is_sublist &: \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{B} \\ is_sublist l_1 [] &:= l_1 = [] \\ is_sublist [] (y :: l_2) &:= \text{true} \\ is_sublist (x :: l_1) (y :: l_2) &:= is_sublist (x :: l_1) l_2 \ \&\& \\ &\quad x = y \ \&\& \ is_sublist l_1 l_2 \end{aligned}$$

We remark that a computationally naive sublist test can be obtained with the power list function and the membership test:

$$is_sublist l_1 l_2 = mem l_1 (pow l_2)$$

Exercise 4.10.1 (Graded power list)

Declare a function $gpow : \forall \alpha. \mathbb{N} \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\mathcal{L}(\alpha))$ such that $gpow k l$ yields a list containing all sublists of l of length k .

Exercise 4.10.2 (Prefixes, Segments, Suffixes)

Given a list $l = l_1 @ l_2 @ l_3$, we call l_1 a **prefix**, l_2 a **segment**, and l_3 a **suffix** of l . The definition is such that prefixes are segments starting at the beginning of a list, and suffixes are segments ending at the end of a list. Moreover, every list is a prefix, segment, and suffix of itself.

- Convince yourself that segments are sublists.
- Give a list and a sublist that is not a segment of the list.
- Declare a function that yields a list containing all prefixes of a list.
- Declare a function that yields a list containing all suffixes of a list.
- Declare a function that yields a list containing all segments of a list.

Exercise 4.10.3 (Splits) Given a list $l = l_1 @ l_2$, we call the pair (l_1, l_2) a **split** of l . Declare a function that yields a list containing all splits of a list.

Exercise 4.10.4 Declare a function $filter : \forall \alpha. (\alpha \rightarrow \mathbb{B}) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ that given a test and a list yields the sublist of all elements that pass the test. For instance, we want $filter (\lambda x. x > 2) [2, 5, 1, 5, 2] = [5, 5]$.

4.11 Folding Lists

Consider the list

$$a_1 :: (a_2 :: (a_3 :: []))$$

If we replace cons with $+$ and nil with 0, we obtain the expression

$$a_1 + (a_2 + (a_3 + 0))$$

which evaluates to the sum of the elements of the list. If we replace cons with \cdot and nil with 1, we obtain the expression

$$a_1 \cdot (a_2 \cdot (a_3 \cdot 1))$$

which evaluates to the product of the elements of the list. More generally, we obtain the expression

$$f a_1 (f a_2 (f a_3 b))$$

if we replace cons with a function f and nil with a value b . Even more generally, we can define a function $fold$ such that $fold f l b$ yields the value of the expression obtained from l by replacing cons with f and nil with b :

$$\begin{aligned} fold &: \forall \alpha \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \mathcal{L}(\alpha) \rightarrow \beta \rightarrow \beta \\ fold f [] b &:= b \\ fold f (a :: l) b &:= f a (fold f l b) \end{aligned}$$

4 Lists

We have the following equations:

$$\begin{aligned}
 \text{fold } (+) [x_1, \dots, x_n] 0 &= x_1 + \dots + x_n + 0 \\
 \text{fold } (\lambda ab. a^2 + b) [x_1, \dots, x_n] 0 &= x_1^2 + \dots + x_n^2 + 0 \\
 l_1 @ l_2 &= \text{fold } (::) l_1 l_2 \\
 \text{flatten } l &= \text{fold } (@) l [] \\
 \text{length } l &= \text{fold } (\lambda ab. b + 1) l 0 \\
 \text{rev } l &= \text{fold } (\lambda ab. b @ [a]) l []
 \end{aligned}$$

Folding of lists is similar to iteration with numbers in that both recursion schemes can express many functions without further recursion.

There is a tail-recursive variant *foldl* of *fold* satisfying the equation $\text{foldl } f l b = \text{fold } f (\text{rev } l) b$:

$$\begin{aligned}
 \text{foldl} : \forall \alpha \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \mathcal{L}(\alpha) \rightarrow \beta \rightarrow \beta \\
 \text{foldl } f [] b &:= b \\
 \text{foldl } f (a :: l) b &:= \text{foldl } f l (f a b)
 \end{aligned}$$

One says that *fold* folds a list from the right

$$\text{fold } f [a_1, a_2, a_3] b = f a_1 (f a_2 (f a_3 b))$$

that *foldl* folds a list from the left

$$\text{foldl } f [a_1, a_2, a_3] b = f a_3 (f a_2 (f a_1 b))$$

If the order of the folding is not relevant, the tail-recursive version *foldl* is preferable over *fold*.

OCaml provides the function *fold* as *List.fold_right*. OCaml also provides a function *List.fold_left*, which however varies the argument order of our function *foldl*. To avoid confusion, we will not use *List.fold_left* in this chapter but instead use our function *foldl*.

Exercise 4.11.1 Using *fold*, declare functions that yield the concatenation, the flattening, the length, and the reversal of lists.

Exercise 4.11.2 Using *foldl*, declare functions that yield the length, the reversal, and the concatenation of lists.

Exercise 4.11.3 We have the equations

$$\begin{aligned}
 \text{foldl } f l b &= \text{fold } f (\text{rev } l) b \\
 \text{fold } f l b &= \text{foldl } f (\text{rev } l) b
 \end{aligned}$$

- a) Show that the second equation follows from the first equation using the equation $\text{rev}(\text{rev } l) = l$.
- b) Obtain *fold* from *foldl* not using recursion.
- c) Obtain *foldl* from *fold* not using recursion.

4.12 Insertion Sort

A sequence x_1, \dots, x_n of numbers is called **sorted** if its elements appear in order: $x_1 \leq \dots \leq x_n$. **Sorting** a sequence means to rearrange the elements such that the sequence becomes sorted. We want to define a function

$$\text{sort} : \mathcal{L}(\mathbb{Z}) \rightarrow \mathcal{L}(\mathbb{Z})$$

such that $\text{sort } l$ is a sorted rearrangement of l . For instance, we want $\text{sort } [5, 3, 2, 7, 2] = [2, 2, 3, 5, 7]$.

For now, we only consider sorting for lists of numbers. Later it will be easy to generalize to other types and other orders.

There are different sorting algorithms. Probably the easiest one is **insertion sort**. For insertion sort one first defines a function that inserts a number x into a list such that the result list is sorted if the argument list is sorted.² Now sorting a list l is easy: We start with the empty list, which is sorted, and insert the elements of l one by one. Once all elements are inserted, we have a sorted rearrangement of l . Here are definitions of the necessary functions:

$$\text{insert} : \mathbb{Z} \rightarrow \mathcal{L}(\mathbb{Z}) \rightarrow \mathcal{L}(\mathbb{Z})$$

$$\text{insert } x [] := [x]$$

$$\text{insert } x (y :: l) := \text{IF } x \leq y \text{ THEN } x :: y :: l \text{ ELSE } y :: \text{insert } x l$$

$$\text{isort} : \mathcal{L}(\mathbb{Z}) \rightarrow \mathcal{L}(\mathbb{Z})$$

$$\text{isort} [] := []$$

$$\text{isort } (x :: l) := \text{insert } x (\text{isort } l)$$

Make sure you understand every detail of the definition. We offer a trace:

$$\begin{aligned} \text{isort } [3, 2] &= \text{insert } 3 (\text{isort } [2]) \\ &= \text{insert } 3 (\text{insert } 2 (\text{isort } [])) \\ &= \text{insert } 3 (\text{insert } 2 []) \\ &= \text{insert } 3 [2] = 2 :: \text{insert } 3 [] = 2 :: [3] = [2, 3] \end{aligned}$$

Note that isort inserts the elements of the input list reversing the order they appear in the input list.

²More elegantly, we may say that the insertion function preserves sortedness.

Comparisons are polymorphically typed

Declaring the functions *insert* and *isort* in OCaml is now routine. There is, however, the surprise that OCaml derives polymorphic types for *insert* and *isort* if no type specification is given:

$$\begin{aligned} \textit{insert} &: \forall\alpha. \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \\ \textit{isort} &: \forall\alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \end{aligned}$$

This follows from the fact that OCaml accommodates comparisons with the polymorphic type

$$\forall\alpha. \alpha \rightarrow \alpha \rightarrow \mathbb{B}$$

it also uses for the equality test (§3.6). We will explain later how comparisons behave on tuples and lists. For boolean values, OCaml realizes the order `false < true`. Thus we have

$$\textit{isort} [\text{true}, \text{false}, \text{false}, \text{true}] = [\text{false}, \text{false}, \text{true}, \text{true}]$$

Exercise 4.12.1 Declare a function *sorted* : $\forall\alpha. \mathcal{L}(\alpha) \rightarrow \mathbb{B}$ that tests whether a list is sorted. Use tail recursion. Write the function with a generalized match and show how the generalized match translates into simple matches.

Exercise 4.12.2 Declare a function *perm* : $\forall\alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{B}$ that tests whether two lists are equal up to reordering.

Exercise 4.12.3 (Sorting into descending order)

Declare a function *sort_desc* : $\forall\alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ that reorders a list such that the elements appear in descending order. For instance, we want *sort_desc* [5, 3, 2, 5, 2, 3] = [5, 5, 3, 3, 2, 2].

Exercise 4.12.4 (Sorting with duplicate deletion)

Declare a function *dsort* : $\forall\alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ that sorts a list and removes all duplicates. For instance, *dsort* [5, 3, 2, 5, 2, 3] = [2, 3, 5].

Insertion order

Sorting by insertion inserts the elements of the input list one by one into the empty list. The order in which this is done does not matter for the result. The function *isort* defined above inserts the elements of the input list reversing the order of the input list. If we define *isort* as

$$\textit{isort} l := \textit{fold insert} l []$$

4 Lists

we preserve the insertion order. If we switch to the definition

$$\text{isort } l := \text{foldl insert } l []$$

we obtain a tail-recursive insertion function inserting the elements of the input list in the order they appear in the input list.

Exercise 4.12.5 (Count Tables) Declare a function

$$\text{table} : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha \times \mathbb{N}^+)$$

such that $(x, n) \in \text{table } l$ if and only if x occurs $n > 0$ times in l . For instance, we want

$$\text{table } [4, 2, 3, 2, 4, 4] = [(4, 3), (2, 2), (3, 1)]$$

Make sure *table* lists the count pairs for the elements of l in the order the elements appear in l , as in the example above.

4.13 Generalized Insertion Sort

Rather than sorting lists using the predefined order \leq , we may sort lists using an order given as argument:

$$\text{insert} : \forall \alpha. (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$$

$$\text{insert } p \ x \ [] := [x]$$

$$\text{insert } p \ x \ (y :: l) := \text{IF } p \ x \ y \ \text{THEN } x :: y :: l \ \text{ELSE } y :: \text{insert } p \ x \ l$$

$$\text{gisort} : \forall \alpha. (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$$

$$\text{gisort } p \ l := \text{fold } (\text{insert } p) \ l \ []$$

Now the function *gisort* (\leq) sorts as before in ascending order, while the function *gisort* (\geq) sorts in descending order:

$$\text{gisort } (\geq) \ [1, 3, 3, 2, 4] = [4, 3, 3, 2, 1]$$

When we declare the functions *insert* and *gisort* in OCaml, we can follow the mathematical definitions. Alternatively, we can declare *gisort* using a local declaration for *insert*:

```
let gisort p l =
  let rec insert x l =
    match l with
    | [] -> [x]
    | y :: l -> if p x y then x :: y :: l else y :: insert x l
  in
  foldl insert l []
```

4 Lists

This way we avoid the forwarding of the argument p .

Exercise 4.13.1 Declare a function

$$\text{reorder} : \forall \alpha \beta. \mathcal{L}(\alpha \times \beta) \rightarrow \mathcal{L}(\alpha \times \beta)$$

that reorders a list of pairs such that the first components of the pairs are ascending. If there are several pairs with the same first component, the original order of the pairs should be preserved. For instance, we want $\text{reorder} [(5, 3), (3, 7), (5, 2), (3, 2)] = [(3, 7), (3, 2), (5, 3), (5, 2)]$. Declare reorder as a one-liner using the sorting function gisort .

4.14 Lexical Order

We now explain how we obtain an order for lists over t from an order for the base type t following the principle used for ordering words in dictionaries. We speak of a **lexical ordering**. Examples for the lexical ordering of lists of integers are

$$[] < [-1] < [-1, -2] < [0] < [0, 0] < [0, 1] < [1]$$

The general principle behind the lexical ordering can be formulated with two rules:

- $[] < x :: l$
- $x_1 :: l_1 < x_2 :: l_2$ if either $x_1 < x_2$, or $x_1 = x_2$ and $l_1 < l_2$.

Following the rules, we define a function that yields a test for the lexical order \leq of lists given a test for an order \leq of the base type:

$$\begin{aligned} \text{lex} : \forall \alpha. (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathbb{B} \\ \text{lex } p [] l_2 &:= \text{true} \\ \text{lex } p (x_1 :: l_1) [] &:= \text{false} \\ \text{lex } p (x_1 :: l_1) (x_2 :: l_2) &:= p \ x_1 \ x_2 \ \&\& \\ &\quad \text{IF } p \ x_2 \ x_1 \ \text{THEN } \text{lex } p \ l_1 \ l_2 \ \text{ELSE } \text{true} \end{aligned}$$

Exercise 4.14.1 (Lexical order for pairs) The idea of lexical order extends to pairs and to tuples in general.

- a) Explain the lexical order of pairs of type $t_1 \times t_2$ given orders for the component types t_1 and t_2 .
- b) Declare a function

$$\text{lexP} : \forall \alpha \beta. (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow (\beta \rightarrow \beta \rightarrow \mathbb{B}) \rightarrow \alpha \times \beta \rightarrow \alpha \times \beta \rightarrow \mathbb{B}$$

testing the lexical order of pairs. For instance, we want

$$\text{lexP } (\leq) (\geq) (1, 2) (1, 3) = \text{false}$$

and $\text{lexP } (\leq) (\geq) (0, 2) (1, 3) = \text{true}$.

4.15 Prime Factorization

Every integer greater than 1 can be written as a product of prime numbers; for instance,

$$\begin{aligned} 60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\ 147 &= 3 \cdot 7 \cdot 7 \\ 735 &= 3 \cdot 5 \cdot 7 \cdot 7 \end{aligned}$$

One speaks of the *prime factorization* of a number. Recall that a prime number is an integer greater than 1 that cannot be obtained as the product of two integers greater than 1 (§3.4).

It is straightforward to compute the smallest prime factor of a number $x \geq 2$: We simply search for the first $k \geq 2$ **dividing** x (i.e., $x \% k = 0$). Such a k exists since x divides x . Moreover, the first k dividing x is always prime since otherwise there would be a smaller number greater than 1 dividing k and thus also x .

If we can compute smallest prime factors, we can compute prime factorizations by dividing with the prime factor found and continuing recursively. We realize this algorithm with an OCaml function

$$\text{prime_fac} : \text{int} \rightarrow \mathcal{L}(\text{int})$$

declared as follows:

```
let rec prime_fac x =
  if x < 2 then []
  else let k = first (fun k -> x mod k = 0) 2 in
    if k = x then [x]
    else k :: prime_fac (x / k)
```

We have $\text{prime_fac } 735 = [3, 5, 7, 7]$, for instance.

As is, the algorithm is slow on large prime numbers (try 479,001,599). The algorithm can be made much faster by stopping the linear search for the first k dividing x once $k^2 > x$ since then the first k dividing x is x . Moreover, if we have a least prime factor $k < x$, it suffices to start the search for the next prime factor at k since we know that no number smaller than k divides x . We realize the optimized algorithm as follows:

```
let rec prime_fac' k x =
  if k * k > x then [x]
  else if x mod k = 0 then k :: prime_fac' k (x / k)
  else prime_fac' (k + 1) x
```

We have $\text{prime_fac}' 2 \ 735 = [3, 5, 7, 7]$, for instance. Interestingly, the optimized algorithm is simpler than the naive algorithm we started from

4 Lists

(as it comes to code, but not as it comes to correctness). It makes sense to define a wrapper function for *prime_fac'* ensuring that *prime_fac'* is applied with admissible arguments:

```
let prime_fac x = if x < 2 then [] else prime_fac' 2 x
```

We used several mathematical facts to derive the optimized prime factorization algorithm:

1. If $2 \leq x < k^2$ and no number $2 \leq n \leq k$ divides x , then x is a prime number.
2. If $2 \leq k < x$, and k divides x , and no number $2 \leq n < k$ divides x , then k is the least prime factor of x .
3. If $2 \leq k < x$, and k divides x , and no number $2 \leq n < k$ divides x , then no number $2 \leq n < k$ divides x/k .

The correctness of the algorithm also relies on the fact that the **safety condition**

- $2 \leq k \leq x$ and no number $2 \leq n < k$ divides x

propagates from every initial application of *prime_fac'* to all recursive applications. We say that the safety condition is an **invariant** for the applications of *prime_fac'*.

It suffices to argue the termination of *prime_fac'* for the case that the safety condition is satisfied. In this case the $x - k \geq 0$ is decreased by every recursion step.

Exercise 4.15.1 Give traces for the following applications:

- a) *prime_fac'* 2 7 b) *prime_fac'* 2 8 c) *prime_fac'* 2 15

Exercise 4.15.2 Declare a function that yields the least prime factor of an integer $x \geq 2$. Make sure that at most $\sqrt[3]{x}$ remainder operations are necessary.

Exercise 4.15.3 Declare a primality test using at most $\sqrt[3]{x}$ remainder operations for an argument $x \geq 2$.

Exercise 4.15.4 Dieter Schlau has simplified the naive prime factorization function:

```
let rec prime_fac x =
  if x < 2 then []
  else let k = first (fun k -> x mod k = 0) 2 in
       k :: prime_fac (x / k)
```

Explain why Dieter's function is correct.

4.16 Key-Value Maps

Recall the use of environments in the typing and evaluation systems for expressions we discussed in Chapter 2. These environments are modeled as sequences of pairs binding variables to types or values, as for instance in the value environment $[x \triangleright 5, y \triangleright 7, z \triangleright 2]$. Abstractly, we can see an environment as a list of pairs

$$(key, value)$$

consisting of a *key* and a *value*, where the key may be a string or a number representing a variable, and the value may represent a type or an evaluation value. One says that key-value lists are **maps** from keys to values. Following this consideration, we now define **maps** as values of the type family

$$map \alpha \beta := \mathcal{L}(\alpha \times \beta)$$

Recall that the typing and evaluation rules need only two operations

$$\begin{aligned} lookup &: \forall \alpha \beta. map \alpha \beta \rightarrow \alpha \rightarrow \mathcal{O}(\beta) \\ update &: \forall \alpha \beta. map \alpha \beta \rightarrow \alpha \rightarrow \beta \rightarrow map \alpha \beta \end{aligned}$$

on environments, where *lookup* yields the value for a given key provided the environment contains a pair for the key, and *update* updates an environment with a given key-value pair. For instance,

$$\begin{aligned} lookup [(\"x\", 5), (\"y\", 13), (\"z\", 2)] \"y\" &= 13 \\ update [(\"x\", 5), (\"y\", 7), (\"z\", 2)] \"y\" 13 &= [(\"x\", 5), (\"y\", 13), (\"z\", 2)] \end{aligned}$$

The defining equations for *lookup* and *update* are as follows:

$$\begin{aligned} lookup [] a &:= \text{None} \\ lookup ((a', b) :: l) a &:= \text{IF } a' = a \text{ THEN Some } b \\ &\quad \text{ELSE } lookup \ l \ a \\ update [] a b &:= [(a, b)] \\ update ((a', b') :: l) a b &:= \text{IF } a' = a \text{ THEN } (a, b) :: l \\ &\quad \text{ELSE } (a', b') :: update \ l \ a \ b \end{aligned}$$

Exercise 4.16.1 Give the values of the following expressions:

- $update (update (update [] \"x\" 7) \"y\" 2) \"z\" 5$
- $lookup (update l \"x\" 13) \"x\"$
- $lookup (update l a 7) a$

4 Lists

Exercise 4.16.2 Decide for each of the following equations whether it is true in general.

- a) $lookup (update\ l\ a\ b)\ a = Some\ b$
- b) $lookup (update\ l\ a'\ b)\ a = lookup\ l\ a$ if $a' \neq a$
- c) $update (update\ l\ a\ b)\ a'\ b' = update (update\ l\ a'\ b')\ a\ b$ if $a \neq a'$
- d) $lookup (update (update\ l\ a\ b)\ a'\ b')\ a = Some\ b$ if $a \neq a'$

Exercise 4.16.3 (Boundedness) Declare a function

$$bound : \forall \alpha \beta. map\ \alpha\ \beta \rightarrow \alpha \rightarrow bool$$

that checks whether a map binds a given key. Note that you can define *bound* using *lookup*.

Exercise 4.16.4 (Deletion) Declare a function

$$delete : \forall \alpha \beta. map\ \alpha\ \beta \rightarrow \alpha \rightarrow map\ \alpha\ \beta$$

deleting the entry for a given key. We want $lookup (delete\ l\ a)\ a = None$ for all environments l and all keys a .

Exercise 4.16.5 (Maps with memory) Note that *lookup* searches maps from left to right until it finds a pair with the given key. This opens up the possibility to keep previous values in the map by modifying *update* so that it simply appends the new key-value pair in front of the list:

$$update\ l\ a\ b := (a, b) :: l$$

Redo the previous exercises for the new definition of *update*. Also define a function

$$lookup_all : \forall \alpha \beta. map\ \alpha\ \beta \rightarrow \alpha \rightarrow \mathcal{L}(\beta)$$

that yields the list of all values for a given key.

4 Lists

Exercise 4.16.6 (Maps as functions) Maps can be realized with functions if all maps are constructed from the empty map

$$\text{empty} : \forall \alpha \beta. \text{map } \alpha \beta$$

with *update* and the only thing that matters is that *lookup* yields the correct results. Assume the definition

$$\text{map } \alpha \beta := \alpha \rightarrow \text{Some } \beta$$

and define *empty* and the operations *update* and *lookup* accordingly. Test your solution with

$$\text{lookup } (\text{update } (\text{update } (\text{update } \text{empty } \text{"y"} 7) \text{"x"} 2) \text{"y"} 5) \text{"y"} = \text{Some } 5$$

Note that you can still define the operations *bound* and *delete* from Exercises 4.16.1 and 4.16.2.

5 Constructor Types, Trees, and Linearization

Trees are a mathematical data structure generalizing lists. Like lists, trees are indispensable for programming, no matter what programming language is used. In functional programming languages, trees are realized with constructors, and tree types are accommodated as constructor types. With list types and option types we have already seen two instances of constructor types. It turns out that the objects of abstract syntax can be elegantly accommodated as the values of constructor types.

Most of the chapter is devoted to the linearization of trees as strings as it relates abstract and concrete syntax. Besides prefix and postfix linearizations we study infix linearizations with operator precedence and associativity.

5.1 Constructor Types

Recall that the elements of list types are obtained with two constructors `nil` and `cons`. Similarly, the elements of option types are obtained with the constructors `None` and `Some`. We say that list types and option types are **constructor types**. We can describe both type families with grammars:

$$\begin{aligned}\mathcal{L}(\alpha) &::= [] \mid \alpha :: \mathcal{L}(\alpha) \\ \mathcal{O}(\alpha) &::= \text{None} \mid \text{Some } \alpha\end{aligned}$$

We can also see the type \mathbb{B} as a constructor type:

$$\mathbb{B} ::= \text{false} \mid \text{true}$$

Moreover, we can see the family of pair types as a family of constructor types:

$$\alpha \times \beta ::= (\alpha, \beta)$$

Simple matches are available for every constructor type. We have seen simple matches for list types and option types before. It turns out that we can express conditionals with simple matches for booleans:

$$\begin{aligned}\text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 &\rightsquigarrow \text{MATCH } e_1 \text{ WITH} \\ &\quad | \text{true} \rightarrow e_2 \\ &\quad | \text{false} \rightarrow e_3\end{aligned}$$

5 Constructor Types, Trees, and Linearization

Note that the translation to boolean matches explains nicely that not both e_2 and e_3 are evaluated and that e_1 must be evaluated first. We can also reduce pair patterns to simple matches for pairs:

$$\text{LET } (x, y) = e_1 \text{ IN } e_2 \quad \rightsquigarrow \quad \text{MATCH } e_1 \text{ WITH} \\ | (x, y) \rightarrow e_2$$

5.2 AB-Trees

Constructor types are a basic mathematical construction. In OCaml, and other functional programming languages as well, constructor types can be freely defined. It turns out that constructor types can serve as types for syntactic objects. We start with a simple class of syntactic objects we call **AB-trees**:

$$\text{tree} ::= A \mid B(\text{tree}, \text{tree})$$

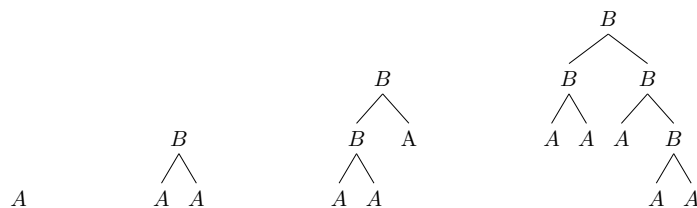
As the definition says, AB-trees are obtained with two constructors

$$A : \text{tree} \\ B : \text{tree} \rightarrow \text{tree} \rightarrow \text{tree}$$

Here are expressions describing values of type *tree*:

$$A \quad B(A, A) \quad B(B(A, A), A) \quad B(B(A, A), B(A, B(A, A)))$$

The graphical tree representations of these values look as follows:



In OCaml we can declare AB-trees as follows:

```
type tree = A | B of tree * tree
```

We remark that in OCaml the constructor B taking arguments is not provided as a function but is only available as part of constructor applications $B(e_1, e_2)$. OCaml imposes the rule that the names of the constructors introduced by a constructor type declaration must start with a capital letter.

