Principles of Programming Languages

Achim Blumensath

Spring Semester 2021

Contents

1 Introduction

The first high-level programming languages where designed at the end of the 1950s. Since then a large number has been created and every year even more are released. Each of them supports different programming paradigms and styles, and thus provides the programmer with different ways to approach algorithmic problems and to express herself. Given this wealth of approaches, programmers today face the problem of which languages to learn, which of them to choose for a given project, and how to use the various features a given language provides.

Unfortunately, there are no easy answers to such questions. In addition to personal preferences, the choice of a language and coding style also very much depends on the project in question. It makes quite a difference whether you are writing an office program, an application for mobile phones, a first-person shooter, or a network server. A competent programmer therefore has to be familiar with many paradigms, coding styles, and ways to solve problems, so that she can always choose the most appropriate tool for the task at hand.

This course is meant to help you with this choice. Instead of discussing individual languages, we will study their features in the abstract. We will define a minimal language (we call it the *toy language*) that exhibits the features under consideration in as pure a way as possible, so we can study them and their ramifications in isolation. Our focus will not be on what these features are and how they work, but on how and when to use them for programming and on the trade-offs involved.

To prevent our discussions from becoming too abstract and vague, we will also present implementations of the kernel languages. This helps to better understand each feature and allows you to experiment with them. Note that these implementations are chosen for clarity and pedagogical merits, not to teach compiler technology. Hence, they are fairly inefficient and naïve. I have chosen Haskell as the programming language for these implementations, mainly as it allows for very concise code without unnecessary clutter. (Although sometimes Haskell's insistence on being pure makes the code more involved than it needs to be.) As the goal is clarity of presentation and in order to help readers unfamiliar with Haskell, I have decided against the use of the more advanced features of the language. Everything is written in the most elementary way possible.

Ultimately, this course is meant to help you become better programmers. Of course, like riding a bike, you cannot learn programming by just listening to lectures. To improve, you need a lot of practice. So ideally every part of this course should be accompanied by coding exercises where you can get familiar with the features under discussion. As this course does not have exercise classes, we have to do without. In order to get the most out of the course, I therefore recommend that you get into the habit of

- paying attention to which language features you use in the programs you write, and
- trying to think of alternative ways you could have written the code in.

For instance, ask questions like: 'How would this piece of code look like if I had written it in a functional style? Would that be better or worse than the code I have now?'

This raises the question of what constitutes 'better' code. Does 'better' mean higher performance, higher reliability, better maintainability, the code being more readable, less security problems, lower compile times, better portability, reduced development time,…? Many of these goals contradict each other, so one has to prioritise. Consequently, there is no one best way to program. Every approach comes with its own trade-offs, which means that in each case the programmer has to choose the right approach to solve the problem at hand. For instance, for many video games performance is of utmost importance whereas security concerns are secondary. The opposite is the case for server software.

For these reasons we will refrain from giving strict rules of how to write programs. At most we might present some rules of thumb that the reader is encouraged to ignore in situations they do not work out. Instead we will try to present each approach impartially and discuss its advantages and disadvantages, so the reader can decide for herself when and how to apply it. This is in marked contrast to the usual situation in software engineering, which is all to often permeated by ideology and dogma with very little empirical data to back it up. ('Object-oriented programming is superior!' 'You have to write unit tests!' 'You always should use pair programming!'…) Frequently rules like these arise when someone notices a problem with a particular way of doing things and then gives the workaround they develop the status of an absolute principle, while ignoring the fact that their solution comes with its own set of problems or that there might be cases where it performs badly. For instance, unit tests were developed to cope with the error proneness of writing substantial applications in dynamically typed languages. While they can drastically reduce the number of bugs in such projects, people often forget that their maintenance places additional burdens on the developers and that they offer little advantages when working with statically typed languages.

General principles

Before getting started, let me make a few general remarks related to code quality and language design.

Readability

What makes code easy or hard to understand? Let us mention several aspects that influence readability.

How self-contained is a piece of code? A piece of code is the harder to understand the more other parts of the program you need to know in order to understand it.

How explicit is the code? There is the question of how many aspects of the code are explicitly written down and how many are left implicit for the reader to figure out by herself. For instance, type annotations make the types of variables explicit, while relying on type inference means that the reader has to work them out each time she looks at the code. For another example, a deeply

nested expression can be harder to understand than a sequence of simpler expressions where each intermediate value is stored in a named variable. Usually, the more is left implicit, the harder it is to read the code. But note that explicitness can be overdone: if there are too many annotations the reader has to read and understand them, which can be overwhelming (see also the next point).

How verbose is the code? The more verbose the code is, i.e., the higher the ratio is between the size of the code and the information contained in it, the more code the programmer has to read. In a concise language a piece of code might fit on a single screen and be taken in all at once by the reader, while in a more verbose version the programmer might have to scroll through several pages and laboriously extract the meaning. For instance,

```
1 + (2 * 3)
```

is much faster to read then

```
Take the number two, multiply it by the number three, and then
  add the number one.
```

As usual, conciseness can be overdone. If you have one of the famous single-line Haskell functions involving a deeply nested expression containing several higher-order functions and a few user-defined binary operators, you might need a few minutes to recall how each of the operators was defined and then to figure out what the expression is doing.

How large is the code? The more code there is, the more code you have to understand.

How complex is the code? Obviously, the more complex a piece of code the harder it is to understand. For instance, if you have several versions of some data structure, it might be better to write separate implementations instead of a single one. An implementation for a special case is usually simpler than one for the general case.

Note that there are two aspects to code complexity: the inherent complexity of a piece of code in itself, and the complexity of its interactions with other parts of the program. For instance, breaking a large function into several smaller ones, reduces their inherent complexity, but increases the interaction complexity. The same goes for breaking a program into several modules.

Principle of Local Reasoning. Let us isolate a general principle which is quite useful to understand issues of readability. Ideally one would like each piece of code to be intelligible by just looking at it in isolation without having to read other parts of the program. We call this *local reasoning*. If, on the other hand, one has to look at the whole program to see what is going on, we speak of *global reasoning*.

Let us give a few examples where the Principle of Local Reasoning comes into play.

(1) Communication via (mutable) global variables requires global reasoning since you have to look at every part of the program that modifies the variable in question. A pure functional style where communication is done solely via function parameters is local instead.

(2) Using abstraction violates the principle since one cannot understand a piece of code without knowing the definitions of the functions and variables appearing in it. One can mitigate this effect

a little by using meaningful names for variables and functions, that make it easier to remember their definitions.

Control

In language design there is a trade-off between the degree of automation and the amount of control offered to the programmer. Certain routine tasks, like memory management for instance, can be automated by a language or a library.

This can save the programmer a lot of tedium, making the language more convenient to use and reducing development time. Furthermore, it prevents bugs: the more the programmer has to do herself, the more opportunities she has to make mistakes.

But automation also means giving up control, which can make it very hard, or even impossible to do things, in a non-standard way. For instance, for performance reasons it might be important to specify the precise point in a program where certain memory is freed. In languages with garbage collection, this is usually not possible.

In addition, automation can lead to surprising behaviour of the program. For instance, if a language supports implicit type conversions, type errors might go unnoticed. Another example is automatic memory management in languages with lazy evaluation where it can be very hard for the programmer to figure out how much memory is used and when it is deallocated again.

Code reuse

Reusing code is usually considered a good thing. Some of the obvious advantages are the following ones.

- The programmer has to write less code.
- Readers have to understand less code.
- The code is already tested and therefore contains fewer bugs.
- Having less code means there are fewer places that can be incorrect.

But note that there is also a downside: reusing code usually requires to make the old code more general in order to handle more cases. This can make the code more complicated and also less efficient.

As a rule of thumb I recommend not to be too general at first. Instead, implement a specialised solution for the problem at hand. Once you have at least, say, three similar pieces of code, you can try to (fully or partially) combine them into a general one, but only if that does not increase code complexity.