5 Constructor Types, Trees, and Linearization

Based on the graphical notation one speaks of the **nodes** and the **edges** of a tree. The nodes are the occurrences of the constructors in the drawing, and the edges are the lines between the nodes. The **root** of a tree is the topmost node of the tree, and the **leaves** of a tree are the bottom-most nodes of the tree. The **size** of a tree is the number of nodes in its drawing, and the **depth** of a tree is the maximal number of edges on a path from the root to a leaf (only proceeding downwards).

Here is a function yielding the size of an AB-tree, defined both mathematically and in OCaml:

$$\begin{aligned} \text{size} : \text{tree} &\rightarrow \mathbb{N} \\ \text{size } A &:= 1 \\ \text{size } (B \ t_1 \ t_2) &:= 1 + \text{size } t_1 + \text{size } t_2 \end{aligned}$$

```
let rec size t =
  match t with
  | A -> 1
  | B(t1,t2) -> 1 + size t1 + size t2
```

Note that in the mathematical definition we take the freedom to write a constructor application $B(t_1, t_2)$ like a function application as $B t_1 t_2$. OCaml, on the other side, insists that constructor applications are written with parentheses and commas (e.g., $B(t_1, t_2)$).

Exercise 5.2.1 Define and declare a function $\text{depth} : \text{tree} \rightarrow \mathbb{N}$ that yields the depth of a *tree*. For instance, we want $\text{depth}(B(BAA)A) = 2$.

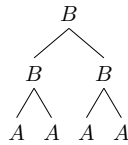
Exercise 5.2.2 Define and declare a function $\text{breadth} : \text{tree} \rightarrow \mathbb{N}$ that yields the number of leaves a tree has.

Exercise 5.2.3 Define and declare a function $\text{mirror} : \text{tree} \rightarrow \text{tree}$ that yields a tree whose graphical representation is the mirror image of the given tree. For instance, we want $\text{mirror}(B(BAA)A) = BA(BAA)$ and $\text{mirror}(B(BA(BAA))A) = BA(B(BAA)A)$.

Exercise 5.2.4 Define and declare a function $\text{dtree} : \mathbb{N} \rightarrow \text{tree}$ that for n yields a tree of depth n and size $2n + 1$.

Exercise 5.2.5 (Maximal Trees)

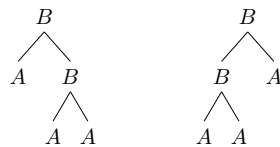
A tree is *maximal* if its size is maximal for all trees of the same depth. The unique maximal tree of depth 2 looks as follows:



- a) Convince yourself that the two subtrees of a maximal tree are maximal and identical.
- b) Declare a function $mtree : \mathbb{N} \rightarrow tree$ that yields the unique maximal tree of depth n . Use a let expression to avoid binary recursion. Also give a tail-recursive function such that $mtree' n A$ yields the maximal tree of depth n .
- c) Give a non-maximal tree t such that $mirror(t) = t$.
- d) Declare a function $check : tree \rightarrow \mathcal{O}(\mathbb{N})$ such that $check(t) = \text{Some}(n)$ if t is a maximal tree of depth n , and $check(t) = \text{None}$ if t is not maximal.

Exercise 5.2.6 (Minimal Trees)

A tree is *minimal* if its size is minimal for all trees of the same depth. The two minimal trees of depth 2 look as follows:



- a) Declare a function $mtree : \mathbb{N} \rightarrow tree$ that yields a minimal tree of depth n . Also give a tail-recursive function such that $mtree' n A$ yields a minimal tree of depth n .
- b) Declare a function $check : tree \rightarrow \mathcal{O}(\mathbb{N})$ such that $check(t) = \text{Some}(n)$ if t is a minimal tree of depth n , and $check(t) = \text{None}$ if t is not minimal.

5.3 Prefix, Postfix, and Infix Linearization

OCaml gives us a linear representation for AB-trees using parentheses and commas; for instance, $B(B(A, A), A)$. There are more elegant linear

5 Constructor Types, Trees, and Linearization

representations for AB-trees not requiring parentheses and commas:

$B(B(A, A), A)$	OCaml linearization
$BBAAA$	prefix linearization
$AABAB$	postfix linearization

Prefix linearizations of AB-trees are obtained by deleting all parentheses and commas in the OCaml linearization, and postfix linearizations are obtained by additionally moving the constructor B behind the associated subtree linearizations. Formally, we define prefix linearizations and postfix linearizations with two functions:

$$\begin{aligned}
 pre &: tree \rightarrow string \\
 pre(A) &:= "A" \\
 pre(B(t_1, t_2)) &:= "B" \wedge pre(t_1) \wedge pre(t_2) \\
 \\
 post &: tree \rightarrow string \\
 post(A) &:= "A" \\
 post(B(t_1, t_2)) &:= post(t_1) \wedge post(t_2) \wedge "B"
 \end{aligned}$$

The caret symbol “ \wedge ” denotes the **concatenation operator**

$$\wedge : string \rightarrow string \rightarrow string$$

for strings. For instance, we have “Dieter” \wedge “Schlau” = “DieterSchlau”.

It turns out that one can define functions $depre$ and $depost$ inverting the linearization functions pre and $post$:

$$depre(pre(t)) = t \qquad \text{depost(post(t))} = t$$

We may say that the functions reconstruct trees from their string representations. The algorithms behind the functions are fundamental for the processing of programming languages in general and will be discussed in a later chapter.

An *infix linearization* of an AB-tree puts the symbol for B between the descriptions of the subtrees. For instance, $(ABA)BA$ is an infix linearization of $B(B(A, A), A)$. Infix linearization must use parentheses for some trees, even if they treat B as a left-associative or a right-associative *syntactic operator*. The most basic infix linearization puts parentheses around every linearization of a tree obtained with the constructor B :

$$\begin{aligned}
 infix &: tree \rightarrow string \\
 infix(A) &:= "A" \\
 infix(B(t_1, t_2)) &:= "(" \wedge infix(t_1) \wedge "B" \wedge infix(t_2) \wedge ")"
 \end{aligned}$$

5 Constructor Types, Trees, and Linearization

The process of translating string representations into tree representations is known as **parsing**. We will see parsing methods for prefix, postfix, and infix linearizations in a later chapter. We call a linearization scheme *invertible* if there is a function translating back the linearization of a tree into the tree. Formally, we want the equation

$$\text{parse}(\text{lin}(t)) = t$$

for all trees t , where $\text{lin} : \text{tree} \rightarrow \text{string}$ is a linearization function and $\text{parse} : \text{string} \rightarrow \text{tree}$ is the inverting parsing function.

Exercise 5.3.1 Declare a function $\text{tree} \rightarrow \text{string}$ linearizing AB-trees in OCaml notation.

Exercise 5.3.2 Declare a function $\mathcal{L}(\mathbb{B}) \rightarrow \text{string}$ linearizing boolean lists using bracket notation.

Exercise 5.3.3 Declare a function $\text{int} \rightarrow \text{string}$ linearizing numbers. As it comes to strings, all you need are string concatenation and string constants (e.g., "5"). First write a function converting digits $0, \dots, 9$ into singleton strings "0", \dots , "9".

Exercise 5.3.4 Declare a function $\mathcal{L}(\text{int}) \rightarrow \text{string}$ linearizing lists of integers using bracket notation.

5.4 Infix Linearization with Dropped Parentheses

Here are two mutually recursive functions $\text{tree} \rightarrow \text{string}$ linearizing AB-trees with a minimum of parentheses given that B is accommodated as a left-associative infix operator:

$$\begin{aligned} \text{tree}(A) &:= "A" \\ \text{tree}(B(t_1, t_2)) &:= \text{tree}(t_1) \wedge "B" \wedge \text{ptree}(t_2) \\ \text{ptree}(A) &:= "A" \\ \text{ptree}(B(t_1, t_2)) &:= "(" \wedge \text{tree}(B(t_1, t_2)) \wedge ")" \end{aligned}$$

We have $\text{tree}(B(B(A, A), A)) = "ABABA"$ and $\text{tree}(B(A, B(A, A))) = "AB(ABA)"$. Both functions linearize trees, where B-trees are linearized either with parentheses (*ptree*) or without parentheses (*tree*). In either case, the left subtree of a B-tree is linearized without parentheses and the right subtree is linearized with parentheses.

5 Constructor Types, Trees, and Linearization

We can describe the **linearization functions** *tree* and *ptree* elegantly with a grammar:

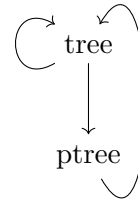
$$\begin{aligned} tree & ::= "A" \mid tree "B" ptree \\ ptree & ::= "A" \mid "(" tree ")" \end{aligned}$$

The linearization reading of the grammar is as follows:

- If *tree* is given a tree A , it yields the string "A". If *tree* is given a tree $B(t_1, t_2)$, it puts "B" in between the linearizations of t_1 and t_2 , where t_1 is linearized with *tree* and t_2 is linearized with *ptree*.
- If *ptree* is given a tree A , it yields the string "A". Otherwise, *ptree* delegates to *tree* and encloses the result into parentheses.

In OCaml the mutual recursion can be expressed as

```
let rec tree t = match t with
  | A -> "A"
  | B(t1,t2) -> tree t1 ^ "B" ^ ptree t2
and ptree t = match t with
  | A -> "A"
  | t -> "(" ^ tree t ^ ")"
```



The graph diagram on the right depicts the mutual recursion present in the linearization functions and in the grammar. We remark that OCaml has separate name spaces for types and functions. Thus we can use the identifier *tree* both for the type of AB-trees and a linearization function.

It is possible to eliminate the mutual recursion by localizing the helper function *ptree*:

```
let rec tree t =
  let ptree t = match t with
    | A -> "A"
    | t -> "(" ^ tree t ^ ")"
  in match t with
  | A -> "A"
  | B(t1,t2) -> tree t1 ^ "B" ^ ptree t2
```

It is straightforward to change the **linearization grammar** for AB-trees such that B is treated as a right-associative syntactic operator:

$$\begin{aligned} tree & ::= "A" \mid ptree "B" tree \\ ptree & ::= "A" \mid "(" tree ")" \end{aligned}$$

Exercise 5.4.1 Declare a function linearizing AB-trees by treating B as a right-associative syntactic operator. For instance, we want $B(A, B(A, A)) \rightsquigarrow "ABABA"$.

5.5 ABC-Trees

We now consider trees with two binary constructors:

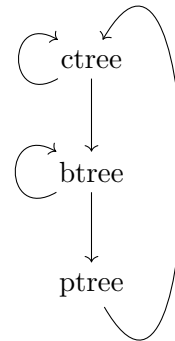
```
type ctree = A | B of ctree * ctree | C of ctree * ctree
```

We leave the prefix, postfix, and fully parenthesized infix linearizations of ABC-trees as exercises since they are routine extensions of the respective linearizations of AB-trees. What is interesting however is the infix linearization of ABC-trees omitting parentheses. Here is a linearization grammar such that B takes its arguments before C , and both B and C are left-associative:

$$\begin{aligned} \text{ctree} &::= \text{ctree } "C" \text{ btree} \mid \text{btree} \\ \text{btree} &::= \text{btree } "B" \text{ ptree} \mid \text{ptree} \\ \text{ptree} &::= "A" \mid "(" \text{ ctree } ")" \end{aligned}$$

You may think of B as multiplication and C as addition. Note that all three rules of the grammar rely on *delegation* to another rule, which can be realized directly with generalized matches:

```
let rec ctree t = match t with
  | C(t1,t2) -> ctree t1 ^ "C" ^ btree t2
  | t -> btree t
and btree t = match t with
  | B(t1,t2) -> btree t1 ^ "B" ^ ptree t2
  | t -> ptree t
and ptree t = match t with
  | A -> "A"
  | t -> "(" ^ ctree t ^ ")"
```



The graph diagram on the right depicts the mutual recursion present in the linearization functions and in the grammar. The diagram may be seen as a graphical representation of the grammar. Note that there are 3 levels, where the 2 upper levels accommodate the syntactic infix operators.

There is the interesting possibility to linearize B-trees with **juxtaposition**, that is, without including “B” into the linearization.¹ This is the way functional programming languages accommodate function applications. With juxtaposition you may think of B as function application

¹In Mathematics, products are often written with juxtaposition; e.g., $2n$ for $2 \cdot n$.

5 Constructor Types, Trees, and Linearization

and of C as addition. We can switch to juxtaposition for B by changing the grammar rule for $btree$ as follows:

$$btree ::= btree\ ptree \mid ptree$$

This preserves the left associativity of B .

Exercise 5.5.1 Declare functions that for ABC-trees yield

- the prefix linearization.
- the postfix linearization.
- the fully parenthesized infix linearization.

Exercise 5.5.2 Declare functions that for ABC-trees yield the infix linearization where B takes its arguments before C and

- both B and C are right-associative.
- B is left-associative and C is right-associative.
- neither B nor C is associative.

In each case give a linearization grammar first.

Exercise 5.5.3 Declare functions that for ABC-trees yield the infix linearization where B takes its arguments before C , both B and C are treated as left-associative syntactic operators, and B is accommodated with juxtaposition.

Exercise 5.5.4 Declare functions that for ABC-trees yield the infix linearization where both B and C are left-associative and take their arguments at the same level. Think of B and C as addition and subtraction. Give a linearization grammar first.

Exercise 5.5.5 Note that the grammar in §5.4 does not use delegation in the first rule for $tree$. Give an equivalent linearization grammar using delegation in both rules.

5.6 Mini-OCaml

Constructor types are the mathematical device for formalizing abstract syntax. We consider a class of abstract expressions given by the following grammar (similar to the grammar in §2.4):

$$\begin{aligned} e ::= & x \mid c \mid e_1 o e_2 \mid e_1 e_2 \\ & \mid \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \\ & \mid \lambda x. e \\ & \mid \text{LET } x = e_1 \text{ IN } e_2 \\ & \mid \text{LET REC } f \ x = e_1 \text{ IN } e_2 \end{aligned}$$

5 Constructor Types, Trees, and Linearization

The letter c ranges over *constants* which we choose to be booleans or integers, the letters x and f range over *variables* which we choose to be strings, and the letter o ranges over the *operators* ' \leq ', '+', '-', and '.'. As discussed before, we see abstract expressions as the abstract syntax of a sublanguage of OCaml we call Mini-OCaml.

The grammar for abstract expressions translates into a constructor type declaration in OCaml as follows:

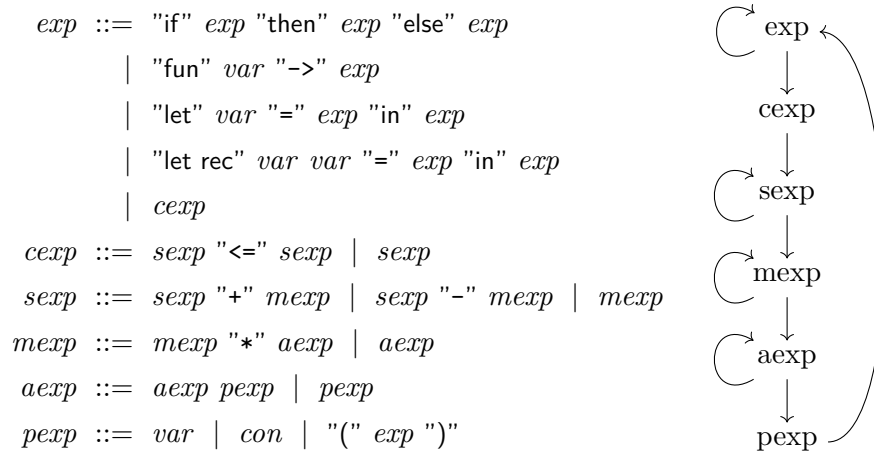
```
type var = string
type con = Bcon of bool | Icon of int
type op  = Add | Sub | Mul | Leq
type exp = Var of var | Con of con
         | Oapp of op * exp * exp
         | Fapp of exp * exp
         | If of exp * exp * exp
         | Lam of var * exp
         | Let of var * exp * exp
         | Letrec of var * var * exp * exp
```

Note the following:

- Variables are represented as strings. The name *var* is not a constructor type but is an additional name for the type *string*.
- Constants are represented as the values of a constructor type *con*. This way constants can be either booleans or integers. A simple match for *con* has 2 rules.
- Operators are represented as the values of a constructor type *op*. A simple match for *op* has 4 rules.
- Expressions are represented as the values of a constructor type *exp*. A simple match for *exp* has 8 rules.

Using the ideas discussed in this chapter, we give a linearization grammar for abstract expressions that is compatible with OCaml's concrete syntax:

5 Constructor Types, Trees, and Linearization



The grammar follows the conventions of OCaml except that the comparison operator is not associative:

- Conditionals, lambda expressions, and let expressions take their arguments after operator and function applications. Hence they must be enclosed in parentheses when they appear as argument expressions of operator and function applications.
- Function applications take their arguments before operator applications.
- Multiplication takes its arguments before addition and subtraction, and all three operators are left-associative.
- Addition and subtraction are at the same level and take their arguments before the comparison operator ' \leq ', which is neither left nor right-associative.

Note that the grammar has 6 levels. The top level accommodates prefix constructs (conditionals and lambda and let expressions), and the bottom level accommodates primitive expressions (variables and constants). Level 2 accommodates function applications, and the remaining 3 levels accommodate the infix operators.

In contrast to the grammars we have seen so far, the linearization functions for the linearization grammar for expressions must put white space characters between the words so that identifiers, keywords, and numbers are properly separated.

OCaml's concrete syntax doesn't have constants for negative integers but represents them with a unary negation operator. There is the subtlety that one can write "2 + -7" but that one can not drop the parentheses in "f(-7)". For our purposes it seems best to linearize negative integers always with parentheses.

5 Constructor Types, Trees, and Linearization

Another feature of OCaml's concrete syntax is the requirement that identifiers used for values must start with lower case letters. Identifiers starting with upper case letters are reserved for constructors.

Exercise 5.6.1 Recall the binding rules from Figure 2.3.

- Declare a function $\mathcal{L}(var) \rightarrow exp \rightarrow bool$ that checks whether a binding judgment $X \vdash e$ is derivable.
- Declare a function $exp \rightarrow bool$ that checks whether an expression is closed.
- Declare a function $exp \rightarrow \mathcal{L}(var)$ that yields a list containing the free variables of an expression.

Exercise 5.6.2 Declare a function $exp \rightarrow string$ that linearizes expressions following the given linearization grammar. Your linearization should be such that the string can serve as OCaml input after removal of the double quotes "", provided only lower-case identifiers are used for variables.

Exercise 5.6.3 Extend the grammar for abstract expressions and the above development with pair expressions and projections:

$$e ::= \dots \mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e$$

As concrete syntax for the projections π_1 and π_2 you may choose the identifiers `fst` and `snd`.

5.7 Natural Numbers as Constructor Type

Consider the constructor type

$$nat ::= O \mid S \ nat$$

introducing two constructors

$$\begin{array}{ll} O : nat & \\ S : nat \rightarrow nat & \text{successor} \end{array}$$

The values of nat

$$O, SO, S(SO), S(S(SO)), \dots$$

may be understood as the natural numbers. We can now easily define addition and multiplication as functions:

$$\begin{array}{ll} + : nat \rightarrow nat \rightarrow nat & \cdot : nat \rightarrow nat \rightarrow nat \\ 0 + n := n & 0 \cdot n := 0 \\ Sm + n := S(m + n) & Sm \cdot n := m \cdot n + n \end{array}$$

5 Constructor Types, Trees, and Linearization

Taken together, we now see that we can define the natural numbers as a constructor type, and the operations on natural numbers as recursive functions. The idea for this setup goes back to the work of Dedekind and Peano. Defining the natural numbers with a constructor type is logically of great importance. On the practical side, however, the naive constructor representation of natural numbers is not helpful since it is hopelessly inefficient.

Exercise 5.7.1 Declare the constructor type *nat* in OCaml and declare functions for addition, multiplication, and exponentiation. Also declare functions converting between nonnegative machine integers and constructor numbers.

5.8 Summary

In this chapter we have learned a lot of fundamental things. Mathematically, trees come into existence through nesting of tuples realized by constructor applications. The fundamental notion of constructor types provides a rich construction scheme for tree types naturally accommodating systems of abstract syntax. Very conveniently, there is not much to learn about constructor types in OCaml since everything generalizes the way it is done for the built-in constructor type of lists.

Much of the chapter is devoted to the interaction of abstract and concrete syntax. Of particular interest are syntactic infix operators with precedence and associativity. We finally discussed an abstract syntax for Mini-OCaml, which will serve as the basis of a type checker and an evaluator in later chapters. We will also see a parser translating strings into abstract syntax trees.

6 Parsing

The process of translating string representations of syntactic objects into tree representations is known as *parsing*. We will focus on a parsing method called *recursive descent* which works well for prefix linearizations and infix linearizations. For postfix linearizations we will see a different parsing method that is tail-recursive and maintains a stack of already parsed phrases. Once we have discussed the necessary techniques at the example of ABC-trees, we will consider a parser for abstract expressions.

6.1 Lexing

The first step in parsing a string translates the string into a list of **tokens** representing words. Most examples in this chapter take ABC-trees

```
type tree = A | B of tree * tree | C of tree * tree
```

as syntactic objects (often we will just use the constructors *A* and *B*). We accommodate the tokens needed for ABC-trees with a constructor type definition

```
type token = AT | BT | CT | LP | RP
```

providing tokens for the words 'A', 'B', 'C' and the parentheses '(' and ')'.

To translate a string into a list of tokens, we need to know a little bit more about strings in OCaml.

Strings provide an efficient and hardware-oriented representation of sequences of *characters*, where characters are represented as sequences of 8 booleans called *bytes*. Altogether there are $256 = 2^8$ characters. OCaml has types for both strings and characters and provides a function

$$String.get : string \rightarrow int \rightarrow char$$

such that *string.get s i* yields the character at position *i* of the string *s*. The positions of a string are numbered starting from 0 like the positions of a list. A main advantage of the hardware-oriented representation of strings is the fact that the *get* operation is fast, no matter how long the string and how large the position are. OCaml also provides a function

$$String.length : string \rightarrow int$$

6 Parsing

that yields the length of a string (the number of positions a string has).

OCaml has constants for characters, for instance 'A', 'B', 'C', '(', and ')'. There are also three constants ' ', '\n', and '\t' corresponding to the *white space characters* associated with the space key, enter key, and tabulator key on a keyboard.

For our examples dealing with ABC-trees, we will use a tail-recursive lexer

$$\text{lex} : \text{string} \rightarrow \mathcal{L}(\text{token})$$

declared as follows:

```
let lex s =
  let n = String.length s in
  let rec lex i l =
    if i >= n then List.rev l
    else match String.get s i with
      | 'A' -> lex (i+1) (AT::l)
      | 'B' -> lex (i+1) (BT::l)
      | 'C' -> lex (i+1) (CT::l)
      | '(' -> lex (i+1) (LP::l)
      | ')' -> lex (i+1) (RP::l)
      | ' ' | '\n' | '\t' -> lex (i+1) l
      | _ -> failwith "lex: illegal character"
  in lex 0 []
```

The tail-recursive helper function recurses on the current position in the string and accumulates the recognized tokens in a list that is reversed once the string is fully processed. The rule for white space characters exploits a convenience of OCaml making it possible to collapse rules with identical right-hand sides into a single rule.

Here is an example for *lex*:

```
lex "A BAC (A B A)" = [AT; BT; AT; CT; LP; AT; BT; AT; RP]
```

Exercise 6.1.1 Write a tail-recursive lexer for ABC-trees treating all characters but 'A', 'B', 'C', '(', and ')' as white space characters. For instance, "A (likes) B" should yield the list $[AT, LP, RP, BT]$.

Exercise 6.1.2 Write a tail-recursive lexer for ABC-trees treating comments ($* \dots *$) as white space as does an OCaml interpreter. Make sure that nested comment parentheses like $(* \dots (* \dots *) \dots *)$ are treated correctly. Hint: Have a counter argument recording how many opening comment parentheses have to be closed.