2 Expressions and Functions

2.1 Arithmetic expressions

In one form or other *expressions* are present in nearly every programming language. In the abstract, an expression can be defined as a program construct that computes a value. A prototypical example are arithmetical expressions in mathematics. In so-called *functional* languages, expressions form the central construct around which the whole language is build. In this chapter we will introduce a functional toy language, starting with arithmetical expressions.

$$\langle expr \rangle \ ::= \ \langle num \rangle \ \mid \texttt{(} \ \langle expr \rangle \ \texttt{)} \mid \ \langle expr \rangle \ \texttt{+} \ \langle expr \rangle \ \mid \ \langle expr \rangle \ \texttt{*} \ \langle expr \rangle$$

The evaluation strategy for such expressions is obvious: we recursively evaluate all subexpressions and then combine their results using the operation at the current position. For instance,

```
1+2*3
=> 1+6
=> 7
```

Let us see how to implement this in Haskell. First, we need a data type for syntax trees. We need to distinguish three different cases: numbers, sums, and products. Hence, we can write

```
1  data Expr = ENum Integer | EPlus Expr Expr | ETimes Expr Expr
```

Evaluation of such a syntax tree is done by a simple recursion.

```
2  eval :: Expr -> Integer
3  eval (ENum n)       = n
4  eval (EPlus  e1 e2) = eval e1 + eval e2
5  eval (ETimes e1 e2) = eval e1 * eval e2
```

If we want to add further operations, we can either add them as a new category to the syntax tree, or we can define them as *syntactic sugar,* i.e., we express it in terms of the operations the core language already provides. For instance, we can define subtraction by

$$expr_1 \texttt{ - } expr_2 \quad \Longrightarrow \quad expr_1 \texttt{ + (-1) * } expr_2 \,,$$

that is, after parsing, but before any further analysis, we replace every occurrence of subtraction by its definition. Of course, this only works if we can express the new feature in terms of the old ones.

Expression.	A part of the program that denotes a value.

Abstraction. Naming a part of the program.

2.2 Local definitions

One central mechanism to improve the readability and maintainability of code is the ability to *name* a given program construct like an expression, a type, etc. and to refer to that construct using its new name. This process is called *abstraction*. It is vital in breaking a program into smaller, easier to understand parts. We can use it to hide unimportant details and thereby decrease complexity of our code. In addition abstraction also facilitates code reuse. Of course, abstraction can also be misused: the more names one introduces the more definitions one has to remember. A good rule of thumb here is to only move code into its own function if you can summarise in one sentence what the code is doing. Using long and descriptive names also helps.

At the moment our toy language has only one construct: expressions. To name an expression, we introduce *local definitions*.

$$\langle expr \rangle \ ::= \ldots \mid \langle id \rangle \mid \texttt{let}\ \langle id \rangle\ \texttt{=}\ \langle expr \rangle\ \texttt{;}\ \langle expr \rangle$$

A remark on terminology: in our current setting without side-effects, we refer to these names as *identifiers*, instead of using the more common term 'variables'. We reserve the latter for *mutable identifiers*, which we will introduce in Chapter 4.

Examples:

```
let x = 1;
let y = 2;
x + 2*y
=> 5

let x = (let y = 2; 2*y);
x + 3
=> 7
```

```
let pi = 3;  // the integer version ;-)
2*pi*5
=> 30

(let x = 2; x * x) - (let x = 1; x+4)
=> -1
```

Apart from making code more readable and easier to write, local definitions can also be used to improve performance. If a complicated expression is used in several places, we can use a let-binding to evaluate the expression only once and then refer to its value via the corresponding identifier. For instance, if we want to rotate a vector, we only need to compute the sine and cosine once.

```
let s = ... compute the sine ...
let c = ... compute the cosine ...
u = c * x - s * y;
v = s * x + c * y;
```

To support let-statements in our implementation we first have to add two new categories to our definition for syntax trees.

Scope. Part of a program where a certain name binding is valid.

Binding. Association of a name to a program entity.

```
1  data Expr =
2      ENum Integer
3    | EId  Symbol
4    | EPlus  Expr Expr
5    | ETimes Expr Expr
6    | ELet Symbol Expr Expr
```

When we try to add the corresponding cases to the `eval` function, we face a problem of how to determine the value of a variable we want to evaluate. We need some kind of lookup table to store the variables values when they are defined so we can retrieve these values when the variables are used. Such a table is commonly called an *environment*. So we need a data type

```
7  type Environment = Table Integer
```

where `Table a` is some data type supporting the operations

```
lookup_variable :: Table a -> Symbol -> a
bind_variable   :: Table a -> Symbol -> a -> Table a
```

Then we can write the evaluation function as

```
8  eval :: Environment -> Expr -> Integer
9  eval env (ENum n)       = n
10 eval env (EId x)        = lookup_variable env x
11 eval env (EPlus  e1 e2) = eval env e1 + eval env e2
12 eval env (ETimes e1 e2) = eval env e1 * eval env e2
13 eval env (ELet x e b)   = eval (bind_variable env x (eval env e)) b
```

When introducing let-bindings a new phenomenon arises called *scoping*. The problem is, when we try to evaluate an expression and come upon an identifier x, which of the possibly several definitions for x do we use? The part of the code where a particular definition of x is in effect is called the *scope* of the definition. In our case, the scope of a definition `let x = e; e'` is the expression e'. That is, every occurrence of x inside e' refers to the value e. Other occurrences of x (for instance, those in e or in other parts of the program) refer to other definitions. We also say that this definition of x is *local* to e' and that the variable x is *bound* (to the value e) by this definition. In general the association of *names* in a program with the *entities* they refer to is called *binding*. The characteristic property of a local variable is that it can be *renamed* without changing the meaning of the program. (The technical term for such a renaming is *α-conversion*.)

```
let x = 2; x*x   ⟺   let y = 2; y*y
```

In most languages scopes can be nested, but they cannot partially overlap. Therefore, they are usually implemented using a stack.

```
1  let x = 2;
2    let y = x-1;
3    x+y
```

} scope of y } scope of x

2.3 Functions

Next we add function definitions to our language. Function definitions are one of the main mechanisms for *control abstraction* in programming languages. They facilitate *code reuse* and they can increase the *readability* of code by hiding low-level details and thereby revealing the logical structures of the code. But note that overuse of this feature can degrade readability again, if functionality is split over too many places of the code. (This is a common problem with inheritance in object-oriented programming.)

For efficiency reasons, many languages (like C++ and Java) only support *non-nested functions*. In this case, a program is of the form

```
1  let f₁(x) { ⟨expr⟩ };
2  ...
3  let fₙ(x) { ⟨expr⟩ };
4  ⟨expr⟩
```

The implementation is straightforward. We extend our data type for the syntax tree by one more clause.

```
1  data Expr =
2      ENum Integer
3    | EId  Symbol
4    | EPlus  Expr Expr
5    | ETimes Expr Expr
6    | EApp Symbol Expr
7    | ELet Symbol Expr Expr
```

The `eval` function takes an additional argument of type

```
8  type FunDefs = Table (Symbol, Expr)
```

containing the function definitions. This is a table that, for each function, contains the name of the parameter and the function body.

```
9  eval :: FunDefs -> Environment -> Expr -> Integer
10 eval fdefs env (ENum n)       = n
11 eval fdefs env (EId x)        = lookup_variable env x
12 eval fdefs env (EPlus  e1 e2) = eval fdefs env e1 + eval fdefs env e2
13 eval fdefs env (ETimes e1 e2) = eval fdefs env e1 * eval fdefs env e2
```

```
14  eval fdefs env (EApp   f  a)  = let val    = eval fdefs env a        in
15                                  let (p, b) = lookup_variable fdefs f in
16                                  eval fdefs (bind_variable empty_env p val) b
17  eval fdefs env (ELet x e b)   =
18    eval fdefs (bind_variable env x (eval fdefs env e)) b
```

As we have seen when implementing non-nested functions, we can evaluate the body of a globally defined function in the empty environment. When allowing nested functions, we have to use the environment of the function definition instead. This complicates the implementation since we have to store this environment somewhere. We extend our language as follows.