6 Parsing

Exercise 6.1.3 (Explode and implode)

The functions

```
let explode s = List.init (String.length s) (String.get s)
let implode l = List.fold_right (fun c s -> String.make 1 c ^ s) l ""
```

convert between strings and character lists such that the equations

$$\begin{aligned} \text{implode}(\text{explode } s) &= s \\ \text{explode}(\text{implode } l) &= l \end{aligned}$$

hold for all strings s and all character lists l . Note the use of OCaml's function *String.make* providing for the translation of characters into strings (*String.make* n c yields the string consisting of $n \geq 0$ occurrences of the character c).

- Try `explode "Saarbrücken"` and try to explain what you see.
- Try `implode ['\240'; '\159'; '\152'; '\138']` and enjoy.
- Try `implode ['\240'; '\159'; '\167'; '\144']` and enjoy.
- Convince yourself that OCaml's order on characters yields *true* for the following comparisons: `'A' < 'B'`, `'Z' < 'a'`, and `'a' < 'b'`. Look up "ASCII" in the web to know more.
- Convince yourself that OCaml's order on strings is the lexical order on character lists obtained with OCaml's order on characters.

Exercise 6.1.4 (ASCII Codes) OCaml has two functions

$$\begin{aligned} \text{Char.code} &: \text{char} \rightarrow \text{int} \\ \text{Char.chr} &: \text{int} \rightarrow \text{char} \end{aligned}$$

converting between characters and ASCII codes, which are numbers between 0 and 255. We have $\text{Char.chr}(\text{Char.code } c) = c$ for all characters c . Moreover $c < c'$ if and only if $\text{Char.code } c < \text{Char.code } c'$. Look up an ASCII table in the web to learn more. Using conversion to ASCII codes, declare functions as follows:

- A function checking whether a character is a digit.
- A function converting characters that are digits into numbers.
- A function checking whether a character is a lower case letter between 'a' and 'z'.
- A function checking whether a character is an upper case letter between 'A' and 'Z'.

6.2 Recursive Descent Parsing

Recursive descent parsing is a standard method for reconstructing syntactic objects from lexical representations. We demonstrate the method for the prefix linearization of AB-trees. We start with the grammar¹

$$tree ::= "A" \mid "B" tree tree$$

The grammar describes a *parsing function*

$$tree : \mathcal{L}(token) \rightarrow tree \times \mathcal{L}(token)$$

processing a token list as follows:

- If the first token of the list is 'A', return the tree A and the tail of the list.
- If the first token of the list is 'B', use recursion to obtain a tree t_1 and a tree t_2 from the list (t_1 first, then t_2). Then return the tree $B(t_1, t_2)$ and the remaining list.

We realize the described parsing function in OCaml as follows:

```
let rec tree l = match l with
| AT::l -> (A,l)
| BT::l ->
  let (t1,l) = tree l in
  let (t2,l) = tree l in
  (B(t1,t2), l)
| _ -> failwith "tree"
```

Note the systematic use of shadowing for the local variable l . The parsing function terminates since both recursive applications are done with smaller token lists. Here is an example:

```
tree (lex "BBAABABAA") = (B (B (A, A), B (A, B (A, A))), [])
```

The recursive calls of the parsing function on the given token list can be visualized with parentheses in the lexical representation:

$$\begin{array}{c}
 BBAABABAA \\
 \underline{\underline{BBAA}} \quad \underline{\underline{BA}} \quad \underline{\underline{BAA}} \\
 B(B(A)(A))(B(A)(B(A)(A)))
 \end{array}$$

It is interesting to extend the grammar and the parsing function such that a prefix linearization can contain redundant parentheses:

$$tree ::= "A" \mid "B" tree tree \mid "(" tree ")"$$

¹In grammars we write tokens as strings.

6 Parsing

The new grammar rule is realized with an additional rule

```
| LP::l -> let (t,l) = tree l in (t, verify RP l)
```

in the parsing function, where the helper function checking the presence of the right parenthesis is declared as follows:

```
let verify c l = match l with
| [] -> failwith "verify: no token"
| c'::l -> if c'=c then l else failwith "verify: wrong token"
```

Note that we use the letter c for variables ranging over tokens (the letter t is already taken for trees).

Exercise 6.2.1 Declare recursive descent parsers for ABC-trees in prefix linearization.

Exercise 6.2.2 Declare recursive descent parsers for AB-trees and linearizations as follows:

- OCaml linearization, with and without extra parentheses.
- Fully parenthesized infix linearization, with and without extra parentheses.

Do the same for ABC-trees.

Exercise 6.2.3 (Linearization with parentheses only)

AB-trees can be linearized with just parentheses. For instance,

$$B(B(A, B(A, A)), B(A, A)) \rightsquigarrow (((()())())())$$

Declare a linearizer and a parser for this format.

Exercise 6.2.4 Declare a function checking that a string consists of balanced parentheses only. For instance, $((()())())$ is fine but $((()()))$ is not.

6.3 Infix with Associativity

Recall the infix linearization of AB-trees where B is treated as a left or a right-associative syntactic operator. To get recursive descent parsers for these linearizations, we use the following grammar:

$$\begin{aligned} tree & ::= ptree tree' \\ tree' & ::= "B" ptree tree' \mid [] \\ ptree & ::= "A" \mid "(" tree ")" \end{aligned}$$

6 Parsing

Note that the rule for $tree'$ has a **nil alternative** which is matched by the empty token list. The recursion of the grammar terminates since every recursion step takes at least one token from the token list. Here are the parsing functions for the grammar treating B as a left-associative infix operator:

```

let rec tree l =
  let (t,l) = ptree l in tree' t l
and tree' t l = match l with
| BT::l -> let (t',l) = ptree l in tree' (B(t,t')) l
| _ -> (t,l)
and ptree l = match l with
| AT::l -> (A,l)
| LP::l -> let (t,l) = tree l in (t, verify RP l)
| _ -> failwith "tree"

```

The trick is that the helper function

$$tree' : tree \rightarrow \mathcal{L}(token) \rightarrow tree \times \mathcal{L}(token)$$

gets the tree that has been parsed before as an argument. Note that realizing the nil alternative of $tree'$ is straightforward.

If we want to treat B as a right-associative infix operator, the grammar remains unchanged and the rule for the token BT of the parsing function $tree'$ is changed as follows:

```

| BT::l -> let (t',l) = ptree l in
           let (t'',l) = tree' t' l in
           (B(t,t''), l)

```

Exercise 6.3.1 Declare a parser for ABC-trees where B and C are left-associative infix operators at the same precedence level.

Exercise 6.3.2 Declare a function that takes an AB-tree in infix linearization and returns an infix linearization of the tree as follows:

- Remove redundant parentheses assuming B is left-associative.
- Remove redundant parentheses assuming B is right-associative.
- Translate from B left-associative to B right-associative.
- Translate from B left-associative to OCaml linearization.

6 Parsing

Exercise 6.3.3 (Simplified right associativity) The parsing grammar given in this section makes it possible to parse B either as a left-associative or a right-associative operator. For the case where B is accommodated as a right-associative operator, the rule for the auxiliary parsing function $tree'$ can be simplified to

$$tree' ::= "B" tree \mid []$$

Declare parsing functions for the simplified grammar.

Exercise 6.3.4 After having seen the simplified grammar for right-associativity, Dieter Schlau gets excited and finally comes up with a simplified grammar he claims works for left-associativity:

$$tree ::= ptree \mid tree "B" ptree$$

Explain why Dieter's grammar cannot be realized with parsing functions.

6.4 Infix with Precedence and Juxtaposition

We now come to the case where we have two infix operators B and C and B takes its arguments before C (we say that B takes precedence over C). As in the linearization grammars the precedence hierarchy is taken care of in the parsing grammar by having rules for every precedence level:

$$\begin{aligned} tree &::= btree tree' \\ tree' &::= "C" btree tree' \mid [] \\ btree &::= ptree btree' \\ btree' &::= "B" ptree btree' \mid [] \\ ptree &::= "A" \mid "(" tree ")" \end{aligned}$$

We leave the realization of the parsing procedures as exercise. Recall that a single infix operator of highest precedence can be accommodated with juxtaposition. For the above grammar this means that the rule for $btree'$ is changed to

$$btree' ::= ptree btree' \mid []$$

As it comes to the parsing function for this rule we are now confronted with the problem that we need to decide whether the first alternative or the nil alternative is to be taken. We base the decision on the first token and take the first alternative if the token is a token a $ptree$ can start with ("A" or "(").

6 Parsing

Exercise 6.4.1 Declare parsers for ABC-trees and infix linearizations where B takes arguments before C , C is left-associative, and

- a) B is left-associative.
- b) B is right-associative.
- c) B is realized with left-associative juxtaposition.
- d) B is realized with right-associative juxtaposition.

Exercise 6.4.2 (Pair Notation) We may linearize trees $B(t_1, t_2)$ as pairs (t_1, t_2) . In this case, B needs neither precedence nor associativity since it can be accommodated at the lowest level *ptree*. Consider the following parsing grammar accommodating C as an infix operator and B with pair notation:

$$\begin{aligned} tree & ::= ptree\ tree' \\ tree' & ::= "C" ptree\ tree' \mid [] \\ ptree & ::= "A" \mid "(" tree\ ")" \mid "(" tree\ "," tree\ ")" \end{aligned}$$

Declare parsing functions for the grammar treating C as left-associative operator.

6.5 Postfix Parsing

Recall the postfix linearization of ABC-Trees:

$$tree ::= "A" \mid tree\ tree\ "B" \mid tree\ tree\ "C"$$

For instance, the tree $B(B(A, A), A)$ has the postfix linearization $AABAB$. As with the prefix linearization, there is no need for parentheses.

There is an elegant tail-recursive method for parsing postfix linearizations that is different from recursive descent parsing. The key idea is to work with a stack of already parsed subtrees so that when the operator B is encountered, the associated subtrees t_1 and t_2 can be taken from the stack and be replaced by the compound tree $B(t_1, t_2)$. We demonstrate the method with a trace for the string "AABAB":

"AABAB"	[]
"ABAB"	[A]
"BAB"	[A, A]
"AB"	[B(A, A)]
"B"	[A, B(A, A)]
""	[B(B(A, A), A)]

6 Parsing

We realize the design with a function

$$\text{deposit} : \mathcal{L}(\text{token}) \rightarrow \mathcal{L}(\text{tree}) \rightarrow \mathcal{L}(\text{tree})$$

such that $\text{deposit}(\text{lex}(\text{post}(t))) [] = [t]$, and more generally the equation

$$\text{deposit}(\text{lex}(\text{post}(t) @ l_1)) l_2 = \text{deposit } l_1 (t :: l_2)$$

holds for all trees t , character lists l_1 , and stacks l_2 (i.e., lists of trees). The defining equations for deposit are now straightforward:

$$\begin{aligned} \text{deposit } [] l_2 &:= l_2 \\ \text{deposit } (AT :: l_1) l_2 &:= \text{deposit } l_1 (A :: l_2) \\ \text{deposit } (BT :: l_1) (t_2 :: t_1 :: l_2) &:= \text{deposit } l_1 ((B(t_1, t_2) :: l_2) \\ \text{deposit } (CT :: l_1) (t_2 :: t_1 :: l_2) &:= \text{deposit } l_1 ((C(t_1, t_2) :: l_2) \\ \text{deposit } - - &:= [] \end{aligned}$$

The reversal of t_1 and t_2 in the equation for the token BT is needed since the linearization of t_1 precedes the linearization of t_2 , which means that t_1 is put on the stack before t_2 .

Exercise 6.5.1 Declare a function $\text{deposit}' : \text{string} \rightarrow \mathcal{O}(\text{tree})$ such that $\text{deposit}'(s) = \text{Some } t$ if and only if $\text{post}(t) = s$.

6.6 Mini-OCaml

The techniques we have seen for recursive decent parsing suffice to construct a parser for Mini-OCaml. We adapt the linearization grammar in §5.6 by adding auxiliary categories as follows:

$$\begin{aligned} \text{cexp} &::= \text{sexp cexp}' \\ \text{cexp}' &::= "<=" \text{sexp} \mid [] \\ \text{sexp} &::= \text{mexp sexp}' \\ \text{sexp}' &::= "+" \text{mexp sexp}' \mid "-" \text{mexp sexp}' \mid [] \\ \text{mexp} &::= \text{aexp mexp}' \\ \text{mexp}' &::= "*" \text{aexp mexp}' \mid [] \\ \text{aexp} &::= \text{pexp aexp}' \\ \text{aexp}' &::= \text{pexp aexp}' \mid [] \end{aligned}$$

The lexer for Mini-OCaml needs work. We start with the constructor type for tokens:

6 Parsing

```
type con  = Bcon of bool | Icon of int
type token = Add | Sub | Mul | LP | RP | Eq | Leq | Arr
           | Var of string | Con of con
           | If | Then | Else | Lam | Let | In | Rec
```

The lexer can be realized with a system of mutually tail-recursive functions. The basic design is as follows:

1. Finish if the string exhausted.
2. If first character is
 - a) white space, skip and recurse.
 - b) '+', '*', '=', '(', or ')', add corresponding token and recurse.
 - c) '<', verify next character is '=', add token *Leq*, and recurse.
 - d) '-' and next character is '>', add token *Arr*. Otherwise, add token *sub*. In both cases recurse.
 - e) digit, read number, add number token, and recurse.
 - f) lower-case letter, read identifier, and recurse after adding token as follows:
 - i. If identifier is "if", "then", "else", "fun", "let", "in", or "rec", add corresponding token.
 - ii. If identifier is "true" or "false", add corresponding constant token.
 - iii. Otherwise add variable token.

Exercise 6.6.1 Declare a function $string \rightarrow int$ reading a number from a string. Assume that the string starts with a digit and read as many digits as possible. For instance, "123 Dieter" should yield the number 123. Realize the function with a tail-recursive helper function and make use of ASCII codes (see Exercise 6.1.4). You may use the function

```
let num c = Char.code c - Char.code '0'
```

to convert characters that are digits into numbers.

Exercise 6.6.2 Declare a function $string \rightarrow string$ reading an identifier starting with a lower case letter from a string. Assume that the string starts with a lower case letter and read as many letters and digits as possible. For instance, "dieter2Schlau+x" should yield the identifier "dieter2Schlau". Realize the function with a tail-recursive helper function and OCaml's function *List.sub s i n* yielding the substring of *s* starting at position *i* and having length *n*. Make use of ASCII codes (see Exercise 6.1.4) to identify lower case letters, upper case letters, and digits.

6 Parsing

Exercise 6.6.3 (Project) Realize a lexer, a parser, and a linearizer for Mini-OCaml. Check that parsing after linearization takes you back to the tree you started with, and that parsing followed by linearization gives you an equivalent string. Extend Mini-OCaml with pair expressions (e_1, e_2) and the accompanying projections.

7 Mini-OCaml Interpreter

In this chapter we describe a programming project in which you will realize a complete Mini-OCaml interpreter. This is the first time we ask you to write and debug a program that altogether has a few hundred lines of code. As you will find out, this makes a dramatic difference to writing and debugging the few-line programs we have been considering so far.

Debugging larger programs can be very difficult. The method working best in practice is having a modular design of the program where each module can be debugged by itself. We speak of *divide and conquer*.

To find bugs you need to come up with tests, and once you have found a bug, you usually need to refine the test showing the bug in order that you find the place in the program that is responsible for the bug.

For the project, a Mini-OCaml interpreter, we provide you with a modular design featuring the following components:

- A lexer translating strings into lists of tokens.
- A parser translating lists of tokens into abstract expressions.
- A type checker computing the type of an expression in an environment.
- An evaluator computing the value of an expression in an environment.

Each of the components can be written and debugged by itself, which is the best one can hope for. The glue between the components is provided by constructor types for expressions, types, and tokens. There is also a constructor type for the values of Mini-OCaml covering plain and recursive closures.

We assume you are familiar with Chapter 2 on syntax and semantics and Chapter 6 on parsing.

7.1 Expressions, Types, Environments

Our starting point is the abstract grammar for the types and expressions of Mini-OCaml:

$$\begin{aligned}
 t &::= \text{bool} \mid \text{int} \mid t_1 \rightarrow t_2 \\
 o &::= + \mid - \mid \cdot \mid \leq \\
 e &::= x \mid c \mid e_1 o e_2 \mid e_1 e_2 && (c : \mathbb{B} \mid \mathbb{Z}) \\
 &\mid \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \\
 &\mid \lambda x. e \mid \lambda x : t. e \\
 &\mid \text{LET } x = e_1 \text{ IN } e_2 \\
 &\mid \text{LET REC } f x = e_1 \text{ IN } e_2 \\
 &\mid \text{LET REC } f (x : t_1) : t_2 = e_1 \text{ IN } e_2
 \end{aligned}$$

The letter c ranges over *constants* which we choose to be booleans or integers, and the letters x and f range over *variables* which we choose to be strings.

An important design decision is to have lambda expressions and recursive let expressions both in untyped and typed form.

We realize the abstract grammar in OCaml with a system of constructor types:

```

type ty = Bool | Int | Arrow of ty * ty
type con = Bcon of bool | Icon of int
type op = Add | Sub | Mul | Leq
type var = string
type exp = Var of var | Con of con
          | Oapp of op * exp * exp
          | Fapp of exp * exp
          | If of exp * exp * exp
          | Lam of var * exp
          | Lamty of var * ty * exp
          | Let of var * exp * exp
          | Letrec of var * var * exp * exp
          | Letrecty of var * var * ty * ty * exp * exp

```

Note that *var* is not a constructor type but simply a second name for *string* that we introduce for readability.

Exercise 7.1.1 Write a tail-recursive faculty function $n!$ in Mini-OCaml. First write the expression in OCaml, then translate it into the abstract syntax of Mini-OCaml in OCaml.

Environments

For the evaluator and the type checker we need environments. We implement environments as key-value maps:

```

type ('a,'b) env = ('a * 'b) list
let empty : ('a,'b) env = []
let update (env : ('a,'b) env) a b : ('a,'b) env = (a,b) :: env
let rec lookup (env : ('a,'b) env) a = match env with
  | (a',b) :: env -> if a = a' then Some b else lookup env a
  | [] -> None

```

Note that *env* is a second name for *list*. There is the possibility to realize environments with a custom constructor type different from lists.

7.2 Type Checker

We realize the type checker for Mini-OCaml with a function

$$check : env \text{ var } ty \rightarrow exp \rightarrow ty$$

that checks whether an expression is well-typed in an environment. If this is the case, *check* yields the unique type of the expression, otherwise it throws an exception. Expressions containing untyped lambda or recursive let expressions count as ill-typed. The type checker matches on the given expression and for each variant follows the algorithmic reading of the respective typing rule (Figure 2.2):

```

let rec check env e : ty = match e with
  | Var x -> ...
  | Con (Bcon b) -> Bool
  | Con (Icon n) -> Int
  | Oapp (o,e1,e2) -> check_op o (check env e1) (check env e2)
  | Fapp (e1,e2) -> check_fun (check env e1) (check env e2)
  | If (e1,e2,e3) -> ...
  | Lam (_,_) -> failwith "fun: missing type"
  | Lamty (x,t,e) -> Arrow (t, check (update env x t) e)
  | Let (x,e1,e2) -> check (update env x (check env e1)) e2
  | Letrec (f,x,e1,e2) -> failwith "let rec: missing types"
  | Letrecty (f,x,t1,t2,e1,e2) -> ...

```

Note that we are using helper functions for operator and function applications for better readability.

7.3 Evaluator

For the evaluator we first define a constructor type providing the values of Mini-OCaml as OCaml values:

7 Mini-OCaml Interpreter

```
type value = Bval of bool | Ival of int
           | Closure of var * exp * (var, value) env
           | Rclosure of var * var * exp * (var, value) env
```

We realize the evaluator with a function

$$eval : env \text{ var } value \rightarrow exp \rightarrow value$$

that evaluates an expression in an environment. The evaluator matches on the given expression and for each variant follows the algorithmic reading of the respective evaluation rule (Figure 2.4):

```
let rec eval env e : value = match e with
| Var x -> ...
| Con (Bcon b) -> Bval b
| Con (Icon n) -> Ival n
| Oapp (o,e1,e2) -> eval_op o (eval env e1) (eval env e2)
| Fapp (e1,e2) -> eval_fun (eval env e1) (eval env e2)
| If (e1,e2,e3) -> ...
| Lam (x,e) | Lamty (x,_,e) -> Closure (x,e,env)
| Let (x,e1,e2) -> eval (update env x (eval env e1)) e2
| Letrec (f,x,e1,e2) | Letrecty (f,x,_,_,e1,e2) -> ...
and eval_fun v1 v2 = match v1 with ...
```

As with the type checker, we use helper functions for operator and function applications for better readability. This time the helper function for function applications is mutually recursive with the master evaluation function.

7.4 Lexer

The lexer for Mini-OCaml needs work, mainly because it needs to recognize identifiers (i.e., variables) and number constants. The first step consists in declaring a constructor type for tokens:

```
type const = BCON of bool | ICON of int
type token = LP | RP | EQ | COL | ARR | ADD | SUB | MUL | LEQ
           | IF | THEN | ELSE | LAM | LET | IN | REC
           | CON of const | VAR of string | BOOL | INT
```

The tokens *LP*, *RP*, *EQ*, *COL*, *ARR*, and *LAM* are realized with the strings "(", ")", "=", ":", "->", and "fun".

7 Mini-OCaml Interpreter

We realize the lexer with 5 mutually tail-recursive functions:

```
let lex s : token list =
  let get i = String.get s i in
  let getstr i n = String.sub s (i-n) n in
  let exhausted i = i >= String.length s in
  let verify i c = not (exhausted i) && get i = c in
  let rec lex i l =
    if exhausted i then List.rev l
    else match get i with
      | '+' -> lex (i+1) (ADD::l)
      ...
      | c when whitespace c -> lex (i+1) l
      | c when digit c -> lex_num (i+1) (num c) l
      | c when lc_letter c -> lex_id (i+1) l l
      | c -> failwith "lex: illegal character"
  and lex_num i n l = ...
  and lex_num' i n l = lex i (CON (ICON n)::l)
  and lex_id i n l = ...
  and lex_id' i n l = match getstr i n with
    | "if" -> lex i (IF::l)
    ...
    | s -> lex i (VAR s::l)
  in lex 0 []
```

There is some cleverness in this design. We avoid considering auxiliary strings by accessing the given string by position using *String.get* and by using *String.sub* to extract an identifier identified by its starting position and length. The constants *true* and *false* and keywords like *if* are first recognized as identifiers and then transformed into the respective constant or keyword tokens.

Digits and letters are recognized by using the function *Char.code* and exploiting the fact that the codes of the respective characters follow each other continuously in a canonical order (there are separate intervals for digits, lower case letters, and upper case letters).

Following OCaml, an identifier must start with a lower case letter and can then continue with digits, lower and upper case letters, and the special characters *'_'* (underline) and *'''* (quote).

Note the use of patterns with **when conditions** in the declaration of the parsing function *lex*. This is a syntactical convenience translating into nested conditionals.

Note that *lex_id'* is the function where you can change the identifier-like keywords of the concrete syntax. For instance, you may replace the keyword *fun* with the keyword *lam* or *lambda*. Such a change will not show at the level of tokens and hence will not be visible in the parser.

7 Mini-OCaml Interpreter

We remark that the lexer follows the *longest munch rule* when it reads identifiers and numbers; that is, it reads as many characters as are possible for an identifier or number. Thus " ifx " is recognized as a single variable token *Var* "ifx" rather than the keyword token *If* followed by the variable token *Var* "x".

Exercise 7.4.1 Rewrite the function *lex* so that no patterns with when conditions are used. Use nested conditionals instead.

Exercise 7.4.2 Extend the lexer with comments.

7.5 Parser

The parsing grammar for Mini-OCaml consists of a grammar for types and a grammar for expressions building on the grammar for types. We start with the grammar for expressions.

The parsing grammar for Mini-OCaml expressions has 6 levels:

<i>exp</i>	if, fun, let	top level
<i>cexp</i>	≤	comparisons
<i>sexp</i>	+, −	additive operators
<i>mexp</i>	·	multiplicative operators
<i>aexp</i>		function applications
<i>pexp</i>		bottom level

The top level takes care of the prefix phrases, which recurse to the top level. The top level delegates to the comparison level in case no keyword for a prefix phrase is present. We then have 4 infix levels taking care of operator applications and function applications. Function applications are at the lowest infix level and are realized with juxtaposition. Additive and multiplicative operators and function applications are realized with left-associativity, and comparisons are realized without associativity, deviating from OCaml. The bottom level takes care of variables, constants, and parenthesized expressions. For the 3 levels realizing left-associativity the parsing grammar has auxiliary categories *sexp'*, *mexp'*, and *aexp'*.

The parsing function for the top level is interesting in that it can make good use of pattern matching:

7 Mini-OCaml Interpreter

```
let rec exp l : exp * token list = match l with
| IF::l -> ...
| LAM::VAR x::ARR::l -> ...
| LAM::LP::VAR x::COL::l -> ...
| LET::VAR x::EQ::l -> ...
| LET::REC::VAR f::VAR x::EQ::l -> ...
| LET::REC::VAR f::LP::VAR x::COL::l -> ...
| l -> cexp l
```

Note that we follow OCaml's concrete syntax for typed lambda expressions in that we require parentheses around typed argument variables; for instance, the abstract lambda expression $\lambda x : int. x$ must be written as "fun(x:int)->x".

We now come to the parsing grammar for Mini-OCaml types:

$$\begin{aligned} ty &::= pty \ ty' \\ ty' &::= \text{"->"} \ ty \mid \square \\ pty &::= \text{"bool"} \mid \text{"int"} \mid \text{"("} \ ty \ \text{")"} \end{aligned}$$

The parsing function ty' accommodates "->" as a right-associative infix operator.

Exercise 7.5.1 Give the parsing grammar for Mini-OCaml expressions.

Exercise 7.5.2 Write the parsing functions for Mini-OCaml types and Mini-OCaml expressions.

7.6 The Project

Your project consists in writing one file realizing two functions

$$\begin{aligned} checkStr &: string \rightarrow type \\ evalStr &: string \rightarrow value \end{aligned}$$

parsing strings as Mini-OCaml expressions and either type checking or evaluating the expressions. Here are our side conditions:

1. Use the CMS to hand in your file in time.
2. Only files that type check are accepted.
3. You can earn a maximum of 15 points with the project. The project counts as the test for this week.
4. If your functions contain errors you know of, please say so in the file.
5. Please use the types and the function and variable names specified in this chapter.

7 Mini-OCaml Interpreter

6. It's fine to explore the files *parser.ml*, *typechecker.ml*, and *evaluator.ml* we provide, but do not cut and paste.
7. It's fine to discuss and exchange ideas with your fellow students.
8. You may ask on the forum to clarify the specification of the project, but you must not ask for solutions or post solutions.
9. You must understand in detail the program you hand in.
10. You must not copy code.

We hope you will enjoy the project. There are many interesting issues and add-ons you may explore. Here are a few proposals:

1. Find an example for every exception your interpreter may raise. Classify into exceptions concerning lexing, parsing, type checking, and evaluation.
2. Extend Mini-OCaml with pairs. Besides pair expressions (e_1, e_2) , provide let expressions `LET $(x_1, x_2) = e_1$ IN e_2` with pair patterns. Alternatively, provide native projections *fst* e and *snd* e , or lambda expressions $\lambda(x_1, x_2).e$ with pair patterns. Show that the 3 alternatives for pair decomposition can express each other.
3. Extend Mini-OCaml with expressions *fail ty* that raise an exception. Explain why fail expressions must be given with a type.
4. Extend the lexer with comments.
5. Challenge. Extend Mini-OCaml with lists. Expressions should provide `nil` and `cons` and simple matches for lists. The difficult part is the typing. To fit the design of the type checker, `nil` must be annotated with the base type of the list type (the situation is similar to fail expressions).

8 Running Time

Often there are several algorithms solving a given computational task, and these algorithm may differ significantly in their running times. A typical example is sorting of lists. Insertion sort, the sorting algorithm we have already seen, requires quadratic worst-case time. We will see a substantially faster algorithm known as merge sort in this chapter whose running time is almost linear.

It turns out that we can define the running time of an expression as the number of function calls needed for its execution. This abstract and technology-independent notion of running time provides an excellent tool for distinguishing between faster and slower algorithms.

8.1 The Idea

Given a specification of a computational task, we can write different functions satisfying the specification. In case the functions are based on different algorithms, the execution times of the functions may differ substantially. There is an abstract notion of asymptotic running time predicting the execution times of functions. Prominent examples of asymptotic running times are

$O(1)$	constant time
$O(\log n)$	logarithmic time
$O(n)$	linear time
$O(n \log n)$	linearithmic time
$O(n^2)$	quadratic time
$O(2^n)$	exponential time

A function that has linear running time has the property that the execution time grows at most linearly with the argument size. Similarly, functions that have logarithmic or quadratic or exponential running time have the property that the execution time grows at most logarithmically or quadratically or exponentially with the argument size. Linearithmic time is almost like linear time and behaves the same in practice. Finally, the execution time of a function that has constant running time is bounded for all arguments by a fixed constant. In practice, constant, linear, and linearithmic running time is what one looks for, quadratic running times signals that there will be performance problems for larger

8 Running Time

arguments, and exponential running time signals that there are arguments of reasonable size for which the execution time is too high for practical purposes.

The running times for list sorting algorithms are interesting. We will see that insertion sort has quadratic running time, and that there is a sorting algorithm (merge sort) with linearithmic running time that is much faster in practice.

Logarithmic running time $O(\log n)$ is much faster than linear time. The standard example for logarithmic running time is binary search, an algorithm for searching ordered sequences we will discuss in this chapter. Logarithmic running time occurs if the input size is halved at each recursion step, and only a constant number of function calls is needed to prepare a recursion step.

A good measure of the abstract running time of an expression is the number of function calls needed for its execution. Expressions whose execution doesn't involve function calls receive the abstract running time 0. As an example, we consider a list concatenation function:

$$\begin{aligned} [] @ l_2 &:= l_2 \\ (x :: l_1) @ l_2 &:= x :: (l_1 @ l_2) \end{aligned}$$

The defining equations tell us that the running time of a concatenation $l_1 @ l_2$ only depends on the length of the list l_1 . In fact, from the defining equations of the concatenation function $@$ we can obtain a recursive running time function

$$\begin{aligned} r(0) &:= 1 \\ r(n+1) &:= 1 + r(n) \end{aligned}$$

giving us the running time for every call $l_1 @ l_2$ where l_1 has length n . A little bit of discrete mathematics gives us an explicit characterization of the running time function:

$$r(n) = n + 1$$

We can now say that the list concatenation function $@$ has linear running time in the length of the first list.

We remark that we distinguish between function applications and function calls. A *function application* is a syntactic expression $e e_1 \dots e_n$. A *function call* is a tuple (v, v_1, \dots, v_n) of values that is obtained by evaluating the expressions of a function application.

8.2 List Reversal

We will now look at two functions for list reversal giving us two interesting examples for abstract running times. We start with a tail-recursive function

$$\begin{aligned} \text{rev_append } [] l_2 &:= l_2 \\ \text{rev_append } (x :: l_1) l_2 &:= \text{rev_append } l_1 (x :: l_2) \end{aligned}$$

that has linear running time in the length of the first argument (following the argumentation we have seen for list concatenation). We can now reverse a list l with a function call $(\text{rev_append } l [])$ taking running time linear in the length of l . We remark at this point that rev_append gives us an optimal list reversal function that cannot be further improved.

Another list reversal function we call **naive reversal** is

$$\begin{aligned} \text{rev } [] &:= [] \\ \text{rev } (x :: l) &:= \text{rev } (l) @ [x] \end{aligned}$$

Practical experiments with an OCaml interpreter show that the performance of naive list reversal is much worse than the performance of tail-recursive list reversal. While rev_append will yield the reversal of a list of length 20000 instantaneously, the naive reversal function rev will take considerable time for the same task (go to length 100000 to see a more impressive time difference).

The above experiments require an interpreter that can handle deeply nested function calls. While this is the case for the native OCaml interpreter, the browser-based TryOCaml interpreter severely limits the height of the call stack (about 1000 function calls at the time of writing). Thus the above experiments require a native OCaml interpreter. We remark that function calls in tail position (as is the case with tail recursion) don't require allocation on the call stack. In fact, TryOCaml can easily execute the tail-recursive function rev_append for lists of length 100000. On the other hand, TryOCaml aborts the execution of the naive list reversal function rev already for lists of length 3000.

We return to the discussion why naive list reversal is so much slower than tail-recursive list reversal. The answer simply is that naive list reversal has quadratic running time while tail-recursive list reversal has linear runtime. For instance, when we reverse a list of length 10^4 , naive reversal roughly requires 100 million function calls while tail-recursive reversal only needs ten thousand function calls (think of money to get a feel for the numbers).

8 Running Time

Tail-recursive list reversal has running time $n + 1$ for lists of length n , which can be shown with the same method we used for list concatenation. For naive reversal we have to keep in mind that list concatenation is not an operation (as suggested by the symbol $@$) but a function. This means we have to count the function calls needed for concatenation for every recursive call of *rev*. This gives us the recursive definition of the running time function for *rev*:

$$\begin{aligned}r(0) &:= 1 \\ r(n + 1) &:= 1 + r(n) + (n + 1)\end{aligned}$$

The term $(n + 1)$ is the number of recursion steps a concatenation $rev(l) @ [x]$ takes if l has length n . We make use of the fact that *rev* leaves the length of a list unchanged. We now have a recursive definition of the exact runtime function for *rev*. With standard techniques from discrete mathematics one can obtain an equation characterizing $r(n)$ without recursion:

$$r(n) = (n + 1)(n + 2)/2$$

One speaks of *recurrence solving*. We will not explain this topic here. For practical purposes it is often convenient to use one of the recurrence solvers in the Internet.¹

Exercise 8.2.1 Define a linear time function concatenating two lists using only tail recursion. Hint: Exploit the equations $l_1 @_r l_2 = rev\ l_1 @ l_2$ and $rev(rev\ l) = l$.

8.3 Insertion Sort

Recall insertion sort from §4.12:

$$\begin{aligned}insert &: \mathbb{Z} \rightarrow \mathcal{L}(\mathbb{Z}) \rightarrow \mathcal{L}(\mathbb{Z}) \\ insert\ x\ [] &:= [x] \\ insert\ x\ (y :: l) &:= x :: y :: l \quad \text{if } x \leq y \\ insert\ x\ (y :: l) &:= y :: insert\ x\ l \quad \text{if } x > y \\ isort &: \mathcal{L}(\mathbb{Z}) \rightarrow \mathcal{L}(\mathbb{Z}) \\ isort\ [] &:= [] \\ isort\ (x :: l) &:= insert\ x\ (isort\ l)\end{aligned}$$

¹We recommend <https://www.wolframalpha.com>. Entering the two defining equations into the solver will suffice to obtain the closed formula characterization of $r(n)$.

8 Running Time

The structure of insertion sort is similar to naive list reversal. However, there is the new aspect that an insertion may cost between 1 and $n + 1$ function calls depending on where the insertion in a list of length n takes place. We say that list insertion has constant running time in the **best case** and linear running time in the **worst case**. We also say that list insertion has linear **worst-case running time**.

We now look at the running time of *isort*. We have linear best-case running time $(2n + 1)$ if the input list is in ascending order ($x_1 \leq x_2 \leq \dots$) and quadratic worst-case running time $((n+1)(n+2)/2)$ if the input list is in strictly descending order ($x_1 > x_2 > \dots$). The running time function for the worst case is identical with the running time function for naive list reversal.

The quadratic worst-case running time shows in practice. While a list of length 20000 is sorted instantaneously if it is in ascending order, sorting takes considerable time if the list is in strictly descending order.

8.4 Merge Sort

Because of its quadratic worst-case running time, insertion sort is not a good sorting function in practice. It turns out that there is a much better sorting algorithm called *merge sort* that in practice has almost linear running time in the length of the input list.

Merge sort first splits the input list into two lists of equal length (plus/minus one). It then sorts the two sublists by recursion and merges the two sorted lists into one sorted list. Both splitting and merging can be carried out in linear time in the length of the input list. The question now is how often merge sort has to recurse. It turns out that the recursion depth (see §8.8) of merge sort is logarithmic in the length of the input length. For instance, if we are given an input list of length 2^{20} (about one million), the recursion depth will be 20, since the length of the lists to be sorted decreases exponentially: 2^{20} , 2^{19} , 2^{18} , 2^{17} , 2^{16} , 2^{15} , \dots .

We formulate merge sort in OCaml as follows:

```
let rec split l l1 l2 = match l with
| [] -> (l1,l2)
| [x] -> (x::l1,l2)
| x::y::l -> split l (x::l1) (y::l2)

let rec merge l1 l2 = match l1, l2 with
| [], l2 -> l2
| l1, [] -> l1
| x::l1, y::l2 when x <= y -> x :: merge l1 (y::l2)
| x::l1, y::l2 -> y :: merge (x::l1) l2
```

8 Running Time

```
let rec msort l = match l with
| x::y::l -> let (l1,l2) = split l [x] [y] in
              merge (msort l1) (msort l2)
| l -> l
```

From the code it is clear that there is no dependence on the order of the input list. It is also clear that the running time of *split* for an input list of length n is at most $n + 1$ (for larger n it is at most $1 + n/2$). The running time of *merge* is $1 + n_1 + n_2$ where n_1 and n_2 are the lengths of the input lists l_1 and l_2 . Thus the running time of *msort* is linear in the length of the input list plus twice the running time of *msort* for input lists of half the length. This results in linearithmic running time $O(n \log n)$ for *msort* and input lists of length n .

Note that in the above code *split* is tail-recursive but *merge* is not. A tail-recursive variant of *merge* is straightforward:

```
let rec merge l1 l2 l = match l1, l2 with
| [], l2 -> List.rev_append l l2
| l1, [] -> List.rev_append l l1
| x::l1, y::l2 when x <= y -> merge l1 (y::l2) (x::l)
| x::l1, y::l2 -> merge (x::l1) l2 (y::l)
```

With the tail-recursive merge function only the recursion depth of *msort* remains, which is logarithmic in the length of the input list.

Exercise 8.4.1 Check with an interpreter that the execution time of *msort* is moderate even for lists of length 10^5 . Convince yourself that the ordering of the input list doesn't matter.

8.5 Binary Search

Suppose we have a function $f : \mathbb{N} \rightarrow \mathbb{N}$ and want to check whether the sequence

$$f(0), f(1), f(2), \dots$$

takes the value x for some k (that is, $f(k) = x$). Without further information about f , we can perform a linear search for the first k such that $f(k) = x$.

We can search faster if we assume that f is *increasing*:

$$f(0) \leq f(1) \leq f(2) \leq \dots$$

8 Running Time

We can then apply a search technique called **binary search**. Binary search modifies the problem such that, given l , r , and x , we look for a position k in the sequence

$$f(l) \leq \dots \leq f(r)$$

such that $l \leq k \leq r$ and $f(k) = x$. Binary search proceeds as follows:

1. If $r < l$, the number k we are looking for does not exist.
2. If $l \leq r$, we determine $m = (l + r)/2$ and consider three cases:
 - a) If $f(m) = x$, m is the number we are looking for.
 - b) If $f(m) < x$, we continue the search in the interval $(m + 1, r)$.
 - c) If $f(m) > x$, we continue the search in the interval $(l, m - 1)$.

We note that on recursion the size of the interval to be searched is halved. Thus, if we start with an interval of length 2^n , we have at most $n + 1$ recursion steps until we terminate. In other words, the worst-case running time of binary search measured in recursion steps is logarithmic in the length of the interval to be searched. This is in contrast to linear search, where the worst-case running time measured in recursion steps is linear in the length of the interval to be searched.

We realize binary search with an OCAML function

$$\text{find} : \forall \alpha. (\text{int} \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{int} \rightarrow \text{int} \rightarrow \mathcal{O}(\text{int})$$

declared as follows:

```
let find f x l r =
  let rec aux l r =
    if r < l then None
    else let m = (l+r)/2 in
         let y = f m in
         if x = y then Some m
         else if x < y then aux l (m-1) else aux (m+1) r
  in aux l r
```

Note that the helper function *aux* is tail recursive.

In contrast to linear search, binary needs an upper bound r . If we work with machine numbers, we may simply use the largest machine integer *max_int* as an upper bound. If we consider the problem mathematically, we first need to find an upper bound r such that $x \leq f(r)$. If such an r exists, we may find it quickly with an **exponential search**. We start with a large number k and double k until $x \leq f(k)$. If f is strictly increasing ($f(0) < f(1) < f(2) < \dots$), we have $x \leq f(x)$ and can hence use x as an upper bound.

8 Running Time

Exercise 8.5.1 The binary search function can be simplified if it returns booleans rather than options. Declare a binary search function checking whether f yields a given value for some number in a given interval. Use the lazy boolean connectives.

Exercise 8.5.2 Binary search can be used for inversion of strictly increasing functions.

- Declare a function *square* deciding whether an integer $x \geq 0$ is a square number (that is, $x = n^2$ for some n). The worst-case running time of *square* x should be logarithmic in x .
- Declare a function *sqr* that given an integer $x \geq 0$ computes the largest n such that $n^2 \leq x$. The worst-case running time of *sqr* x should be logarithmic in x .
- Declare a function *inv* that given a strictly increasing function $f : \mathbb{N} \rightarrow \mathbb{N}$ and an integer $x \geq 0$ computes the largest n such that $f(n) \leq x$. The worst-case running time of *inv* f x should be logarithmic in x if f has constant running time.

8.6 Trees

Recall the definition of AB-trees:

$$tree ::= A \mid B(tree, tree)$$

Two essential notions for trees in general and AB-trees in particular are *size* and *depth*:

$$size\ A := 1$$

$$size\ (B\ t_1\ t_2) := 1 + size\ t_1 + size\ t_2$$

$$depth\ A := 0$$

$$depth\ (B\ t_1\ t_2) := 1 + \max(depth\ t_1)\ (depth\ t_2)$$

Note that the function *size* computes its own running time; that is, the running time of *size* for a tree t is *size* (t) .

Here is a function *mtree* : $\mathbb{N} \rightarrow tree$ that yields an AB-tree of depth n that has maximal size (see also Exercise 5.2.5):

$$mtree(0) := A$$

$$mtree(n + 1) := B(mtree(n), mtree(n))$$

The running time of *mtree* is

$$r(0) := 1$$

$$r(n + 1) := 1 + r(n) + r(n)$$

8 Running Time

A recurrence solver yields the explicit characterization

$$r(n) = 2^{n+1} - 1$$

which tells us that *mtree* has exponential running time $O(2^n)$.

It is straightforward to rewrite *mtree* such that the running time becomes linear:

$$\begin{aligned} \text{mtree}(0) &:= A \\ \text{mtree}(n+1) &:= \text{LET } t = \text{mtree}(n) \text{ IN } B(t, t) \end{aligned}$$

The insight is that a maximal tree for a given depth has two identical subtrees. Thus it suffices to compute the subtree once and use it twice.

For maximal trees the running time of *size* is exponential in the depth of the input tree. Consequently, the running time of $\lambda n. \text{size}(\text{mtree } n)$ is exponential in n even if the linear-time variant of *mtree* is used.

Exercise 8.6.1 (Minimal Trees) We call an AB-tree of depth n *minimal* if its size is minimal for all AB-tree of depth n .

- Argue that there are two different minimal AB-trees of depth 2.
- Declare a linear-time function that given n yields a minimal AB-tree of depth n .
- Declare a linear-time function that given n yields the size of minimal AB-trees of depth n .
- Give an explicit formula for the size of minimal AB-trees of depth n .

Exercise 8.6.2 (Ternary Trees) Consider *ternary trees* as follows:

$$\text{tree} ::= A \mid B(\text{tree}, \text{tree}, \text{tree})$$

- Declare a function that yields the size of ternary trees.
- Declare a function that yields the depth of ternary trees.
- Declare a linear-time function that yields a ternary tree of depth n that has maximal size.
- Declare a linear-time function that yields a ternary tree of depth n that has minimal size.
- Give an explicit formula for the size of maximal ternary trees of depth n .
- Give an explicit formula for the size of minimal ternary trees of depth n .

8.7 Summary and Background

A key idea in this chapter is taking the number of function calls needed for the execution of an expression as running time of the expression. This definition of running time works amazingly well in practice, in particular as it comes to identifying functions whose execution time may be problematic in practice. As one would expect, in a functional programming language significant running times can only be obtained with recursion.

Another basic idea is looking at the maximal running time for arguments of a given size. For lists, typically their length is taken as size. The idea of taking the maximal running time for all arguments of a given size is known as *worst-case assumption*. A good example for the worst-case assumption is insertion sort, where the best-case running time is linear and the worst-case running time is quadratic in the length of the input list.

Once a numeric argument size is fixed, one can usually define the worst-case running time function for a given function as a recursive function $\mathbb{N} \rightarrow \mathbb{N}$ following the defining equations of the given function.

Given a recursive running time function, one can usually obtain an explicit formula for the running time using so-called recurrence solving techniques from discrete mathematics. Once one has an explicit formula for the running time, one can obtain the asymptotic running time in big O notation (e.g., $O(n)$, $O(n \log n)$, $O(n^2)$). Big O notation is also known as *Bachmann–Landau notation* or *asymptotic notation*.

Big O notation is based on an *asymptotic dominance relation* $f \preceq g$ for functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ defined as follows:

$$f \preceq g := \exists n_0, k, c. \forall n \geq n_0. f(n) \leq k \cdot g(n) + c$$

Speaking informally, we have $f \preceq g$ iff f is upwards bounded by a pumped version of g for all n but finitely many exceptions.

Asymptotic dominance gives us the *asymptotic equivalence relation*

$$f \approx g := f \preceq g \wedge g \preceq f$$

If $f \approx g$, one says that f and g have the same **growth rate**.

Here are a few examples to help your intuition:

$$\begin{aligned} 2n + 13 &\approx n \\ 7n^2 + 2n + 13 &\approx n^2 \\ 2^{n+3} + 7n^2 + 2n + 13 &\approx 2^n \end{aligned}$$

8 Running Time

We also have

$$\underbrace{n^1 \preceq n^2 \preceq n^3 \preceq \dots}_{\text{polynomial}} \preceq \underbrace{2^n \preceq 3^n \preceq 4^n \preceq \dots}_{\text{exponential}}$$

where the converse directions do not hold. One speaks of polynomial and exponential functions.

We now say that a function f is $O(g)$ if f and g are asymptotically equivalent. Big O notations like $O(n)$ and $O(n^2)$ are in fact abbreviations for $O(\lambda n.n)$ and $O(\lambda n.n^2)$. Moreover, $O(n \log n)$ abbreviates $O(\lambda n. \lceil \log_2(n+1) \rceil)$.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. The letter O is used because the growth rate of a function is also referred to as the order of the function.

In the literature, “ f is $O(g)$ ” stands for $f \preceq g$ (upper bound) and “ f is $\Theta(g)$ ” stands for $f \approx g$ (tight upper bound). We have taken the freedom to write $O(g)$ in place of $\Theta(g)$ because in practice when one says that insertion sort has running time $O(n^2)$ one means that the running time function of insertion sort is $\Theta(g)$.

Exercise 8.7.1 Give the order of the following functions using big O notation.

- a) $\lambda n. 2n + 13$
- b) $\lambda n. 5n^2 + 2n + 13$
- c) $\lambda n. 5n^3 + n(n+1) + 13n + 2$
- d) $\lambda n. 2^n + n^4$
- e) $\lambda n. 2^{n+5} + n^2$

Exercise 8.7.2 (Characterizations of asymptotic dominance)

Show that the following propositions are equivalent for all functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$:

1. $\exists n_0, k, c. \forall n \geq n_0. f(n) \leq k \cdot g(n) + c$
2. $\exists k. \forall n. f(n) \leq k \cdot (g(n) + 1)$

8.8 Call Depth

The execution of a function call usually requires the execution of further function calls. This is particularly true for calls of recursive functions. The function calls that must be executed for a given initial function call are naturally organized into a *call tree* where the calls needed for a particular call appear as descendants of the particular call. Note that the size of a call tree is the abstract running time of the initial function call appearing as the root of the tree.

8 Running Time

When a call is executed, all calls in its call tree must be executed. When an interpreter executes a call that is a descendent of the the initial call, it must store the path from the root to the current call so that it can return from the current call to the parent call. The path from the initial call to the current call is stored in the so-called *call stack*.

The length of the path from the initial call to a lower call is known as *call depth* (often called *recursion depth*). In contrast to functional programming languages, most programming languages severely limit the available call depth, typically to a few thousand calls. At this point tail-recursion and tail-recursive calls turn out to be important since tail-recursive calls don't count for the call depth in some programming systems limiting the call depth. The reason is that a tail-recursive call directly yields the result of the parent call so that it can replace the parent call on the call stack when its execution is initiated.

In contrast to native OCaml interpreters, the browser-based TryOCaml interpreter (realized with JavaScript) limits the recursion depth to a few thousand calls. If you want to attack computationally demanding problems with TryOCaml, it is necessary that you limit the necessary recursion depth by replacing full recursion with tail recursion. A good example for this problem is merge sort, where the helper functions *split* and *merge* can be realized with tail recursion and only the main function requires full recursion. Since the main function halves the input size upon recursion, it requires call depth that is only logarithmic in the size of the initial input (for instance, call depth 20 for a list of length 10^6).