$$\langle expr \rangle ::= \ldots \mid \langle id \rangle \texttt{(} \langle expr \rangle \texttt{)} \mid \texttt{let} \langle id \rangle \texttt{(} \langle id \rangle \texttt{)} \texttt{\{} \langle expr \rangle \texttt{\};} \langle expr \rangle$$

For the implementation, it is convenient to store the function definitions inside the environment. To do so, we have to modify the `eval` function to support two kinds of values: integers and functions. We introduce the following data type for this.

```
1  data Value =
2      VNum Integer
3    | VFun Symbol Environment Expr
```

Then we add one new clause for function definitions to the syntax tree

```
4  data Expr =
5      ...
6    | ELetFun Symbol Symbol Expr Expr
```

and we modify the `eval` function as follows. Besides adding a clause for the new construct, there are mainly two changes to the code: (i) we can remove the argument for the function definitions again and (ii) we have to add a few case distinctions that check for values of the correct type.

```
7  eval :: Environment -> Expr -> Value
8  eval env (ENum n)        = VNum n
9  eval env (EId x)         = lookup_variable env x
10 eval env (EPlus e1 e2)   =
11   case (eval env e1, eval env e2) of
12     (VNum x1, VNum x2) -> VNum (x1 + x2)
13     _                  -> error "addition of non-numbers"
14 eval env (ETimes e1 e2)  =
15   case (eval env e1, eval env e2) of
16     (VNum x1, VNum x2) -> VNum (x1 * x2)
17     _                  -> error "multiplication of non-numbers"
18 eval env (EApp f a)      =
19   let val = eval env a in
20   case eval env f of
21     VFun p e b -> eval (bind_variable e p val) b
22     _          -> error "application to non-function"
```

Static scoping. A function's body is evaluated in the scope of the function's definition.

Dynamic scoping. A function's body is evaluated in the scope of the function call.

```
23  eval env (ELet x e b) =
24    eval (bind_variable env x (eval env e)) b
25  eval env (ELetFun f arg body e) =
26    eval (bind_variable env f (VFun arg env body)) e
```

2.4 Static and dynamic scoping

When invoking a function, *static scoping* evaluates the function body in the scope of the function's *definition,* while *dynamic scoping* uses the scope of the function's *caller.*

Examples:

```
1  let x = 1;
2  let f(y) { x+y };
3  let x = 2;
4  f(3)
5
```

```
let x = 1;
let g(y) { x+y };
let f(y) { g(y) };
let x = 2;
f(3)
```

```
let x = 1;
let g(y) { x+y };
let f(x) { g(0) };
let x = 2;
f(3)
```

Dynamic scoping Examples of languages using dynamic scope are: the original Lisp, Emacs Lisp, TeX, Perl, and many scripting languages including early versions of Python and JavaScript.

Today, dynamic scoping is generally considered to be a mistake. The main problem is that dynamic scoping is not robust: changing local variables in some part of the program can have drastic influences on other parts. Hence, with dynamic scoping bound variables are not local in the sense defined above since renaming them *can* change the meaning of the program. This means that understanding code with dynamic scoping requires *global reasoning* about the program, which violates one of the fundamental principles of code readability.

For instance, consider a GUI library that provides an event-loop where the program can install call-backs to react to user input. If the event-loop and the user code happen to share a variable, the call-back will get the event-loop's variable instead of its own. Problems of dynamic scoping include:

- As seen in the above example, with dynamic scoping, every 3rd party library needs to document the names of all local variables it uses. This makes library maintenance more difficult, as new versions cannot introduce new local variables.
- Dynamic scoping also presents a security risk as it enables other parts of the code to access and modify sensitive information stored in local variables.