9 Inductive Correctness Proofs

In this chapter we will prove properties of functions defined with equations and terminating recursion. To handle recursive functions, terminating recursive proofs commonly known as inductive proofs will be used.

We assume familiarity with equational reasoning, a basic mathematical proof technique obtaining valid equations by rewriting with valid equations. For instance, $5 + (x - x) = 5$ follows with the equations $x - x = 0$ and $x + 0 = x$. One speaks of *replacement of equals by equals*.

9.1 Propositions and Proofs

Propositions are mathematical statements that may be true or false. Equations are a basic form of propositions. Proofs are mathematical constructions that demonstrate the truth of propositions. Propositions can be combined with three basic connectives:

conjunction	$P \wedge Q$	P and Q
disjunction	$P \vee Q$	P or Q
implication	$P \rightarrow Q$	if P then Q

There is a special proposition \perp called falsity that has no proof. Negation $\neg P$ is expressed as $P \rightarrow \perp$. Variables occurring in propositions can be qualified with typed quantifications:

universal quantification	$\forall x : t. P$	for all x of type t , P
existential quantification	$\exists x : t. P$	for some x of type t , P

Propositions are used all the time in mathematical reasoning. Often propositions are formulated informally using natural language.

For every proposition but \perp , there is a basic form of proof:

- To prove a conjunction $P \wedge Q$, one proves both P and Q .
- To prove a disjunction $P \vee Q$, one proves either P or Q .
- To prove an implication $P \rightarrow Q$, one describes a function that given a proof of P yields a proof of Q .
- To prove a universal quantification $\forall x : t. P$, one describes a function that given a value x of type t yields a proof of P .
- To prove an existential quantification $\exists x : t. P$, one gives a value x of type t and a proof of P .

9 Inductive Correctness Proofs

Usually propositions and proofs are written in natural language using symbolic notation where convenient. There are typed functional programming languages in which one can write propositions and proofs, and these languages are implemented with proof assistants helping with the construction of proofs.¹

9.2 List induction

Many properties of the basic list functions defined in Chapter 4 can be expressed with equations. The functions for list concatenation and list reversal, for instance, satisfy the following equations:

$$\begin{aligned}(l_1 @ l_2) @ l_3 &= l_1 @ (l_2 @ l_3) \\ rev(l_1 @ l_2) &= rev l_2 @ rev l_1 \\ rev(rev l) &= l\end{aligned}$$

The letters l_1 , l_2 , l_3 , and l act as variables ranging over lists over some base type α . With all typed quantifications made explicit, the second equation, for instance, becomes

$$\forall \alpha : \mathbb{T}. \forall l_1 : \mathcal{L}(\alpha). \forall l_2 : \mathcal{L}(\alpha). rev(l_1 @ l_2) = rev l_2 @ rev l_1$$

where the symbol \mathbb{T} represents the type of types. We will usually not give the types of the variables, but for the correctness of the mathematical reasoning it is essential that types can be assigned consistently.

The equations can be shown by equational reasoning using the defining equations of the functions and a recursive proof technique called list induction.

Fact 9.2.1 (List induction) To show that a proposition $p(l)$ holds for all lists l , it suffices to show the following propositions:

1. *Nil case:* $p([])$
2. *Cons case:* $\forall x. \forall l. p(l) \rightarrow p(x :: l)$

Proof We assume proofs of the nil case and the cons case. Let l be a list. We prove $p(l)$ by case analysis and recursion on l . If $l = []$, the nil case gives us a proof. If $l = x :: l'$, recursion gives us a proof of $p(l')$. Now the function proving the cons case gives us a proof of $p(l)$. ■

The predicate p is known as **induction predicate**,² and the proposition $p(l)$ in the cons case is known as **inductive hypothesis**.

¹A popular proof assistant we use in Saarbrücken is the Coq proof assistant.

²A predicate may be thought of as a function that yields a proposition.

9 Inductive Correctness Proofs

We will demonstrate equational reasoning and the use of list induction by proving some properties of list concatenation. We will make extensive use of the defining equations for @ :

$$\begin{aligned} @ & : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \\ [] @ l_2 & := l_2 \\ (x :: l_1) @ l_2 & := x :: (l_1 @ l_2) \end{aligned}$$

We say that an equation follows by **simplification** if its two sides can be made equal by applying defining equations (from left to right). For instance,

$$[1, 2] @ ([3, 5] @ [5, 6]) = [1, 2, 3, 5] @ [5, 6]$$

follows by simplification (both sides simplify to $[1, 2, 3, 5, 5, 6]$).

Once variables are involved, simplification is often not enough. For instance, $[] @ l = l$ follows by simplification, but $l @ [] = l$ does not. To prove $l @ [] = l$, we need list induction in addition to simplification.

Fact $l @ [] = l$.

Proof By list induction on l .

- Nil case: $[] @ [] = []$ holds by simplification.
- Cons case: We assume a proof of $l @ [] = l$ and show $(x :: l) @ [] = x :: l$. By simplification it suffices to prove $x :: (l @ []) = x :: l$. Holds by the inductive hypothesis, which gives us a proof of $l @ [] = l$. ■

The above proof is quite wordy for a routine argument. We will adopt a more compact format.

Fact 9.2.2 $l @ [] = l$.

Proof Induction on l . The nil case holds by simplification. Cons case:

$$\begin{array}{ll} (x :: l) @ [] = x :: l & \text{simplification} \\ x :: (l @ []) & \text{induction} \\ x :: l & \blacksquare \end{array}$$

The compact format arranges things such that the inductive hypothesis is identical with the claim that is proven. This way there is no need to write down the inductive hypothesis.

Our second example for an inductive proof concerns the *associativity* of list concatenation.

9 Inductive Correctness Proofs

Fact 9.2.3 $(l_1 @ l_2) @ l_3 = l_1 @ (l_2 @ l_3)$.

Proof Induction on l_1 . The nil case holds by simplification. Cons case:

$$\begin{aligned} ((x :: l_1) @ l_2) @ l_3 &= (x :: l_1) @ (l_2 @ l_3) && \text{simplification} \\ x :: ((l_1 @ l_2) @ l_3) &= x :: (l_1 @ (l_2 @ l_3)) && \text{induction} \\ x :: (l_1 @ (l_2 @ l_3)) &&& \blacksquare \end{aligned}$$

Exercise 9.2.4 Prove $\text{map } f (l_1 @ l_2) = \text{map } f l_1 @ \text{map } f l_2$.

9.3 Properties of List Reversal

We now use equational reasoning and list induction to prove equational properties of list reversal. For some cases a quantified inductive hypothesis is needed, an important feature we have not seen before. We start with the definition of naive list reversal:

$$\begin{aligned} \text{rev} &: \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \\ \text{rev } [] &:= [] \\ \text{rev } (x :: l) &:= \text{rev } (l) @ [x] \end{aligned}$$

We prove a distribution property for list reversal and list concatenation. The proof requires lemmas for list concatenation we have shown before.

Fact 9.3.1 $\text{rev } (l_1 @ l_2) = \text{rev } l_2 @ \text{rev } l_1$.

Proof Induction on l_1 . Nil case:

$$\begin{aligned} \text{rev } ([] @ l_2) &= \text{rev } l_2 @ \text{rev } [] && \text{simplification} \\ \text{rev } l_2 &= \text{rev } l_2 @ [] && \text{Fact 9.2.2} \end{aligned}$$

Cons case:

$$\begin{aligned} \text{rev } ((x :: l_1) @ l_2) &= \text{rev } l_2 @ \text{rev } (x :: l_1) && \text{simplification} \\ \text{rev } (l_1 @ l_2) @ [x] &= \text{rev } l_2 @ (\text{rev } l_1 @ [x]) && \text{induction} \\ (\text{rev } l_2 @ \text{rev } l_1) @ [x] &&& \text{Fact 9.2.3 } \blacksquare \end{aligned}$$

Next we show that naive lists reversal agrees with tail-recursive list reversal. Recall the definition of reversing concatenation (rev_append):

$$\begin{aligned} @_r &: \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \\ [] @_r l_2 &:= l_2 \\ (x :: l_1) @_r l_2 &:= l_1 @_r (x :: l_2) \end{aligned}$$

9 Inductive Correctness Proofs

Fact 9.3.2 $l_1 @_r l_2 = rev\ l_1 @\ l_2$.

Proof We prove $\forall l_2. l_1 @_r l_2 = rev\ l_1 @\ l_2$ by induction on l_1 . The quantification of l_2 is needed so we get a strong enough inductive hypothesis.

The nil case holds by simplification. Cons case:

$$\begin{array}{ll}
 (x :: l_1) @_r l_2 = rev\ (x :: l_1) @\ l_2 & \text{simplification} \\
 l_1 @_r (x :: l_2) = (rev\ l_1 @\ [x]) @\ l_2 & \text{induction} \\
 rev\ l_1 @\ (x :: l_2) & \text{Fact 9.2.3} \\
 & rev\ l_1 @\ ([x] @\ l_2) \quad \text{simplification}
 \end{array}$$

Note that the inductive hypothesis is used with the instance $l_2 := x :: l_2$. ■

The above proof uses a **quantified inductive hypothesis**, a feature we have not seen before. The induction predicate for the proof is $\lambda l_1. \forall l_2. l_1 @_r l_2 = rev\ l_1 @\ l_2$. The quantification of l_2 is essential since the function $l_1 @_r l_2$ changes both arguments upon recursion. While induction on l_1 takes care of the change of l_1 , quantification takes care of the change of l_2 .

Fact 9.3.3 $rev\ l = l @_r []$.

Proof Follows with Facts 9.3.2 and 9.2.2. ■

Fact 9.3.4 (Self inversion) $rev\ (rev\ l) = l$.

Proof By induction on l using Fact 9.3.1. ■

Tail-recursive linear-time list concatenation

With reversing concatenation $l_1 @_r l_2$ we can define a function concatenating two lists using only tail-recursion and taking running time linear in the length of the first list:

$$con\ l_1\ l_2 := (l_1 @_r []) @_r l_2$$

Fact 9.3.5 $l_1 @\ l_2 = (l_1 @_r []) @_r l_2$.

Proof Follows with Facts 9.3.2, 9.3.3, and 9.3.4. ■

Exercise 9.3.6 Give the induction predicate for the proof of Fact 9.3.2.

Exercise 9.3.7 Prove $rev\ (rev\ l) = l$.

9 Inductive Correctness Proofs

Exercise 9.3.8 Prove $\text{map } f (\text{rev } l) = \text{rev} (\text{map } f l)$.

Exercise 9.3.9 In this exercise we will write $|l|$ for $\text{length } l$ for better mathematical readability. Prove the following properties of the length of lists:

- a) $|l_1 @ l_2| = |l_1| + |l_2|$.
- b) $|\text{rev } l| = |l|$.
- c) $|\text{map } f l| = |l|$.

Exercise 9.3.10 (Tail-recursive list concatenation)

Define a function *con* concatenating two lists using only tail-recursion whose running time is linear in the length of the first list.

Exercise 9.3.11 Give the running time function for *con* l_1 l_2 and the length of l_1 not using recursion.

Exercise 9.3.12 (Power lists)

Consider a power list function defined as follows:

$$\begin{aligned} \text{pow} &: \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\mathcal{L}(\alpha)) \\ \text{pow } [] &:= [[]] \\ \text{pow } (x :: l) &:= \text{pow } l @ \text{map } (\lambda l. x :: l) (\text{pow } l) \end{aligned}$$

- a) Prove $|\text{pow } l| = 2^{|l|}$ where $|l|$ is notation for $\text{length } l$. The equation says that a list of length n has 2^n sublists.
- b) Give the running time function for function *pow*.
- c) Give the running time function for a variant of *pow* which avoids the binary recursion with a let expression.

Exercise 9.3.13 (Tail-recursive length of lists) Consider the tail-recursive function

$$\begin{aligned} \text{len } [] a &= a \\ \text{len } (x :: l) a &= \text{len } l (a + 1) \end{aligned}$$

Prove $\text{len } l a = a + \text{length } l$ by induction on l . Note that a needs to be quantified in the inductive hypothesis. Give the induction predicate.

9.4 Natural Number Induction

Inductive proofs are not limited to lists. The most basic induction principle is induction on natural numbers, commonly known as *mathematical induction*.

9 Inductive Correctness Proofs

Fact 9.4.1 (Mathematical induction) To show that a proposition $p(n)$ holds for all natural numbers n , it suffices to show the following propositions:

- *Zero case:* $p(0)$
- *Successor case:* $\forall n. p(n) \rightarrow p(n + 1)$

Proof A proof of the successor case provides us with a function that for every number n and for every proof of $p(n)$ yields a proof of $p(n + 1)$. Given proofs of the zero case and the successor case, we can construct for all numbers n a proof of $p(n)$ by case analysis and recursion on n :

- If $n = 0$, the zero case provides a proof of $p(n)$.
- If $n = n' + 1$, we obtain a proof of $p(n')$ by recursion. The function provided by the successor case then gives us a proof of $p(n' + 1)$, which is a proof of $p(n)$. ■

If you are familiar with mathematical induction, you may be wondering why we are using the terms “zero case” and “successor case” in place of the standard terms “base case” and “induction step”. The reason is that we want to emphasize the similarity between list induction and mathematical induction. In fact, we can see natural numbers as values that are obtained with the constructors zero and successor, similar to how lists are obtained with the constructors nil and cons. Written with constructors, the number 3 takes the form $\text{succ}(\text{succ}(\text{succ } 0))$, where the successor constructor satisfies $\text{succ } n = n + 1$.

We demonstrate number induction with the function

$$\begin{aligned} \text{sum} &: \mathbb{N} \rightarrow \mathbb{N} \\ \text{sum } 0 &:= 0 \\ \text{sum } (n + 1) &:= \text{sum } n + (n + 1) \end{aligned}$$

Fact 9.4.2 $2 \cdot \text{sum } n = n \cdot (n + 1)$.

Proof By induction on n . The zero case holds by simplification.

We argue the successor case:

$$\begin{aligned} 2 \cdot \text{sum } (n + 1) &= (n + 1) \cdot ((n + 1) + 1) && \text{simplification} \\ 2 \cdot \text{sum } n + 2 \cdot n + 2 &= (n + 1) \cdot (n + 2) && \text{induction} \\ n \cdot (n + 1) + 2 \cdot n + 2 &&& \text{simplification} \quad \blacksquare \end{aligned}$$

Exercise 9.4.3 (Little Gauss formula) Prove the following equations by induction on n .

9 Inductive Correctness Proofs

a) $0 + 1 + \cdots + n = n \cdot (n + 1)/2$

b) $0 + 1 + \cdots + n = \text{sum } n$

Equation (a) is known as *little Gauss formula* and gives us an efficient method to compute the sum of all numbers up to n by hand. For instance, the sum of all numbers up to 100 is $100 \cdot 101/2 = 5050$. The second equation tells us that the mathematical notation $0 + 1 + \cdots + n$ may be defined with the recursive function *sum*.

Exercise 9.4.4 (Running time functions)

In Chapter 8, we derived the following running time functions:

$$\begin{aligned} r_1(0) &:= 1 & r_2(0) &:= 1 \\ r_1(n+1) &:= 1 + r_1(n) & r_2(n+1) &:= 1 + r_2(n) + (n+1) \\ r_3(0) &:= 1 \\ r_3(n+1) &:= 1 + r_3(n) + r_3(n) \end{aligned}$$

Using a recurrence solver, we obtained explicit characterizations of the functions:

$$\begin{aligned} r_1(n) &= n + 1 & r_2(n) &= (n + 1)(n + 2)/2 \\ r_3(n) &= 2^{n+1} - 1 \end{aligned}$$

Use mathematical induction to verify the correctness of the explicit characterizations. Note that the proof obligations imposed by mathematical induction boil down to checking that the explicit characterizations satisfy the defining equations of the running time functions.

Exercise 9.4.5 (Sum of odd numbers)

Define a function $f : \mathbb{N}^+ \rightarrow \mathbb{N}^+$ computing the sum $1 + 3 + \cdots + (2n - 1)$ of the odd numbers from 1 to $2n - 1$. Prove $f(n) = n^2$.

Exercise 9.4.6 (Sum of square numbers)

Define a function $f : \mathbb{N}^+ \rightarrow \mathbb{N}^+$ computing the sum $0^2 + 1^2 + \cdots + n^2$ of the square numbers. Prove $f(n) = n \cdot (2n^2 + 3n + 1)/6$.

9.5 Correctness of Tail-Recursive Formulations

Given a recursive function that is not tail-recursive, one can often come up with a tail-recursive version collecting intermediate results in additional accumulator arguments. One then would like to prove that the tail-recursive version agrees with the original version. We already saw

9 Inductive Correctness Proofs

such a proof in §9.3 for the tail-recursive version of list-reversal. For list reversal we have to prove

$$\text{rev } l = l @_r []$$

using list recursion. The induction will only go through if we generalize the correctness statement to

$$l_1 @_r l_2 = \text{rev } l_1 @ l_2$$

If we think about it, the generalized correctness statement amounts to an equational specification of $l_1 @_r l_2$. Since the tail-recursive function $l_1 @_r l_2$ recurses on l_1 and modifies l_2 , a proof by list induction on l_1 quantifying the accumulator arguments l_2 works.

Recall the Fibonacci function from §3.3:

$$\begin{aligned} \text{fib} &: \mathbb{N} \rightarrow \mathbb{N} \\ \text{fib}(0) &:= 0 \\ \text{fib}(1) &:= 1 \\ \text{fib}(n+2) &:= \text{fib}(n) + \text{fib}(n+1) \end{aligned}$$

One speaks of a binary recursion since there are two parallel recursive applications in the 3rd defining equation. It is known that the running time of *fib* is exponential in the sense that it is not bounded by any polynomial.

To come up with a tail-recursive version of *fib*, we observe that a Fibonacci number is obtained as the sum of the preceding two Fibonacci numbers starting from the initial Fibonacci numbers 0 and 1. In §3.3 this fact is used to obtain the Fibonacci numbers with iteration. We now realize the iteration using two accumulator arguments for the preceding Fibonacci numbers:

$$\begin{aligned} \text{fib}_o &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ \text{fib}_o \ 0 \ a \ b &:= a \\ \text{fib}_o \ (n+1) \ a \ b &:= \text{fib}_o \ n \ b \ (a+b) \end{aligned}$$

We would like to show

$$\text{fib } n = \text{fib}_o \ n \ 0 \ 1$$

A proof by induction on n will not go through since *fib_o* modifies its two accumulator arguments. What we need is a more general equation specifying *fib_o* more accurately:

$$\text{fib}_o \ n \ (\text{fib } m) \ (\text{fib } (m+1)) = \text{fib } (n+m)$$

9 Inductive Correctness Proofs

We prove the equation by induction on n with m quantified. The zero case holds by simplification, and the successor case goes as follows:

$$\begin{array}{ll}
 \text{fib } (n + 1) (\text{fib } m) (\text{fib } (m + 1)) = \text{fib } (n + 1 + m) & \text{simplification} \\
 \text{fib } n (\text{fib } (m + 1)) (\text{fib } m + \text{fib } (m + 1)) & \text{def. eq. fib } \leftarrow \\
 \text{fib } n (\text{fib } (m + 1)) (\text{fib } (m + 2)) & \text{induction} \\
 \text{fib } (n + (m + 1)) & \text{simplification}
 \end{array}$$

Note the right-to-left application of the 3rd defining equation of *fib*.

Exercise 9.5.1 (Tail-recursive factorial function)

Consider the tail-recursive function

$$\begin{aligned}
 \text{fac} &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
 \text{fac } 0 \ a &:= a \\
 \text{fac } (n + 1) \ a &:= \text{fac } n \ ((n + 1) \cdot a)
 \end{aligned}$$

Prove $\text{fac } n \ a = a \cdot n!$ by induction on n assuming the equations $0! = 1$ and $(n + 1)! = (n + 1) \cdot n!$. Note that a needs to be quantified in the inductive hypothesis. Give the induction predicate.

Exercise 9.5.2 (Tail-recursive power function)

Consider the tail-recursive function

$$\begin{aligned}
 \text{pow} &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
 \text{pow } x \ 0 \ a &:= a \\
 \text{pow } x \ (n + 1) \ a &:= \text{pow } x \ n \ (x \cdot a)
 \end{aligned}$$

Prove $\text{pow } x \ n \ a = x^n \cdot a$ by induction on n . Note that a needs to be quantified in the inductive hypothesis. Give the induction predicate.

Exercise 9.5.3 (Tail-recursive sum) Define a tail-recursive variant sum' of sum and prove $\text{sum}' \ n \ 0 = \text{sum } n$.

Exercise 9.5.4 There is an alternative proof of the correctness of *fib* using a lemma for *fib*.

a) Prove $\text{fib } (n + 2) \ a \ b = \text{fib } n \ a \ b + \text{fib } (n + 1) \ a \ b$.

b) Prove $\text{fib } n = \text{fib } n \ 0 \ 1$ by induction on n using (a).

Hint: (a) follows by induction on n with a and b quantified.

9.6 Properties of Iteration

Using number induction, we will prove some properties of iteration §3.2.

We start with the iteration function

$$\begin{aligned} it &: \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \mathbb{N} \rightarrow \alpha \rightarrow \alpha \\ it\ f\ 0\ x &:= x \\ it\ f\ (n+1)\ x &:= f(it\ f\ n\ x) \end{aligned}$$

Note that *it* is defined differently from the tail-recursive iteration function *iter* in §3.2. We will eventually show that both iteration functions agree.

We first show that factorials

$$\begin{aligned} !0 &:= 1 \\ !(n+1) &:= (n+1) \cdot n! \end{aligned}$$

can be computed with iteration.

Fact 9.6.1 $(n, n!) = it\ (\lambda(k, a). (k+1, (k+1) \cdot a))\ n\ (0, 1)$.

Proof We define $f(k, a) := (k+1, (k+1) \cdot a)$ and prove

$$(n, n!) = it\ f\ n\ (0, 1)$$

by induction on n . The zero case holds by simplification. We argue the successor case:

$$\begin{aligned} (n+1, (n+1)!) &= it\ f\ (n+1)\ (0, 1) && \text{simplification} \\ f(n, n!) &= f(it\ f\ n\ (0, 1)) && \text{induction} \quad \blacksquare \end{aligned}$$

Next we prove an important property of *it* that we need for showing the agreement with *iter*.

Fact 9.6.2 (Shift Law) $f(it\ f\ n\ x) = it\ f\ n\ (f\ x)$.

Proof Induction on n . The zero case holds by simplification.

Successor case:

$$\begin{aligned} f(it\ f\ (n+1)\ x) &= it\ f\ (n+1)\ (f\ x) && \text{simplification} \\ f(f(it\ f\ n\ x)) &= f(it\ f\ n\ (f\ x)) && \text{induction} \quad \blacksquare \end{aligned}$$

Recall the tail-recursive iteration function *iter* from §3.2:

$$\begin{aligned} iter\ f\ 0\ x &:= x \\ iter\ f\ (n+1)\ x &:= iter\ f\ n\ (f\ x) \end{aligned}$$

9 Inductive Correctness Proofs

Fact 9.6.3 (Agreement) $it\ f\ n\ x = iter\ f\ n\ x$.

Proof We prove $\forall x. it\ f\ n\ x = iter\ f\ n\ x$ by induction on n . The quantification of x is needed since $iter$ changes this argument upon recursion. The zero case holds by simplification. Successor case:

$$\begin{array}{ll} it\ f\ (n + 1)\ x = iter\ f\ (n + 1)\ x & \text{simplification} \\ f(it\ f\ n\ x) = iter\ f\ n\ (f\ x) & \text{shift law} \\ it\ f\ n\ (f\ x) & \text{induction} \quad \blacksquare \end{array}$$

Exercise 9.6.4 Prove $x^n = it\ (\lambda a. a \cdot x)\ n\ 1$.

Exercise 9.6.5 Recall the Fibonacci function fib from §9.5. Prove $(fib\ n, fib\ (n + 1)) = it\ f\ n\ (0, 1)$ for $f(a, b) := (b, a + b)$.

Exercise 9.6.6 Prove the shift law for $iter$ in two ways: (1) by a direct inductive proof, and (2) using the agreement with it and the shift law for it .

9.7 Induction for Terminating Functions

Given a terminating recursive function, and a property concerning the results of the function, we can prove the property following the defining equations of the function and assuming the property for the recursive calls of the function. Such a proof is an inductive proof following the recursion scheme underlying the defining equations. We speak of **function induction**.

Our first example for function induction concerns the remainder operation:

$$\begin{array}{l} \% : \mathbb{N} \rightarrow \mathbb{N}^+ \rightarrow \mathbb{N} \\ x \% y := x \quad \text{if } x < y \\ x \% y := (x - y) \% y \quad \text{if } x \geq y \end{array}$$

Note that the type of $\%$ ensures that $y > 0$ in $x \% y$, a property that in turn ensures termination of the function. From the definition it seems that we have $x \% y < y$. A proof of this property needs induction to account for the recursion of the second defining equation.