Let us conclude with an example of a programming idiom that is enabled by dynamic scope: one can use it to simulate *default parameters* for functions. If a certain function parameter has nearly always the same value, one can use a variable instead. For example, if we write a function converting numbers to strings we might want to support other bases than decimal. The code could look like this.

```
let base = 10;
let num_to_string(n) {
  ... convert n into base base ...
};

let f(x) {
  ...
  let base = 16;
  let str  = num_to_string(137);
  ...
};
```

Of course, with languages supporting default parameters one could simply write:

```
let num_to_string(n, base=10) {
  ... convert n into base base ...
};

let f(x) {
  ...
  let str = num_to_string(137,16);
  ...
};
```

Static scoping While static scoping is clearly superior to dynamic scoping, it is not without its problems. The way it is usually implemented, the scoping structure of a program is determined by its syntactic structure. This is a very simplistic way to specify scoping rules, which is not always adequate. Sometimes one would like to have more fine-grained control over scoping, say, by specifying which parts of the program are allowed to see a given identifier. Some languages have therefore tried to untie scoping from the syntactic structure by making it explicit. One example is the concept of *namespaces* in C++, which allows complete control over scoping. Slightly less general are *modules* or *packages* which are supported by most modern languages.

2.5 Higher-order and first-class functions

In many languages, functions are not values. You cannot assign them to variables or pass them as arguments to functions. Some languages, like C and C++, allow passing functions as arguments, but not returning them as results. This can be used for example to implement call-backs in GUI frameworks. Such languages support what is called *higher-order functions*.

Higher-order function. A function that takes other functions as arguments or that returns a function.

First-class functions. Functions are treated as values.

```
let f(x) { x+1 };
let g(s) { s(1) };
g(f)
=> 2
```

In some languages, like Lisp, ML, or JavaScript, functions are values like any others. In this case we speak of *first-class function*. In such languages, we need an operation to create new functions. This is called a *lambda abstraction*.

⟨*expr*⟩ ::= ... | **fun** **(** ⟨*id*⟩ **)** **{** ⟨*expr*⟩ **}**

Example

```
let adder(n) { fun(x) { x + n } };
let add3 = adder(3);
add3(4)
=> 7
```

First-class functions are frequently used, for instance, in GUI frameworks where they are called *call-backs*.

```
let mouse_button(button,x,y) {
  ...
  react to a mouse button being pressed
  ...
};
register_call_back(MouseDown, mouse_button);
```

In functional programming, first-class functions are one of the main concepts used for reduce dependencies between different parts of the code. They allow the separation of the *action* to be performed on some data structure from the *traversal* of said data structure. For instance,

`map(update, lst)`	applies `update` to every element of `lst`
`fold(sum, 0, lst)`	adds all elements of `lst`

There is not much to do to add support for first-class functions to our interpreter. We add a clause for lambda abstraction to our syntax tree and we also remove the special **`let`**-binding for functions, as we can now use the usual one for both integers and functions.

```
data Expr =
    EId Symbol
  | ENum Integer
  | EPlus  Expr Expr
```

```
5     | ETimes Expr Expr
6     | EFun Symbol Expr
7     | EApp Expr Expr
8     | ELet Symbol Expr Expr
```

In the definition of the `eval` function we have to add one line for `EFun` and we can remove the code for `ELetFun`.

```
9   eval :: Environment -> Expr -> Value
10  ...
11  eval env (EFun arg body) = VFun arg env body
12  ...
```

When using dynamic scoping first-class functions cause additional problems. Traditionally there are two possible ways to implement dynamic scoping for such functions: *shallow binding* and *deep binding*. The question is, which environment is used when calling a function value. With deep binding it is the environment where the function was declared, i.e., the same as when using static scoping. With shallow binding it is the environment of the function call instead, which is more in the spirit of dynamic scoping.

```
1   let f(x) { fun(y) { x } };
2   let x = 3;
3   f(1)(2)
```

Finally, note that, once we have first-class functions, we can simplify our toy language by removing the let-construct and implementing it as syntactic sugar instead.

`let x =` $expr_1$`;` $expr_2$ $\quad\Longrightarrow\quad$ `(fun (x) {` $expr_2$ `})(`$expr_1$`)`

Discussion First-class functions provide a powerful method to decouple various parts of a program from each other: if some code requires functions from other parts, one can pass them as parameters instead of referring to them directly. For instance, many frameworks to build user interfaces make heavy use of call-backs, which are a poor man's version of first-class functions. Another advantage of supporting first-class functions is that they facilitate code reuse: instead of writing several versions of a procedure, one can factor out the general logic and parametrise it by functions doing the specific work. For instance, in functional languages data types often provide a general traversal procedure which takes as parameter a function containing the actual work to be performed for each entry.

The main disadvantage of using first-class functions is that they are less efficient since, in their presence, the activation records of functions do not follow a stack regime anymore. This leads to poor cache behaviour of such programs.

2.6 Function parameters

Multiple parameters As function application is one of the most frequently used mechanism in programming, many languages provide features making it more convenient. The first such feature

Currying. Converting a function with multiple arguments into one with only a single one that returns another function taking the rest.

we consider are functions with multiple arguments. There are two ways to add such functions to our language. The first one implements them as syntactic sugar in terms of first-class functions. This is called *currying*. It is present in many functional languages like OCaml or Haskell. The idea is simple. We view a function with two parameters as a function that take the first argument and returns a function taking the second argument and returning the result.

```
    fun (x,y) { x*x + y*y }

⟹  fun (x) { fun(y) { x*x + y*y }}
```

In our toy language, we can implement currying as syntactic sugar. We translate

```
    let f(x,y,...,z) { body }; expr

⟹  let f(x) { fun (y) { ... fun (z) { body } ... }};
    expr
```

and

```
    f(a,b,...,c)   ⟹   f(a)(b)...(c)
```

Note that this syntactic sugar allows us to use partially applied functions, that is, expressions like the following one.

```
1  let f(x,y) { ... };
2  f(1)
```

If we do not want to allow this, we have to add a pass for arity checks before doing the desugaring.

The other way of implementing functions with multiple parameters does not require first-class functions, but uses a tuple datatype instead. Instead of passing several arguments to a function, we pass a single tuple containing them. This is done for example in Standard ML.

```
    fun (x,y) { x*x + y*y }

⟹  fun (p) { p.x * p.x + p.y * p.y }
```

Keyword parameters One such feature are *named parameters* or *keyword parameters*. Ordinarily, arguments are passed to a function by *position*, that is, the i-th argument will be bound to the i-th formal parameter of the function. If a function takes many arguments, it becomes hard to remember the correct order of the parameters. In many languages it is therefore possible to assign names to the parameters and use these names when invoking a function. In this case, the arguments can be given in any order.

```
1  let f(serial_number, price, weight) { ... };
2
3  f(serial_number = 83927, weight = 60, price = 120);
```

To avoid ambiguities, if both positional and keyword parameters are used in the same function call, one usually requires all positional parameters to be listed first.

Default arguments Another feature are *default arguments*. When a function is frequently called with a fixed value for some argument, one can specify this value as the default and allow the programmer to omit the argument from a function call.

```
let int_to_string(num, base = 10) { ... };

int_to_string(17)
```

To avoid ambiguities, if positional parameters are used in a function call where some default argument is omitted, one usually requires all arguments after the omitted one to be keyword parameters.

Variable number of arguments Some languages like C allow the definition of functions where the number of arguments is not fixed. There is a minimal number of arguments, but every function invocation can use more if needed.

```
let printf(format, ...) { ... };

printf("f(%d) = %d", x, f(x));
```

Conceptually what happens is that the first arguments are passed to the function as usual and the remaining arguments are passed in an array which the function body can inspect.

Implicit arguments Such arguments were invented in Scala. They work similarly to default arguments. When no argument is specified, some default value is used. But this default value is not constant. Instead the compiler searches the current scope for a value of the correct type and uses that one.

```
let f(x : int, implicit p : bar) {
  ...
  f(x-1)      // uses p for the second argument
  ...
}
```

As always when one adds an implicit mechanism like this to the language, implicit parameters provide some convenience for the programmer, but they can make the resulting code much harder to understand.

2.7 Conditionals

In preparation for adding recursion, let us implement conditionals first. These are needed to add a termination condition to a recursive function call. For simplicity, we only support equality predicates.

⟨*expr*⟩ ::= ... | **if** ⟨*expr*⟩ **==** ⟨*expr*⟩ **then** ⟨*expr*⟩ **else** ⟨*expr*⟩

Truthy/Falsy values.	Are implicitly converted to *true* and *false* in a conditional.