Fact $x \% y < y$.

9 Inductive Correctness Proofs

Proof By induction on $x \% y$. If $x \% y$ is obtained with the first defining equation, we have $x < y$ and $x \% y = x$, which yields the claim. If $x \% y$ is obtained with the second defining equation, the claim follows by induction. More precisely, we have $x \% y = (x - y) \% y$ and $(x - y) \% y < y$ by induction for the recursive call. ■

The induction used in the proof is function induction. A proof by function induction follows the case analysis established by the defining equations and assumes that the claim holds for the recursive applications. The use of function induction is admissible for functions with terminating defining equations. What happens is that function induction constructs a recursive proof following the terminating recursion of the defining equations.

We impose the restriction that the proposition to be shown by function induction contains exactly one application of the function governing the induction. A proof by function induction now case analyses the function application following the defining equations of the function. By simplifying with the defining equations recursive calls of the function are introduced, and for these calls we do have the inductive hypothesis.

Using the above example, we introduce a more formalized way to write down proofs by function induction.

Fact 9.7.1 $x \% y < y$.

Proof By induction on $x \% y$.

1. $x \% y = x$ and $x < y$. The claim follows.
2. $x \% y = (x - y) \% y$ and $x \geq y$. By induction we have $(x - y) \% y < y$. The claim follows. ■

Exercise 9.7.2 Let $y > 0$. Prove $x = (x/y) \cdot y + x \% y$ by function induction on x/y . First write down the type and the defining equations for x/y and $x \% y$.

9.8 Euclid's Algorithm

There is an elegant algorithm known as *Euclid's algorithm* computing the greatest common divisor of two integers $x, y \geq 0$. The algorithm is based on two straightforward rules:

1. If one of the numbers is 0, the other number is the greatest common divisor of the two numbers.
2. If $0 < y \leq x$, replace x with $x - y$.

9 Inductive Correctness Proofs

Note that either the first or the second rule is applicable, and that application of the second rule terminates since the second rule decreases the sum of the two numbers.

We now recall that $x \% y$ is obtained by subtracting y from x as long as $y \leq x$. This gives us the following function for computing greatest common divisors:

$$\begin{aligned} \text{gcd} &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ \text{gcd } x \ 0 &:= x \\ \text{gcd } x \ y &:= \text{gcd } y \ (x \% y) \quad \text{if } y > 0 \end{aligned}$$

The function terminates since recursion decreases the second argument (ensured by Fact 9.7.1). Here is a trace:

$$\text{gcd } 91 \ 35 = \text{gcd } 35 \ 21 = \text{gcd } 21 \ 14 = \text{gcd } 14 \ 7 = \text{gcd } 7 \ 0 = 7$$

Our goal here is to carefully explain how the algorithm can be derived from basic principles so that the correctness of the algorithm is obvious.

In this section, all numbers will be nonnegative integers. We first define the **divides relation** for numbers:

$$k \mid x := \exists n. x = n \cdot k \qquad k \text{ divides } x$$

By our convention, k , x , and n are all nonnegative integers. When we have $k \mid x$, we say that k is a **divisor** or a **factor** of x . Moreover, when we say that k is a **common divisor** of x and y we mean that k is a divisor of both x and y .

We observe that every number x has 1 and x as divisors. Moreover, a positive number x has only finitely many divisors, which are all between 1 and x . In contrast, 0 has infinitely many divisors since it is divided by every number. It follows that two numbers have a greatest common divisor if and only if not both numbers are zero.

We call a number z the **GCD** of two numbers x and y if the divisors of z are exactly the common divisors of x and y . Here are examples:

- The GCD of 15 and 63 is 3.
- The GCD of 0 and 0 is 0.

More generally, we observe the following facts.

Fact 9.8.1 (GCD)

1. The GCD of two numbers uniquely exists.
2. The GCD of x and x is x .

9 Inductive Correctness Proofs

3. *Zero rule:* The GCD of x and 0 is x .
4. *Symmetry:* The GCD of x and y is the GCD of y and x .
5. The GCD of two numbers that are not both 0 is the greatest common divisor of the numbers.

Our goal is to find an algorithm that computes the GCD of two numbers. If one of the two numbers is 0, we are done since the other number is the GCD of the two numbers. It remains to find a rule that given two positive numbers x and y yields two numbers x' and y' that have the same GCD and whose sum is smaller (i.e., $x + y > x' + y'$). We then can use recursion to reduce the initial numbers to the trivial case where one of the numbers is 0.

It turns out that the GCD of two number $x \geq y$ stays unchanged if x is replaced with the difference $x - y$.

Fact 9.8.2 (Subtraction Rule) If $x \geq y$, then the common divisors of x and y are exactly the common divisors of $x - y$ and y .

Proof We need to show two directions.

Suppose $x = n \cdot a$ and $y = n \cdot b$. We have $x - y = n \cdot a - n \cdot b = n \cdot (a - b)$.

Suppose $x - y = n \cdot a$ and $y = n \cdot b$. We have $x = x - y + y = n \cdot a + n \cdot b = n \cdot (a + b)$. ■

Fact 9.8.3 (Modulo Rule) If $y > 0$, then the common divisors of x and y are exactly the common divisors of $x \% y$ and y .

Proof Follows with the subtraction rule (Fact 9.8.2) since $x \% y$ is obtained from x by repeated subtraction of y . More formally, we argue the claim by function induction on $x \% y$.

1. $x \% y = x$ and $x < y$. The claim follows.
2. $x \% y = (x - y) \% y$ and $x \geq y$. By induction we have that the common divisors of $x - y$ and y are the common divisors of $(x - y) \% y$ and y . The claim follows with the subtraction rule (Fact 9.8.2). ■

Euclid's algorithm now simply applies the modulo rule until the zero rule applies, using the symmetry rule when needed. Recall our formulation of Euclid's algorithm with the function

$$\begin{aligned} \text{gcd} &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ \text{gcd } x \ 0 &:= x \\ \text{gcd } x \ y &:= \text{gcd } y \ (x \% y) \quad \text{if } y > 0 \end{aligned}$$

Recall that gcd terminates since recursion decreases the second argument (ensured by Fact 9.7.1).

9 Inductive Correctness Proofs

Fact 9.8.4 (Correctness) $\text{gcd } x \ y$ is the GCD of x and y .

Proof By function induction on $\text{gcd } x \ y$.

1. $\text{gcd } x \ 0 = x$. Claim follows by zero rule.
2. $\text{gcd } x \ y = \text{gcd } y \ (x \% y)$ and $y > 0$. By induction we know that $\text{gcd } y \ (x \% y)$ is the GCD of y and $x \% y$. Hence $\text{gcd } x \ y$ is the GCD of y and $x \% y$. It follows with the modulo rule and the symmetry rule that $\text{gcd } x \ y$ is the GCD of x and y . ■

Running Time

The running time of $\text{gcd } x \ y$ is at most y since each recursion step decreases y . In fact, one can show that the asymptotic running time of $\text{gcd } x \ y$ is logarithmic in y . For both considerations we exploit that $x \% y$ is realized as a constant-time operation on usual computers. One can also show that gcd at least halves its second argument after two recursion steps. For an example, see the trace at the beginning of this section.

Exercise 9.8.5 Define a function $g : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ computing the GCD of two numbers whose recursion decreases the first argument.

Exercise 9.8.6 Define a function $g : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ computing the GCD of two numbers using subtraction and not using the remainder operation. Argue the correctness of your function. Hint: Define the function such that recursion decreases the sum of the two arguments.

Exercise 9.8.7 Define a function $g : \mathbb{N}^+ \rightarrow \mathbb{N}^+ \rightarrow \mathbb{N}^+$ computing the GCD of two positive numbers using the remainder function. Argue the correctness of your function. Use only one conditional.

Exercise 9.8.8 Define a function $g : \mathcal{L}(\mathbb{N}) \rightarrow \mathbb{N}$ such that the common divisors of the elements of a list l are exactly the divisors of $g(l)$.

Exercise 9.8.9 Argue the correctness of the claims of Fact 9.8.1.

9.9 Complete Induction

Complete induction is a generalization of mathematical induction making it possible to assume in a proof of $p(n)$ a proof of $p(k)$ for all $k < n$. Complete induction may be seen as a natural scheme for recursive proofs where the recursion is on natural numbers. Using complete induction, we can justify the function inductions we have seen so far. While complete induction appears to be more powerful than mathematical induction, it can be obtained from mathematical induction with an elegant proof.

9 Inductive Correctness Proofs

Fact 9.9.1 (Complete induction) To show that a proposition $p(n)$ holds for all natural numbers n , it suffices to show the following proposition:

$$\forall n. (\forall k < n. p(k)) \rightarrow p(n)$$

Proof We assume $H_1 : \forall n. (\forall k < n. p(k)) \rightarrow p(n)$. To show $\forall n. p(n)$, we show more generally

$$\forall n. \forall k < n. p(k)$$

by induction on n . So the trick is to do mathematical induction on the upper bound introduced by the generalization.

The zero case $\forall k < 0. p(k)$ is obvious since there is no natural number $k < 0$.

In the successor case, we have the inductive hypothesis

$$H_2 : \forall k < n. p(k)$$

and show $\forall k < n + 1. p(k)$. We assume $H_3 : k < n + 1$ and prove $p(k)$. By H_1 it suffices to prove

$$\forall k' < k. p(k')$$

We assume $k' < k$ and prove $p(k')$. By H_3 we have $k' < n$. Now the claim follows by H_2 . ■

Informally, we may say that complete induction allows us to assume proofs of all propositions $p(k)$ with $k < n$ when we prove $p(n)$. In a certain sense, we get this inductive hypothesis for free, in the same way we get recursion for free when we define a function. In contrast to mathematical induction, complete induction does not impose a case analysis.

We can now explain the function induction on $x \% y$ used for Fact 9.7.1, Exercise 9.7.2, and Fact 9.8.3 with complete induction on x . Similarly, the function induction on $\text{gcd } x y$ used for Fact 9.8.4 can be explained with complete induction on y . In each case, the complete induction agrees with the termination argument for the underlying function ($x \% y$ decreases x , $\text{gcd } x y$ decreases y). The induction predicates for the complete inductions are as follows:

- Fact 9.7.1: $p(x) := x \% y < y$.
- Exercise 9.7.2: $p(x) := x = (x/y) \cdot y + x \% y$.
- Fact 9.8.3: $p(x) :=$ the cds^3 of x, y are the cds of $y, x \% y$.

³common divisors

9 Inductive Correctness Proofs

- Fact 9.8.4: $p(y) := \forall x. \text{gcd } x \ y$ is the GCD of x, y .

That x is quantified in the induction predicate for Fact 9.8.4 is necessary since gcd changes both arguments. We will not go further into logical details here but instead rely on our mathematical intuition.

We give three examples for the use of complete induction.

Fact 9.9.2 A list of length n has 2^n sublists.

Proof For $n = 0$, the claim is obvious. For $n > 0$, the list has the form $x :: l$. Every sublist of $x :: l$ now either keeps x or deletes x . A sublist of $x :: l$ that deletes x is a sublist of l . By complete induction we know that l has 2^{n-1} sublists. We now observe that the sublists of $x :: l$ that keep x are exactly the sublists of l with x added in front. This gives us that l has $2^{n-1} + 2^{n-1} = 2^n$ sublists. ■

Note that the proof suggests a function that given a list obtains a list of all sublists.

Recall that a prime number is an integer greater 1 that cannot be obtained as the product of two integers greater 1 (§3.4).

Fact 9.9.3 Every integer greater than 1 has a prime factorization.

Proof Let $x > 1$. If x is prime, then $[x]$ is a prime factorization of x . Otherwise, $x = x_1 \cdot x_2$ with $1 < x_1, x_2 < x$. By complete induction we have prime factorizations l_1 and l_2 for x_1 and x_2 . We observe that $l_1 @ l_2$ is a prime factorization of x . ■

Note that the proof suggest a function that given a number greater 1 obtains a prime factorization of the number. The function relies on a helper function that given $x > 1$ decides whether there are $x_1, x_2 > 1$ such that $x = x_1 \cdot x_2$.

Fact 9.9.4 (Euclidean division) Let $x \geq 0$ and $y > 0$. Then there exist $a \geq 0$ and $0 \leq b < y$ such that $x = a \cdot y + b$.

Proof By complete induction on x . If $x < y$, we satisfy the claim with $a = 0$ and $b = x$. Otherwise $x \geq y$. By the inductive hypothesis we have $x - y = a \cdot y + b$ with $a \geq 0$ and $0 \leq b < y$. We have $x = (a + 1) \cdot y + b$, which yields the claim. ■

Exercise 9.9.5 Declare a prime factorization function following the proof of Fact 9.9.3.

9 Inductive Correctness Proofs

Exercise 9.9.6 Declare a division function following the proof of Fact 9.9.4. Given x and y as required, the function should return a quotient-remainder pair (a, b) for x and y .

Exercise 9.9.7 Prove $fib(n) < 2^n$ by complete induction.

Exercise 9.9.8 Derive mathematical induction from complete induction.

9.10 Prime Factorization Revisited

We have discussed prime factorization algorithms in §4.15. We now review the correctness arguments in the light of the formal foundations we have laid out in this chapter. The verification of the optimized prime factorization algorithm will require invariants, an important feature we have not yet seen in this chapter.

We recall two important definitions:

$$\begin{aligned} k \mid x &:= \exists n. x = n \cdot k && k \text{ divides } x \\ \text{prime } x &:= x \geq 2 \wedge \neg \exists k, n \geq 2. x = k \cdot n \end{aligned}$$

The definitions assume that k and x are natural numbers.

We start with a straightforward prime factorization algorithm:

$$\begin{aligned} \text{pfac} &: \mathbb{N} \rightarrow \mathcal{L}(\mathbb{N}) \\ \text{pfac } x &:= \begin{cases} [] & \text{if } x < 2 \\ \text{LET } k = \text{first}(\lambda k. k \mid x) \text{ 2 IN } k :: \text{pfac}(x/k) & \text{if } x \geq 2 \end{cases} \end{aligned}$$

We first argue the termination of pfac . The linear search realized with first terminates since $x \geq 2$, the search starts with 2, and x divides x . The linear search yields a k such that $2 \leq k \leq x$ and thus $x/k < x$. Thus pfac terminates since it decreases its argument.

We now show

$$(x < 2 \wedge \text{pfac } x = []) \vee (x \geq 2 \wedge \text{pfac } x \text{ is prime fact. of } x)$$

by functional induction on $\text{pfac } x$. The interesting case is $x \geq 2$. The linear search gives us the least prime factor k of x satisfying $2 \leq k \leq x$. We have either $k < x$ or $k = x$. Thus either $x/k \geq 2$ or $x/k = 1$. With the inductive hypothesis it now follows that $k :: \text{pfac}(x/k)$ is the prime factorization of x .

9 Inductive Correctness Proofs

Next we consider an optimized prime factorization algorithm:

$$\begin{aligned}
 & pfac' : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathcal{L}(\mathbb{N}) \\
 & pfac' \ k \ x := \begin{cases} [x] & \text{if } k^2 > x \\ k :: pfac' \ k \ (x/k) & \text{if } k^2 \leq x \text{ and } k \mid x \\ pfac' \ (k+1) \ x & \text{if } k^2 \leq x \text{ and } \neg(k \mid x) \end{cases}
 \end{aligned}$$

We would like to show that $pfac' \ 2 \ x$ yields a prime factorization of x if $x \geq 2$. Since k (the counter) is modified upon recursion, it is clear that $pfac' \ 2 \ x$ cannot be verified by itself and we have to look more generally at $pfac' \ k \ x$ where the arguments satisfy some safety condition. The safety condition must be satisfied by the initial arguments and must be preserved for the recursive calls identified by the defining equations of $pfac'$. Safety conditions that are preserved for the recursive calls of a function are called **invariants**. Safety conditions act as preconditions for a function specifying the admissible arguments of the function.

So far, all functions we did correctness proofs for did terminate for all arguments admitted by the types of the functions. This is not the case for $pfac'$, which diverges for $x = 1$ and $k = 1$. Thus we need an invariant for $pfac'$ ensuring termination. We choose the condition

$$safe_1 \ k \ x := 2 \leq k \leq x$$

which is satisfied by the initial arguments and is preserved for both recursive calls. Thus $safe_1$ is an invariant for $pfac'$, as required by the termination argument. Given $safe_1 \ k \ x$, termination follows from the fact that the difference $x - k \geq 0$ is decreased upon recursion.

With the invariant $safe_1$ in place, we can use function induction on $pfac' \ k \ x$ to show that $pfac' \ k \ x$ always yields a factorization of x (that is, x is the product of the numbers in the list $pfac' \ k \ x$). Using function induction, we can also verify that the numbers in the list $pfac' \ k \ x$ appear in ascending order starting with the number 2.

It remains to show that all numbers in the list $pfac' \ k \ x$ are prime. For this to be true, the invariant $safe_1$ is not strong enough. To have this property, we need to ensure that k is a lower bound for all numbers $m \geq 2$ dividing x :

$$safe_2 \ k \ x := 2 \leq k \leq x \wedge \forall m \geq 2. \ m \mid x \rightarrow m \geq k$$

It is not difficult to verify that $safe_2 \ k \ x$ is an invariant for $pfac' \ k \ x$. With function induction we can now verify that every number in $pfac' \ k \ x$ is prime if $safe_2 \ k \ x$ is satisfied.

9 Inductive Correctness Proofs

To summarize, the essential new concept we need for the verification of the optimized prime factorization algorithm are invariants. An invariant is a precondition for the arguments of a function that is preserved under recursion. The idea then is that a function is only verified for arguments satisfying the invariant. This way we can exclude arguments for which the function does not terminate or operations are not defined (e.g., division or remainder for 0). We may see an invariant for a function as a refinement of the type of the function. Like the type of a function, an invariant for a function must be checked for every (recursive) application of the function. We may see invariant checking as a refined form of type checking that in contrast to ordinary type checking often cannot be done algorithmically.

Running times

Let us look at the running times of $pfac\ x$ and $pfac'\ 2\ x$ for the case where x is a prime number. Then $pfac'\ 2\ x$ will increment k starting from 2 until $k^2 > x$ and then finish. This give us a running time $O(\lceil\sqrt{x}\rceil)$. On the other hand, $pfac\ x$ will increment k starting from 2 until x in the recursion realizing *first*. This give us a running time $O(x)$. We conclude that the running time of $pfac\ x$ is quadratic in the running time of $pfac'\ 2\ x$.

Exercise 9.10.1 For each of the recursive applications of $pfac'$ give the condition one has to verify to establish $safe_2$ as an invariant.

Exercise 9.10.2 Convince yourself that the proposition $safe_2\ k\ x$ is equivalent to $k \geq 2 \wedge x \geq 2 \wedge \forall m \geq 2. m \mid x \rightarrow m \geq k$.

Exercise 9.10.3 Declare the functions $pfac$ and $pfac'$ in OCaml.

Exercise 9.10.4 Consider the function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ defined as

$$f(x) := \text{IF } x = 7 \text{ THEN } x \text{ ELSE } f(x - 1)$$

Give a predicate $p(x)$ holding exactly for the numbers f terminates on. Convince yourself that p is an invariant for f .

Exercise 9.10.5 Consider the function $f : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$ defined as

$$f\ x\ y := \text{IF } x = 0 \wedge y = 27 \text{ THEN } x + y \text{ ELSE } f(x - 1)(y + 1)$$

Give an invariant pxy for f holding exactly for the numbers f terminates on.

9 Inductive Correctness Proofs

Exercise 9.10.6 Consider the function $f : \mathbb{N}^+ \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ defined as

$$f\ n\ k := \text{IF } k/n = 2 \text{ THEN } k \text{ ELSE } f\ n\ (k + 1)$$

- a) Give an invariant $p\ n\ k$ for f holding exactly for the numbers f terminates on.
- b) Give an invariant $p\ n\ k$ for f that is strong enough to show $f\ n\ 0 = 2 \cdot n$.

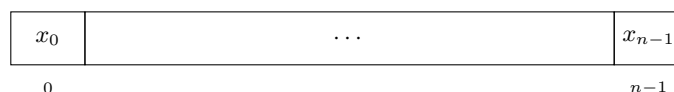
10 Arrays

Arrays are mutable data structures representing computer memory within programming languages. The mutability of arrays is in contrast to the immutability of values (e.g., numbers, lists). Mutability is an intrinsic aspect of physical objects while immutability is a characteristic feature of mathematical objects (a clock changes its state, but a number has no state that could change).

Arrays are made available through special values acting as names and through operations acting on arrays identified by array values. Due to their mutability, arrays don't have a direct mathematical representation, making functions using arrays more difficult to reason about.

10.1 Basic Array Operations

Arrays may be described as fixed-length mutable sequences with constant-time field access and update.



The length and the element type of an array are fixed at *allocation time* (the time an array comes into existence). The *fields* or *cells* of an array are boxes holding values known as *elements* of the array. The fields are numbered starting from 0. One speaks of the *indices* or *positions* of an array. The key difference between lists and arrays is that arrays are mutable in that the values in their fields can be overwritten by an update operation. Arrays are represented by special *array values* acting as immutable names for the mutable array objects in existence. Each time an array is allocated in the computer's memory, a fresh name identifying the new array is chosen.

OCaml provides arrays through the standard structure *Array*. The operation

$$\text{Array.make} : \forall \alpha. \text{int} \rightarrow \alpha \rightarrow \text{array}(\alpha)$$

allocates a fresh array of a given length where all fields are initialized with a given value. An *array type* has the form $\text{array}(t)$ and covers all

10 Arrays

arrays whose elements have type t , no matter what the length of the array is. There is also a special constructor type *unit*

$$\text{unit} ::= ()$$

with a single constructor $()$ serving as target type of the array update operation. There are access and update operations for the fields of an array:

$$\begin{aligned} \text{Array.get} &: \forall\alpha. \text{array}(\alpha) \rightarrow \text{int} \rightarrow \alpha \\ \text{Array.set} &: \forall\alpha. \text{array}(\alpha) \rightarrow \text{int} \rightarrow \alpha \rightarrow \text{unit} \end{aligned}$$

An operation $\text{Array.get } a \ i$ yields the i th element of a (i.e., the value in the i th field of the array a), and an operation $\text{Array.set } a \ i \ x$ updates the i th element of a to x (i.e., the value in the i th field of the array a is replaced with the value x). Both operations raise an invalid argument exception

Exception: `Invalid_argument "index out of bounds"`

if the given index is not admissible for the given array.

OCaml supports convenient notations for field access and field update:

$$\begin{aligned} e_1.(e_2) &\rightsquigarrow \text{Array.get } e_1 \ e_2 \\ e_1.(e_2) \leftarrow e_3 &\rightsquigarrow \text{Array.set } e_1 \ e_2 \ e_3 \end{aligned}$$

Let's step through a first demo:

```
1 let a = Array.make 3 7
2 let v_before = a.(1)
3 let _ = a.(1) <- 5
4 let v_after = a.(1)
```

The 1st declaration allocates an array a whose initial state is $[7, 7, 7]$. The 2nd declaration binds the identifier v_before to 7 (the 1st element of a at this point). The 3rd declaration updates a to the state $[7, 5, 7]$. The 4th declaration binds the identifier v_after to 5 (the 1st element of a at this point). Note that the identifier v_before is still bound to 7.

There is also a basic array operation that yields the length of an array:

$$\text{Array.length} : \forall\alpha. \text{array}(\alpha) \rightarrow \text{int}$$

As a convenience, OCaml offers the possibility to allocate an array with a given initial state:

10 Arrays

```
let a = [|1;2;3|]
```

This declaration binds a to a new array of length 3 whose fields hold the values 1, 2, 3. The declaration may be seen as derived syntax for

```
let a = Array.make 3 1
let _ = a.(1) <- 2
let _ = a.(2) <- 3
```

Sequentializations

OCaml offers expressions of the form

$$e_1; e_2$$

called **sequentializations**, which execute e_1 before e_2 and yield the value of e_2 . The value of e_1 is ignored. The reason e_1 is executed is that it may update an array, a so-called **side effect**. OCaml issues a warning in case the type of e_1 is not *unit*. We can now write the expression

```
let a = Array.make 3 1 in a.(1) <- 2; a.(2) <- 3; a
```

to explain the expression $[|1; 2; 3|]$. We remark that the sequentialization operator $e_1; e_2$ is treated as a right-associative operator so that iterated sequentializations $e_1; \dots; e_n$ can be written without parentheses.

Sequentializations may be considered as a derived form that can be expressed with a let expression $\text{LET } _ = e_1 \text{ IN } e_2$.

10.2 Conversion between Arrays and Lists

The state of an array can be represented as a list, and given a list, one can allocate a fresh array whose state is given by the list. OCaml provides the conversions with two predefined functions:

$$\begin{aligned} \text{Array.to_list} &: \forall \alpha. \text{array}(\alpha) \rightarrow \mathcal{L}(\alpha) \\ \text{Array.of_list} &: \forall \alpha. \mathcal{L}(\alpha) \rightarrow \text{array}(\alpha) \end{aligned}$$

At this point we need to say that OCaml provides an immutable empty array for every element type (i.e., an array of length 0 with no fields).

The conversion from arrays to lists can be defined with the operations *Array.length* and *Array.get*:

```
let to_list a =
  let rec loop i l =
    if i < 0 then l else loop (i-1) (a.(i)::l)
  in loop (Array.length a - 1) []
```

10 Arrays

The loop function shifts a pointer i from right to left through the array and collects the values in the scanned field in a list l .

The conversion from lists to arrays can be defined with the operations *Array.make* and *Array.set*:

```
let of_list l =
  match l with
  | [] -> [|]
  | x::l ->
    let a = Array.make (List.length l + 1) x in
    let rec loop l i =
      match l with
      | [] -> a
      | x::l -> (a.(i) <- x; loop l (i + 1))
    in loop l 1
```

Here the loop function recurses on the list and shifts a pointer from left to right through the array to copy the values from the list into the array. There is the subtlety that given an empty list we can obtain the corresponding empty array only with the expression `[|]` since *Array.make* requires a default value even if we ask for an empty array. Thus we consider `[|]` as a native constant rather than a derived form. In fact, the empty array `[|]` is an immutable value.

Note that the worker functions in both conversion functions are tail-recursive. We reserve the identifier *loop* for tail-recursive worker functions.

Exercise 10.2.1 (Cloning)

Declare a function $\forall\alpha. \text{array}(\alpha) \rightarrow \text{array}(\alpha)$ cloning arrays. Make sure your function also works for empty arrays. OCaml offers a cloning function as *Array.copy*.

Exercise 10.2.2 OCaml offers a predefined function *Array.init* such that *Array.init* n f allocates an array with initial state *List.init* n f . Declare such an *init* function for arrays using neither *List.init* nor *Array.of_list*. Use only tail recursion.

Exercise 10.2.3 Declare a function that yields the sum of the elements of an array over *int*.

10.3 Binary Search

Given a sorted array a and a value x , one check whether x occurs in a with a running time that is at most logarithmic in the length of the array. The trick is to check whether the value in the middle of the array

10 Arrays

is x , and in case it is not, continue with either the left half or the right half of the array. The calculation of the running time assumes that field access is constant time, which is the case for arrays. One speaks of binary search to distinguish the algorithm from linear search, which has linear running time.¹ We have seen binary search before in generalized form (§8.5).

Binary search is usually realized such that in the positive case it returns an index where the element searched for occurs.

To ease the presentation of binary search and related algorithms, we work with a three-valued constructor type

```
type comparison = LE | EQ | GR
```

and a polymorphic comparison function

```
let comp x y : comparison =  
  if x < y then LE  
  else if x = y then EQ else GR
```

The binary search function

$$bsearch : \forall \alpha. array(\alpha) \rightarrow \alpha \rightarrow \mathcal{O}(\mathbb{N})$$

can now be declared as follows:

```
let bsearch a x : int option =  
  let rec loop l r =  
    if l > r then None  
    else let m = (l+r)/2 in  
      match comp x a.(m) with  
      | LE -> loop l (m-1)  
      | EQ -> Some m  
      | GR -> loop (m+1) r  
  in loop 0 (Array.length a - 1)
```

Note that the worker function *loop* is tail-recursive and that the segment of the array to be considered is at least halved upon recursion.

Binary search satisfies two invariants:

1. The state of the array agrees with the initial state (that is, remains unchanged).
2. All elements to the left of l are smaller than x and all elements to the right of r are greater than x .

Given the invariants, the correctness of *bsearch* is easily verified.

¹Linear search generalizes binary search in that it doesn't assume the array is sorted.

10 Arrays

Exercise 10.3.1 (Linear search) Declare a function

$$lsearch : \forall \alpha. array(\alpha) \rightarrow \alpha \rightarrow \mathcal{O}(\mathbb{N})$$

that checks whether a value occurs in an array and in the positive case yields an index where the value occurs. The running time of *lsearch* should be $O(n)$ where n is the length of the array.

Exercise 10.3.2 (Binary search for increasing functions)

Declare a function

$$search : \forall \alpha. (\mathbb{N} \rightarrow \alpha) \rightarrow \mathbb{N} \rightarrow \alpha \rightarrow \mathcal{O}(\alpha)$$

that given an increasing function f , a number n , and a value x , checks whether there is some $i \leq n$ such that $f(i) = x$ using at most $O(\log n)$ calls of f . A function f is increasing if $f(i) \leq f(j)$ whenever $i \leq j$.

10.4 Reversal

There is an elegant algorithm reversing the elements of an array in time linear in the length of the array. The algorithm is based on a function

$$swap : \forall \alpha. array(\alpha) \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow unit$$

swapping the values at two given positions of a given array:

```
let swap a i j : unit =
  let x = a.(i) in a.(i) <- a.(j); a.(j) <- x
```

The algorithm now reverses a segment $i < j$ in an array by swapping the values at i and j and continuing with the smaller segment $(i + 1, j - 1)$:

```
let reverse a : unit =
  let rec loop i j =
    if i > j then ()
    else (swap a i j; loop (i+1) (j-1))
  in loop 0 (Array.length a - 1)
```

One may see i and j as pointers to fields of the array that are moved inside from the leftmost and the rightmost position.

Function *reverse* satisfies two invariants:

1. The state of the array agrees with the initial state up to repeated swapping.
2. The segments $l \leq i$ and $j \leq r$ agree with the reversed initial state.

Exercise 10.4.1 Declare a function *rotate* : $\forall \alpha. array(\alpha) \rightarrow unit$ rotating the elements of an array by shifting each element by one position to the right except for the last element, which becomes the new first element. For instance, rotation changes the state $[1, 2, 3]$ of an array to $[3, 1, 2]$.

10.5 Sorting

A naive sorting algorithm for arrays is **selection sort**: To sort a segment $i < j$, one swaps the least element of the segment into the first position of the segment and then continues sorting the smaller segment $i + 1 \leq j$.

```
let ssort a : unit =
  let r = Array.length a - 1 in
  let rec min k j : int =
    if k >= r then j
    else if a.(k+1) < a.(j) then min (k+1) (k+1)
    else min (k+1) j
  in let rec loop i : unit =
    if i >= r then ()
    else (swap a (min i i) i; loop (i+1))
  in loop 0
```

Function *ssort* satisfies the following invariants:

1. The state of the array agrees with the initial state up to repeated swapping.
2. The segment $0 < l$ is sorted.
3. Every element of the segment $0 < l$ is less or equal than every element of the segment $l \leq r$.
4. The element at j is the least element of the segment $i \leq k$.

The running time of selection sort is quadratic in the length of the array. Note that the two worker functions *min* and *loop* use only tail recursion.

A prominent sorting algorithm for arrays is **quick sort**. The top level of quick sort is reminiscent of merge sort: Quick sort partitions an array into two segments using swapping, where all elements of the left segment are smaller than all elements of the right segment. It then sorts the two segments recursively.

```
let qsort a =
  let partition l r = ... in
  let rec qsort' l r =
    if l >= r then ()
    else let m = partition l r in
          qsort' l (m-1); qsort' (m+1) r
  in qsort' 0 (Array.length a - 1)
```

Given $l < r$, *partition* yields m such that $l < m < r$ and

1. $a.(i) < a.(m)$ for all i such that $l \leq i \leq m - 1$
2. $a.(m) \leq a.(i)$ for all i such that $m + 1 \leq i \leq r$

10 Arrays

It is easy to verify that *qsort* terminates and sorts the given array if *partition* has the properties specified. Moreover, if *partition* splits a segment in two segments of the same size (plus/minus 1) and runs in linear time in the length of the given segment, the running time of *qsort* is $O(n \log n)$ (following the argument for merge sort).

We give a simple partitioning algorithm that chooses the rightmost element of the segment as **pivot element** that will be swapped to the dividing position m once it has been determined:

```

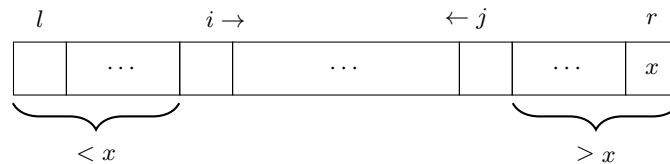
let partition l r =
  let x = a.(r) in
  let rec loop i j =
    if j < i
    then (swap a i r; i)
    else if a.(i) < x then loop (i+1) j
    else if a.(j) >= x then loop i (j-1)
    else (swap a i j; loop (i+1) (j-1))
  in loop l (r-1)

```

The function expects a segment $l < r$ in a and works with two pointers i and j initialized as l and $r - 1$ and satisfying the following invariants for the pivot element $x = a.(r)$:

1. $l \leq i \leq r$ and $l - 1 \leq j < r$
2. $a.(k) < x$ for all positions k left of i (meaning $l \leq k < i$)
3. $x \leq a.(k)$ for all positions k right of j (meaning $j < k \leq r$)

The situation may be described with the following diagram:



The pointers are moved further inside the segment as long as the invariants are preserved. If this leads to $j < i$, the dividing position is chosen as i and the elements at i and r are swapped. If $i \leq j$, we have $a.(j) < x$ and $x \leq a.(i)$. Now the elements at i and j are swapped and both pointers are moved inside by one position.

The partitioning algorithm is subtle in that it cannot be understood without the invariants. The algorithm moves the pointers i and j inside where the moves must respect the invariants and the goal is $j < i$. There is one situation where a swap preserves the invariants and enables a move of both i and j . As long as $i \leq j$, a move is always possible.

The given partitioning algorithm may yield an empty segment and a segment only by one position shorter than the given segment, which

10 Arrays

means that the worst case running time of *qsort* will be quadratic. Better partitioning algorithms can be obtained by first swapping a cleverly determined pivot element into the rightmost position of the given segment.

Exercise 10.5.1 Declare a function *sorted* : $\forall\alpha. \text{array}(\alpha) \rightarrow \mathbb{B}$ that checks whether an array is sorted. Do not convert to lists. Also write a function that checks whether an array is strictly sorted (i.e., sorted and no element occurring twice).

Exercise 10.5.2 Declare functions that for a nonempty array yield an index such that the value at the index is minimal (or maximal) for the array.

Exercise 10.5.3 Declare a variant of selection sort swapping the greatest element to the right.

Exercise 10.5.4 (Quick sort with middle element as pivot)

The performance of quick sort for sorted arrays can be dramatically improved by swapping the middle and the rightmost element before partitioning (thus making the middle element the pivot). Realize the improved quick sort algorithm in OCaml and check the performance claim for the array *Array.init* 100000 ($\lambda i.i$).

Exercise 10.5.5 (Quick sort with median-of-three pivots)

Some implementations of quick sort choose the pivot as the median of the leftmost, middle, and rightmost element of the segment (the median of three numbers is the number that is in the middle after sorting). This can be realized by swapping the median into the rightmost position before the actual partition starts.

a) A segment in an array has the *median property* if the rightmost element of the segment is the median of the leftmost, middle, and rightmost element of the segment. Declare a function *ensure_median* that given an array and a nonempty segment establishes the median property by possibly swapping two elements.

Hint: The necessary case analysis is amazingly tricky. Do a naive case analysis whether the leftmost or the rightmost or the middle element is the median using lazy boolean connectives.

b) Declare a quick sort function where partition establishes the median property before it partitions.

10.6 Equality for Arrays

OCaml's polymorphic equality test

$$= : \forall \alpha. \alpha \rightarrow \alpha \rightarrow \mathbb{B}$$

considers two arrays as equal if their states are equal. However, different arrays may have the same state. There is a second polymorphic equality test

$$== : \forall \alpha. \alpha \rightarrow \alpha \rightarrow \mathbb{B}$$

testing the equality of two array values without looking at their state:

```
let a = Array.make 2 1
let b = Array.make 2 1
let test2 = (a = b)           (* true *)
let test1 = (a == b)         (* false *)
let test3 = (a.(1) <- 2; a=b) (* false *)
let test4 = (a.(1) <- 1; a=b) (* true *)
```

We will refer to `=` as **structural equality** and to `==` as **physical equality**. The name physical equality refers to the fact that the operator `==` when applied to arrays tests whether they are the same physical object (i.e., are stored at the same address in the computer memory). For arrays and other mutable objects, physical equality implies structural equality. We will consider physical equality only for arrays.

10.7 Execution Order

As soon as mutable objects are involved, the order in which expressions and declarations are executed plays an important role. Consider for instance a get and a set operation on the field of an array. If the get operation is executed before the set operation, one gets the old value of the field, while if the set operation is executed before the get operation, one gets the new value of the field.

Like most programming languages, OCaml fixes the execution order only for certain expressions:

- Given $e_1; e_2$, subexpression e_1 is executed before e_2 .
- Given `LET e_1 IN e_2` , subexpression e_1 is executed before e_2 .
- Given `IF e_1 THEN e_2 ELSE e_3` , subexpression e_1 is executed before e_2 or e_3 , and only one of the two is executed.

In contrast, the execution order of operator applications $e_1 \circ e_2$, function applications $e_0 e_1 \dots e_n$, and tuple expressions (e_1, \dots, e_n) is not

10 Arrays

fixed and the interpreter can choose which order it realizes. This design decision gives compilers² more freedom in generating efficient machine code.

You can find out more about the execution order your interpreter realizes by executing the following declarations:

```
1 let test = invalid_arg "1" + invalid_arg "2"
2 let test = (invalid_arg "1", invalid_arg "2")
3 let test = (invalid_arg "1"; invalid_arg "2")
```

Depending on the execution order, the declarations raise different exceptions.

We remark that execution order as we discuss it here is only observable through side effects (i.e., raised exceptions and array updates).

One distinguishes between **pure** and **impure functional languages**. Exceptions, mutable objects, and physical equality make functional languages impure. Pure functional languages are very close to mathematical constructions, while impure functional languages integrate mutable objects, exceptions, and aspects concerning their implementation on computers.

10.8 Cells

OCaml provides single field arrays called **cells** using a type family $ref(t)$ and three operations:

$ref : \forall\alpha. \alpha \rightarrow ref(\alpha)$	allocation
$! : \forall\alpha. ref(\alpha) \rightarrow \alpha$	dereference
$:= : \forall\alpha. ref(\alpha) \rightarrow \alpha \rightarrow unit$	assignment

The allocation operation ref creates a new cell initialized with a value given; the dereference operation $!$ yields the value stored in a cell; and the assignment operation $:=$ updates a cell with a given value (i.e., the value in the cell is replaced with a given value). Cell values (i.e., the names of cells) are called **references**.

With cells we can declare a function

$$next : unit \rightarrow int$$

that counts how often it was called:

```
let c = ref 0
let next () = (c := !c + 1; !c)
```

²A compiler is a tool translating programs in a high-level language (e.g., OCaml) into machine language.

10 Arrays

Now the declarations

```
1 let test1 = next ()           1
2 let test2 = (next (), next ()) (3, 2)
3 let test3 = next () + next () + next () 15
```

bind their identifiers to the values given at the right. The second declaration assumes that the tuple expressions is executed from right to left.

We will see the function *next* as a stateful enumerator of the sequence 1, 2, 3, ... We can change the declaration of *next* such that the function encapsulates the cell storing its state:

```
let next = let c = ref 0 in
  fun () -> (c := !c + 1; !c)
```

This version is preferable for situations where only *next* is supposed to manipulate its state.

Exercise 10.8.1 Declare a function *newEnum* : *unit* → *unit* → *int* such that *newEnum* () yields a function enumerating 1, 2, 3, ... Make sure that each call *newEnum* () yields a fresh enumerator function starting from 1.

Exercise 10.8.2 Declare a function *unit* → *int* that enumerates the square numbers 0, 1, 4, 9, 16, ...

Exercise 10.8.3 Declare a function *unit* → *int* that enumerates the factorials 1, 1, 2, 6, 24, 120, ...

Exercise 10.8.4 Declare a function *unit* → *int* that enumerates the Fibonacci numbers 0, 1, 1, 2, 3, 5, ...

Exercise 10.8.5 Complete the declaration of *next* such that

```
let next = ...
let x = next ()
let y = next () + next () + next ()
```

binds *x* to 2 and *y* to 28.

Exercise 10.8.6 (Enumerator with reset function)

Declare a function

$$unit \rightarrow (unit \rightarrow int) \times (unit \rightarrow unit)$$

that on each application yields a pair consisting of an enumerator for 1, 2, 3... and a function resetting the enumerator to 1.

Exercise 10.8.7 Implement cells with arrays using the identifiers *cell*, *alloc*, *get*, and *set*.

10.9 Mutable Objects Mathematically

Mutable objects don't have a direct mathematical representation. What we can do is represent the state of a mutable object as an immutable value. The other aspect of an immutable object we can represent mathematically is a name fixing its identity. We may then describe the allocation of a mutable object as choosing a fresh name (a name that has not been used so far) and an initial state associated with this name. We thus represent the state of an object as a pair consisting of the name of the object and the current state of the object. As names we may use natural numbers.

11 Data Structures

In this chapter we consider array-based data structures known as stacks, queues, and heaps that are essential for machine-oriented programming. We explain how values of constructor types are stored in heaps using linked blocks, thus providing a first model for the space consumption of OCaml functions. We implement the data structures studied in this chapter using structure and signature declarations, two OCaml features we have not seen so far.

11.1 Structures and Signatures

OCaml offers a facility to bundle declarations into a **structure** and to restrict the visibility of the declarations with a **signature**. As a straightforward example we consider a signature and a structure for cells. We declare the signature as follows:

```
module type CELL = sig
  type 'a cell
  val make : 'a -> 'a cell
  val get  : 'a cell -> 'a
  val set  : 'a cell -> 'a -> unit
end
```

The keywords in the first line are explained by the fact that OCaml refers to structures also as **modules**, and to signatures also as **module types**. We also note that the signature *CELL* keeps the type constructor *cell* abstract, making it possible to give different implementations for cells.

An obvious implementation of the signature *CELL* would just mirror OCaml's native cells. For the purpose of demonstration, we choose an implementation of *CELL* with arrays of length 1. We realize the implementation with a declaration of a structure *Cell* realizing the signature *CELL*.

```
module Cell : CELL = struct
  type 'a cell = 'a array
  let make x = Array.make 1 x
  let get c = c.(0)
  let set c x = c.(0) <- x
end
```


11 Data Structures

There is type checking that ensures that the structure implements the fields the signature requires with the required types. Before checking compliance with the signature, the declarations of the structure are checked as if they would appear at the top level.

Once the structure *Cell* is declared, we can use its fields with the dot notation:

```
let enum = let c = Cell.make 0 in
  fun () -> let x = Cell.get c in Cell.set c (x+1); x
```

The declaration gives us an enumerator function for 0, 1, 2,

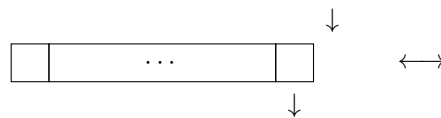
Exercise 11.1.1 Implement the signature *CELL* with OCaml's predefined reference cells.

Exercise 11.1.2 Arrays can be implemented with cells and lists. Declare a signature with a type family *array*(α) and the operations *make*, *get*, *set*, and *length*, and implement the signature with a structure not using arrays. We remark that expressing arrays with cells and lists comes at the expense that *get* and *set* cannot be realized as constant-time operations.

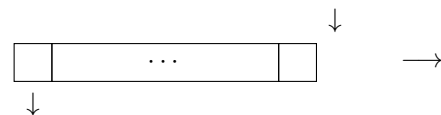
11.2 Stacks and Queues

An **agenda** is a mutable data structure holding a sequence of values called **entries**. The entries are entered into the agenda one after the other, and the sequence reflects the order in which the entries were entered. Depending on the kind of agenda, there are specific operations for inspecting and deleting entries.

Two prominent kinds of agendas are stacks and queues. A **stack** realizes a last-in-first-out (LIFO) policy, where only the most recent entry can be removed.



A **queue** realizes a first-in-first-out (FIFO) policy, where only the oldest entry can be removed.



11 Data Structures

We may think of a stack as oscillating at the right getting larger and smaller with insertion and removal, and of a queue as moving to the right with insertion and as shrinking at the left with removal.

For stacks we may have the signature

```
module type STACK = sig
  type 'a stack
  val make    : 'a -> 'a stack
  val push   : 'a stack -> 'a -> unit
  val pop    : 'a stack -> unit
  val top    : 'a stack -> 'a
  val height : 'a stack -> int
end
```

and the implementation

```
module Stack : STACK = struct
  type 'a stack = 'a list ref
  exception Empty
  let make x = ref [x]
  let push s x = s := x :: !s
  let pop s = match !s with
    | [] -> raise Empty
    | _::l -> s := l
  let top s = match !s with
    | [] -> raise Empty
    | x::_ -> x
  let height s = List.length (!s)
end
```

The operation *make* allocates a stack with a given single element. One says that *push* puts an additional element on the stack, *pop* removes the topmost element of the stack, *top* yields the topmost element of the stack, and *height* yields the height of the stack.

A stack may become empty after sufficiently many pop operations. Note that the structure *Stack* declares an exception *Empty* that is raised by *pop* if applied to an empty stack.

We require that *make* is given an initial element so that the element type of the stack is fixed. Alternatively one can have a *make* operation that throws away the given element.

For queues we may have the signature

11 Data Structures

```
module type QUEUE = sig
  type 'a queue
  val make    : 'a -> 'a queue
  val insert  : 'a queue -> 'a -> unit
  val remove  : 'a queue -> unit
  val first   : 'a queue -> 'a
  val length  : 'a queue -> int
end
```

and the implementation

```
module Queue : QUEUE = struct
  type 'a queue = 'a list ref
  exception Empty
  let make x = ref [x]
  let insert q x = q := !q @ [x]
  let remove q = match !q with
    | [] -> raise Empty
    | _::l -> q := l
  let first q = match !q with
    | [] -> raise Empty
    | x::_ -> x
  let length q = List.length (!q)
end
```

The operation *make* allocates a queue with a given single element. One says that *insert* appends a new element to the queue, *remove* removes the first element of the queue, *first* yields the first element of the queue, and *length* yields the length of the queue.

As it comes to efficiency, our naive implementations of stacks and queues can be improved. For queues, one would like that *insert* takes constant time rather than linear time in the length of the queue.

11.3 Array-Based Stacks

We will now realize a stack with an array. The elements of the stack are written into the array, and the height of stack is bounded by the length of the array. This approach has the advantage that a push operation does not require new memory but reuses space in the existing array. Such memory-aware techniques are important when data structures are realized close to the machine level.

We will realize a **bounded stack** with a structure and a single array. To make this possible, we fix the element type of the stack to machine integers. We declare the signature

11 Data Structures

```
module type BSTACK = sig
  val empty : unit -> bool
  val full   : unit -> bool
  val push   : int  -> unit
  val pop    : unit -> unit
  val top    : unit -> int
end
```

and realize the stack with the following structure:

```
module S : BSTACK = struct
  let size = 100
  let a = Array.make size 0
  let h = ref 0 (* height of stack *)
  exception Empty
  exception Full
  let empty () = !h = 0
  let full () = !h = size
  let push x = if full() then raise Full else (a.(!h) <- x; h := !h + 1)
  let pop () = if empty() then raise Empty else h := !h - 1
  let top () = if empty() then raise Empty else a.(!h - 1)
end
```

The height of the stack will oscillate in the array depending on the push and pop operations executed. If the size of the array is exhausted, the exception *Full* will be raised.

Note that with the structure approach stacks are realized as structures. Since structures are not values, structures cannot be passed as arguments to functions nor can they be the result of functions.

Note that all operations of bounded stacks are constant time.

Exercise 11.3.1 (Array-based stacks as values)

Array-based stacks can also be represented as values combining an array with a cell holding the current height of the stack. Modify the signature *Stack* from §11.2 such that *make* takes the maximal height of the stack object to be allocated as argument, and implement the signature with a structure such that all operations but *make* are constant time. Note that this approach provides a family of stack types where the element type can be freely chosen.

Exercise 11.3.2

Assume a structure $S : BSTACK$.

- Declare a function $toList : unit \rightarrow \mathcal{L}(int)$ that yields the state of the stack S as a list with the most recent entry appearing first.
- Declare a function $ofList : \mathcal{L}(int) \rightarrow unit$ that updates the state of the stack S to the sequence given by the list.

Use only operations declared in the signature *BSTACK*. OCaml's type checking will ensure this if S is declared with signature *BSTACK*.

11 Data Structures

As an interesting variant, extend the signature *BSTACK* with the operations *toList* and *ofList* so that they can be implemented within the structure with constant running time.

Exercise 11.3.3 Extend signature *BSTACK* and structure *S* with an operation *height* : *unit* → *int* that yields the current height of the stack.

11.4 Circular Queues

Similar to what we have seen for stacks, length-bounded queues can be realized within arrays. This time we use the signature

```
module type BQUEUE = sig
  val empty : unit -> bool
  val full   : unit -> bool
  val insert : int -> unit
  val remove : unit -> unit
  val first  : unit -> int
end
```

and implement it with the structure

```
module Q : BQUEUE = struct
  let size = 100
  let a = Array.make size 0
  let s = ref 0 (* position of first entry *)
  let l = ref 0 (* number of entries *)
  exception Empty
  exception Full
  let empty () = !l = 0
  let full () = !l = size
  let pos x = x mod size
  let insert x = if full() then raise Full
                 else (a.(pos(!s + !l)) <- x; l := !l + 1)
  let remove () = if empty() then raise Empty
                  else (s := pos (!s + 1); l := !l - 1)
  let first () = if empty() then raise Empty else a.(!s)
end
```

As with stacks, there is no recursion and all operations are constant time. This time we represent the queue with an array and two cells holding the position of the first entry and the number of entries in the queue. To fully exploit the capacity of the array, the sequence of entries may start at some position of the array and continue at position 0 of the array if necessary. This way the sequence of entries can move through the array as if the array was a ring. Technically, this is realized with the constant-time remainder operation.

11 Data Structures

The ring implementation of queues described above is known as a **circular buffer** or as a **ring buffer**.

Exercise 11.4.1 (Array-based queues as values)

Similar to what we have seen for stacks in Exercise 11.3.1, array-based queues can be represented as values combining an array with two cells holding the position of the first entry and the number of entries in the queue. Modify the signature *Queue* from §11.2 such that *make* takes the maximal length of the queue object to be allocated as argument, and implement the signature with a structure such that all operations but *make* are constant time.

Exercise 11.4.2 Assume a structure $Q : BQUEUE$.

- Declare a function $toList : unit \rightarrow \mathcal{L}(int)$ that yields the state of the queue Q as a list with the most recent entry appearing first.
- Declare a function $ofList : \mathcal{L}(int) \rightarrow unit$ that updates the state of the queue Q to the sequence given by the list.

Use only operations declared in the signature *BQUEUE*. OCaml's type checking will ensure this if Q is declared with signature *BQUEUE*.

As an interesting variant, extend the signature *BQUEUE* with the operations *toList* and *ofList* so that they can be implemented within the structure with constant running time.

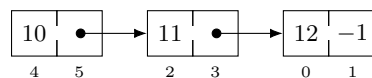
11.5 Block Representation of Lists in Heaps

Values of constructor types like lists, AB-trees, and expressions must be stored in computer memory. We demonstrate the underlying technique by showing how values of constructor types can be represented in an array-based data structure called a **heap**.

A heap is an array where mutable fixed-length sequences of machine integers called **blocks** are stored. For instance, the list

[10, 11, 12]

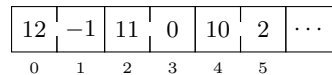
may be stored with three blocks connected through links:



Each of the 3 blocks has two fields, where the first field represents a number in the list and the second field holds a **link**, which is either a link to the next block or -1 representing the empty list. The numbers below the fields are the indices in the array where the fields are allocated.

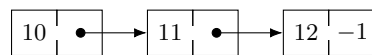
11 Data Structures

The fields of a block are allocated consecutively in the heap array. The **address** of a block is the index of its first field in the heap array. Links always point to blocks and are represented through the addresses of the blocks. Thus the above linked block representation of the list [10, 11, 12] is represented with the following segment in the heap:

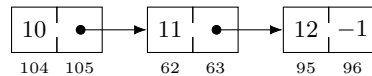


Note that the representation of the empty list as -1 is safe since the addresses of blocks are nonnegative numbers (since they are indices of the heap array).

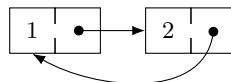
Given a linked block representation of a list



we can freely choose where the blocks are allocated in the heap. There is no particular order to observe, except that the fields of a block must be allocated consecutively. We might for instance choose the following addresses:



It is also perfectly possible to represent **circular lists** like



We emphasize that mathematical lists cannot be circular. It is a consequence of the physical block representation that circular lists come into existence. Lists as they appear with the block representation are commonly called **linked lists**.

Exercise 11.5.1 How many blocks and how many fields are required to store the list [1, 2, 3, 4] in a heap using the linked block representation?

11.6 Realization of a Heap

We will work with a heap implementing the signature

```

module type HEAP = sig
  type address = int
  type index = int
  val alloc    : int -> address
  val get     : address -> index -> int

```

11 Data Structures

```
val set      : address -> index -> int -> unit
val release : address -> unit
end
```

The function *alloc* allocates a new block of the given length in the heap and returns the address of the block. Blocks must have at least length 1 (that is, at least one field). The function *get* yields the value in the field of a block, where the field is given by its index in the block, and the block is given by its address in the heap. The function *set* replaces the value in the field of a block. Thus we may think of blocks as mutable objects. The function *release* is given the address of an allocated block and deallocates this block and all blocks allocated after this block. We implement the heap with the following structure:

```
module H : HEAP = struct
  let maxSize = 1000
  let h = Array.make maxSize (-1)
  let s = ref 0 (* current size of heap *)
  exception Address
  exception Full
  type address = int
  type index = int
  let alloc n = if n < 1 then raise Address
                else if !s + n > maxSize then raise Full
                else let a = !s in s := !s + n; a
  let check a = if a < 0 || a >= !s then raise Address else a
  let get a i = h.(check(a+i))
  let set a i x = h.(check(a+i)) <-x
  let release a = s := check a
end
```

Note that we use the helper function *check* to ensure that only allocated addresses are used by *set*, *get*, and *release*. The exception *address* signals an address error that has occurred while allocating or accessing a block.

Given the heap *H*, we declare a high-level allocation function

$$\text{alloc}' : \mathcal{L}(\text{int}) \rightarrow \text{address}$$

allocating a block for a list of integers:

```
let alloc' l =
  let a = H.alloc (List.length l) in
  let rec loop l i = match l with
    | [] -> a
    | x::l -> H.set a i x; loop l (i+1)
  in loop l 0
```


11 Data Structures

Note that the low level allocation function ensures that an allocation attempt for the empty list fails with an address exception.

It is now straightforward to declare a function

$$putlist : \mathcal{L}(int) \rightarrow address$$

allocating a linked block representation of a given list in the heap:

```
let rec putlist l = match l with
| [] -> -1
| x::l -> alloc' [x; putlist l]
```

Next we declare a function

$$getlist : address \rightarrow \mathcal{L}(int)$$

that given the address of the first block of a list stored in the heap recovers the list as a value:

```
let rec getlist a =
  if a = -1 then []
  else H.get a 0 :: getlist (H.get a 1)
```

Note that *putlist* and *getlist* also work for the empty list. We have

$$getlist (putlist l) = l$$

for every list l whose block representation fits into the heap.

Exercise 11.6.1 Given an expression allocating the list $[1, 2, 3]$ in the heap H .

Exercise 11.6.2 Given an expression allocating a circular list $[1, 2, \dots]$ as shown in §11.5 in the heap H .

Exercise 11.6.3 Declare a function that given a number $n > 0$ allocates a cyclic list $[1, 2, \dots, n, \dots]$ in the heap H .

Exercise 11.6.4 Declare a function that checks whether a linked list in the heap H is circular.

11.7 Block Representation of AB-Trees

Block representation works for constructor types in general. The idea is that a constructor application translates into a block holding the arguments of the constructor application. If there are no arguments, we don't need a block and use a negative number to identify the nullary

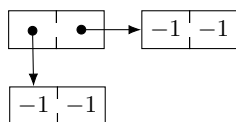
11 Data Structures

constructor. If there are arguments, we use a block having a field for every argument. If there are several constructors taking arguments, the blocks will also have a **tag field** holding a number identifying the constructor. This will work for constructor types where the arguments of the constructors are either numbers, booleans, or values of constructor types.

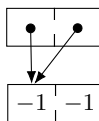
As example we consider AB- and ABC-trees (§5.2 and §5.5). For AB-trees we don't need a tag field since there is only one constructor taking arguments. Thus the block representation of AB-trees is as follows:

1. A tree A is represented as -1 .
2. A tree $B(t_1, t_2)$ is represented as a block with 2 fields carrying the addresses of t_1 and t_2 .

For instance, the block representation of the tree $B(B(A, A), B(A, A))$ looks as follows:



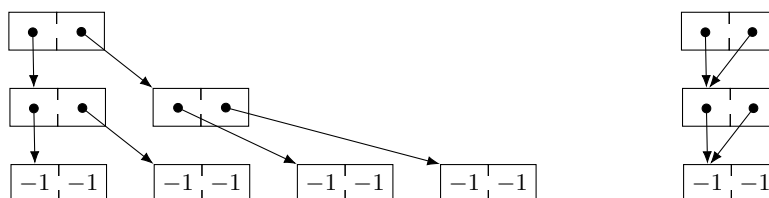
At this point we may ask why in the tree $B(A, A)$ is represented with two boxes rather than just one:



In fact, if the boxes represent immutable values, there is no need for this double allocation. One says that the more compact representation realizes **structure sharing**. For the maximal tree of depth 3

$$B(B(B(A, A), B(A, A)) , B(B(A, A), B(A, A)))$$

the naive representation and the representation with maximal structure sharing look as follows:



We now see that the block representation with no structure sharing is exponentially larger than the block representation with maximal structure sharing.

Exercise 11.7.1 Consider the ABC-tree

$$B(C(A, B(A, A)), B(A, A))$$

with a block representation employing the tags 0 and 1 for the constructors B and C .

- a) Draw a block representation without structure sharing.
- b) Draw a block representation with structure sharing.
- c) Give an expression allocating the tree with structure sharing.

Exercise 11.7.2 Declare functions such that $getTree(putTree\ t) = t$ holds for all AB-trees that fit into the heap.

Exercise 11.7.3 ABC-trees can be stored in a heap by a representing trees of the forms $B(t_1, t_2)$ and $C(t_1, t_2)$ with blocks with three fields, where two fields hold the subtrees and the additional tag field says whether B or C is used. Declare functions such that $getTree(putTree\ t) = t$ holds for all ABC-trees t that fit into the heap.

11.8 Structure Sharing and Physical Equality

An OCaml interpreter stores the values of constructor types it computes with in a heap using a block representation, and physical equality (§10.6) of values of constructor types is equality of heap addresses. If we use the function (§8.6)

```
let rec mtree n =
  if n < 1 then A
  else let t = mtree (n-1) in B(t,t)
```

to obtain the maximal AB-tree of depth n , we are guaranteed to obtain a heap representation with maximal structure sharing whose size $O(n)$. On the other hand, the function

```
let rec mtree' n =
  if n < 1 then A else B(mtree (n-1), mtree (n-1))
```

will construct a heap representation without structure sharing whose size is $O(2^n)$. Thus $mtree'$ is naive in that it takes exponential space and exponential time where linear space and linear time would suffice.

11 Data Structures

Exercise 11.8.1 Consider the block representation of AB-trees in the heap H .

- a) Declare a function that writes the maximal AB-tree of depth n into the heap such that running time and space consumption in the heap are linear in the depth.
- b) Declare a function that reads AB-trees from a heap preserving structure sharing between subtrees that are siblings (as in $B(t, t)$). On the heap representations of the trees obtained with the function from (a) your function should only use linear time in the depth of the tree.
- c) Using physical equality, you can find out in constant time whether two AB-trees in OCaml have the same heap representation. Declare a function writing AB-trees into the heap such that structure sharing between sibling subtrees in OCaml is preserved. Test your function with *mtree'*.

12 Appendix: Example Exams

The 2021 iteration of the course came with 3 written exams:

- A midterm exam of 90 minutes on December 11, 2021.
- An endterm exam of 120 minutes on February 26, 2022.
- A late endterm exam of 120 minutes on March 26, 2022.

To pass the course, students had to score 50% in one of the two endterm exams. To participate in the endterm exams, students had to score 50% in the midterm exam. In addition, students had to score enough points in the weekly tests and the Mini-OCaml project.

Below we give the problems of the exams of the 2021 iteration so that future instructors and students can get an idea what we expect students learn in the course.

The course presents more topics than can be covered in the exams. Students don't know which topics are chosen for the exams. Important topics missing in the exams shown here are binary search, parsing of infix operators, and the quick sort algorithm for arrays.

Midterm

First 8P

Declare a function *foo* that for a non-negative integer x yields the largest number n such that $n^3 \leq x$. For instance, $foo(26) = 2$ and $foo(27) = 3$.

Fibonacci numbers with iteration 8P

Recall the sequence of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, Using iteration, declare a function *fib* that for $n \geq 0$ yields the n th Fibonacci number. For instance, $fib(0) = 0$, $fib(1) = 1$, and $fib(4) = 3$. Hint: $fib(n + 2) = fib(n) + fib(n + 1)$.

List reversal 8P

Declare a polymorphic function *rev* reversing lists. For instance, $rev[1, 2, 3] = [3, 2, 1]$. Use only tail recursion. Do not use list concatenation @.

List construction 8P

Declare a polymorphic function *init* : $\forall \alpha. \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \mathcal{L}(\alpha)$ such that $init\ n\ f = [f(0), \dots, f(n - 1)]$. For instance, $init\ f\ 0 = []$ and $init\ f\ 2 = [f(0), f(1)]$. Use only tail recursion.

Decimal representation 8P

Declare a function *dec* that yields the decimal representation of a number as a list. For instance, $dec(456) = [4, 5, 6]$ and $dec(0) = []$.

Prime factorization 8P

Declare a function *pfac* that yields the prime factorization of a number $x \geq 2$ as a list. For instance, $pfac(60) = [2, 2, 3, 5]$.

Insertion sort 8P

Declare a polymorphic function *sort* sorting lists following the insertion sort algorithm. For instance, $sort [5, 3, 2, 3, 2] = [2, 2, 3, 3, 5]$.

List prefixes 8P

Declare a polymorphic function *pre* that given a list yields a list containing all prefixes of the list. For instance, $pre [1, 2, 3] = [[], [1], [1, 2], [1, 2, 3]]$.

AB-tree construction 8P

Declare a function *ctree* that for a number $n \geq 0$ yields an AB-tree of depth n and size $2n + 1$. For instance, $ctree(1) = B(A, A)$.

AB-tree infix linearization 8P

Declare a function *lin* linearizing AB trees such that *B* is treated as a left-associative infix operator. For instance, $lin(B(B(A, A), B(A, A))) = \text{"ABAB(ABA)"}$.

Typing and evaluation rules 10P

Complete the typing and evaluation rules for lambda expressions and function applications.

$$\frac{}{E \vdash \lambda x.e :} \qquad \frac{}{V \vdash \lambda x.e \triangleright}$$

$$\frac{}{E \vdash e_1 e_2 :} \qquad \frac{}{V \vdash e_1 e_2 \triangleright}$$

Early Endterm

Euclid's algorithm 6P

Declare a function *gcd* : $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ computing the GCD of two numbers using the remainder operation $x \% y$. Use only tail recursion.

List construction 6P

Declare a function *make* : $\forall \alpha. \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \mathcal{L}(\alpha)$ such that $make\ n\ f = [f(0), \dots, f(n - 1)]$. Use only tail recursion.

12 Appendix: Example Exams

Fibonacci numbers 6P

Declare a constant-time function $enum : unit \rightarrow int$ enumerating the sequence of Fibonacci numbers:

$$\begin{aligned} fib\ 0 &:= 0 \\ fib\ 1 &:= 1 \\ fib\ (n + 1) &:= fib\ n + fib\ (n + 1) \end{aligned}$$

Prime numbers 10P

Recall the sequence of prime numbers: 2, 3, 5, 7, 11, ...

- Declare a function $prime : int \rightarrow bool$ testing primality.
- Declare a function $enum : unit \rightarrow int$ enumerating the sequence of prime numbers.

You may use the function *first*.

Prefixes, suffixes, segments 12P

Given a list $l = l_1 @ l_2 @ l_3$, we call l_1 a prefix, l_2 a segment, and l_3 a suffix of l . The lists l_1 , l_2 , and l_3 may be empty.

- Declare a function $\forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\mathcal{L}(\alpha))$ that yields a list containing all suffixes of a list.
- Declare a function $\forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\mathcal{L}(\alpha))$ that yields a list containing all prefixes of a list.
- Declare a function $\forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\mathcal{L}(\alpha))$ that yields a list containing all segments of a list.

You may use the functions *List.cons*, *List.map*, and *List.concat*.

Merge sort with duplicate removal 12P

The merge sort algorithm for lists can be modified such that it sorts a list and also removes duplicates. For instance, for the list [3, 2, 3, 2, 1] we want the list [1, 2, 3].

- Declare a function $merge : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ merging two duplicate-free sorted lists into one duplicate-free sorted list. For instance, we want $merge\ [1, 2]\ [2, 3] = [1, 2, 3]$. The running time of merge should be at most linear in the sum of the lengths of the two input lists.
- Given a function $split : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha) \times \mathcal{L}(\alpha)$ splitting lists and the merge function from (a), declare a function $sort : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ that sorts and removes duplicates. The running time of *sort* should be $O(n \log n)$. Assume that *split* splits a list into two lists of equal size (plus/minus 1) and has running time linear in the size of the input list.

12 Appendix: Example Exams

Array reversal 10P

Declare a function $rev : \forall \alpha. \text{array}(\alpha) \rightarrow \text{unit}$ reversing arrays. For instance, an array with state $[1, 2, 3]$ should be updated to the state $[3, 2, 1]$.

Let expressions 14P

We consider expressions consisting of variables and let expressions:

```
type var = string
type exp = Var of var | Let of var * exp * exp
```

Note that the expressions considered can be type checked and evaluated in environments without making assumptions about types and values.

a) Realize environments with three declarations as follows:

```
type 'a env
val lookup : 'a env -> var -> 'a option
val update : 'a env -> var -> 'a -> 'a env
```

b) Declare a type checker

```
val check : 'a env -> exp -> 'a option
```

Do not raise exceptions.

c) Can the type checker be used as an evaluator? Answer yes or no.

Postfix linearization 12P

We consider AB-trees and their postfix linearization following the grammar $tree ::= "A" \mid tree \ tree \ "B"$. We assume the type declarations

```
type tree = A | B of tree * tree
type token = AT | BT
```

a) Declare a function $lin : tree \rightarrow \mathcal{L}(token)$ computing the postfix linearization of AB-trees.

b) Declare a tail-recursive function $par : \mathcal{L}(token) \rightarrow \mathcal{L}(tree) \rightarrow \mathcal{O}(tree)$ parsing the postfix linearization of AB-trees. We want $par (lin t) [] = Some\ t$ for all AB-trees t and $par\ l\ [] = None$ if l is not a postfix linearization of an AB-tree.

Prime factorization 10P

Declare a function $pfac : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathcal{L}(\mathbb{N})$ such that $pfac\ x\ 2$ computes the prime factorization of $x \geq 2$. Do not use helper functions. Give the invariant $pfac$ must satisfy so that $pfac\ x\ k$ computes the prime factorization of x .

Decimal representation 12P

- a) Declare a function $dec : \mathbb{N} \rightarrow \mathcal{L}(\mathbb{N})$ computing the decimal representation of x . For instance, $dec\ 453 = [4, 5, 3]$. Use only tail recursion.
- b) Declare a function $num : array(int) \rightarrow int$ converting a decimal representation stored in an array into the number represented. For instance, we want $undec\ [[5; 6; 2]] = 562$. Use only tail recursion.

Heap representation with structure sharing 12P

We consider AB-trees

```
type tree = A | B of tree * tree
```

and assume a heap with an operation

```
alloc : int list -> address
```

allocating blocks and an operation

```
get : address -> index -> int
```

reading the fields of a block. As usual, *address* and *index* are names for the type *int*, and the fields of a block are numbered starting from 0.

- a) Declare a function $putMaxTree : int \rightarrow address$ that for $n \geq 0$ stores the maximal AB-tree of depth n in the heap using blocks of length 2 and taking running time $O(n)$.
- b) Declare a function $getTree : address \rightarrow tree$ that reads AB-trees from the heap preserving structure sharing between siblings. A call $getTree(putMaxTree\ n)$ should take running time $O(n)$.

Late Endterm**Strictly ascending lists** 7P

Declare a tail-recursive function $test : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathbb{B}$ testing whether a list is strictly ascending. A list $[x_1, x_2, \dots, x_n]$ is strictly ascending if $x_1 < x_2 < \dots < x_n$.

Enumerators 8P

Declare a function $enum : \forall \alpha. (int \rightarrow \alpha) \rightarrow unit \rightarrow \alpha$ such that each call $enum\ f$ yields a new enumerator for the sequence $f(0), f(1), f(2), \dots$

Linear search 8P

Declare a function $find : \forall \alpha. \alpha \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{O}(\mathbb{N})$ that for x and l returns the first position x occurs in l . For instance, $find\ 3\ [3, 2, 3] = \text{Some } 0$ and $find\ 1\ [3, 2, 3] = \text{None}$. Use only tail recursion.

Sorting and removing duplicates 12P

Declare a function $sortRem : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ that sorts a list and removes duplicates. For instance, $sortRem [3, 2, 3, 4, 2] = [2, 3, 4]$. Follow the insertion sort algorithm.

Sublists 13P

The sublists of a list are obtained by deleting $n \geq 0$ positions. For instance, the sublists of $[1, 2]$ are $[], [1], [2], [1, 2]$.

- Declare a function $pow : \forall \alpha. \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\mathcal{L}(\alpha))$ that yields a list containing all sublists of a list.
- Declare a function $pow' : \forall \alpha. \mathbb{N} \rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\mathcal{L}(\alpha))$ such that $pow' n l$ computes a list containing all sublists of l whose length is n .

Prefix linearization 13P

We consider AB-trees and their prefix linearization following the grammar $tree ::= "A" \mid "B" tree tree$. We assume the type declarations

```
type tree = A | B of tree * tree
type token = AT | BT
```

- Declare a function $lin : tree \rightarrow \mathcal{L}(token)$ computing the prefix linearization of AB-trees. For instance, $lin(B(A, B(A, A))) = [BT, AT, BT, AT, AT]$.
- Declare a function $par : \mathcal{L}(token) \rightarrow tree$ parsing the prefix linearization of AB-trees. We want $par(lin t) = t$ for all AB-trees t . If l is not a prefix linearization of an AB-tree, par should raise an exception with *failwith* "par". For instance, par should raise an exception on $[AT, AT]$.

Minimal and maximal AB-trees 16P

We consider AB-trees

```
type tree = A | B of tree * tree
```

- Declare a function $min : \mathbb{N} \rightarrow tree$ that for n yields an AB-tree of depth n that has minimal size. For depth 2 the trees $B(B(A, A), A)$ and $B(A, B(A, A))$ have minimal size.
- Declare a function $max : \mathbb{N} \rightarrow tree$ that for n yields an AB-tree of depth n that has maximal size. Make sure the running time of your function is linear in the depth of the tree.
- Declare a function $check : tree \rightarrow \mathcal{O}(\mathbb{N})$ such that $check(t) = \text{Some}(n)$ if t is a minimal tree of depth n , and $check(t) = \text{None}$ if t is not minimal.

12 Appendix: Example Exams

Expressions 15P

We consider expressions $e ::= c \mid x \mid \lambda x.e \mid e_1 e_2$ implemented with the declarations

```
type var = string
type exp = Con of int | Var of var | Lam of var * exp | App of exp * exp
```

a) Declare a function

```
closed : var list -> exp -> bool
```

that checks whether an expression is closed in a list of variable. For instance, $\lambda x.fxy$ is closed in $[f, y]$.

b) Declare a function

```
eval : value env -> exp -> value
```

evaluating an expression in an environment. We assume that values and environments are implemented as follows:

```
type 'a env
val lookup : 'a env -> var -> 'a option
val update : 'a env -> var -> 'a -> 'a env
type value = Int of int | Clo of var * exp * value env
```

If an expression cannot be evaluated, *eval* should raise an exception with *failwith* "eval".

Array rotation 12P

Declare a function $rotate : \forall \alpha. array(\alpha) \rightarrow unit$ rotating the elements of an array by shifting each element by one position to the right except for the last element, which becomes the new first element. For instance, rotation changes the state $[1, 2, 3]$ of an array to $[3, 1, 2]$. Use only tail recursion.

Correctness 16P

Consider two iteration functions $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \mathbb{N} \rightarrow \alpha \rightarrow \alpha$ defined as follows:

$$\begin{aligned} it\ f\ 0\ x &:= x & it'\ f\ 0\ x &:= x \\ it\ f\ (n+1)\ x &:= it\ f\ n\ (f\ x) & it'\ f\ (n+1)\ x &:= f\ (it'\ f\ n\ x) \end{aligned}$$

Prove that the two iteration functions agree, that is, $it\ f\ n\ x = it'\ f\ n\ x$. Hint: You will need a lemma for the proof. State and prove the lemma.