Examples

```
let f(n) {
  if n == 0 then
    0
  else
    n-1
};
```

The implementation is straightforward. We add one clause to the definition of the syntax trees

```
data Expr =
    ...
  | EIf Expr Expr Expr Expr
```

and a corresponding clause to the `eval` function.

```
eval :: Environment -> Expr -> Value
...
eval env (EIf c1 c2 t e) =
  let v1 = eval env c1 in
  let v2 = eval env c2 in
  case (v1,v2) of
    (VNum n1, VNum n2) -> if n1 == n2 then eval env t else eval env e
    _                  -> error "comparison of non-numbers"
```

There are two approaches to boolean values in programming languages. Languages with a strict type discipline define a type for boolean values and demand that the condition in an if-statement is of that type. Languages with a looser type discipline allow the condition to be of a different type and automatically coerce it to a boolean value. For such languages one uses the terminology of *truthy* values (those that are treated as *true*) and *falsy* values (those that are treated as *false*).

Automatic coercions are more convenient, but also more error-prone and make the type system much more complicated. (In general, coercions also make the code harder to understand, but for booleans in conditionals that is not an issue.) While for languages with a simple type system like, say, C, the rules for boolean conversions are easily understood and remembered. But for languages with a richer type system like JavaScript, Python, or Ruby, these rules become very complicated. (Is the empty array *false*? What about the empty string, or the string `"0"`? Are `"00"` and `"0.0"` treated the same as `"0"`?) What makes matters worse is that none of these languages agree on the precise rules.

2.8 Constructors and pattern matching

So far, our toy language does not support any composite data structures. We only have numbers and functions. To add composite types like records and arrays, we need operations that create

new data objects. In imperative languages like Java there are usually two different kinds of such operations. There is an operation like `new` that allocates a piece of memory that then has to be initialised by the programmer. Furthermore, some of the types allow the programmer to write down values of the type directly. These constructs are called *literals* or *(data) constructors.* In a language without side-effects, we cannot initialise a data structure after it has already been allocated. We have to do both in one step. Therefore such languages usually only have constructors.

For our toy language we will provide several built-in constructors and also allow user-defined ones. Each number literal is treated as a constructor. Furthermore, we have constructors for records, two constructors `True` and `False` for the boolean values, constructors `()` and `Pair(`x`,` y`)` for the empty tuple and pairs, and two constructors `Cons(`x`,` y`)` and `Nil` to build lists. Using these last two constructors, we can represent a list like `[1,2,3]` in the form

```
Cons(1, Cons(2, Cons(3, Nil))) .
```

The more convenient notation `[1,2,3]` will be provided as syntactic sugar.

We add three new constructs to the language. We can define new constructors, we can call a constructor to create a data structure, and we can match a given data structure with a template to extract its fields.

⟨*expr*⟩ ::= ... | **type** ⟨*id*⟩ **=** **|** ⟨*variant*⟩ ... **|** ⟨*variant*⟩ **;** ⟨*expr*⟩
| **type** ⟨*id*⟩ **=** **[** ⟨*id*⟩ **=** ⟨*id*⟩ **,** ... **,** ⟨*id*⟩ **=** ⟨*id*⟩ **];** ⟨*expr*⟩
| ⟨*ctor*⟩ **(** ⟨*expr*⟩ **,** ... **,** ⟨*expr*⟩ **)**
| **[** ⟨*id*⟩ **=** ⟨*expr*⟩ **,** ... **,** ⟨*id*⟩ **=** ⟨*expr*⟩ **]**
| ⟨*expr*⟩ **.** ⟨*id*⟩
| **case** ⟨*expr*⟩ **|** ⟨*pattern*⟩ **=>** ⟨*expr*⟩ **|** ... **|** ⟨*pattern*⟩ **=>** ⟨*expr*⟩

⟨*pattern*⟩ ::= ⟨*id*⟩ | ⟨*num*⟩ | ⟨*ctor*⟩ **(** ⟨*id*⟩ **,** ... **,** ⟨*id*⟩ **)** | **else**

⟨*variant*⟩ ::= ⟨*id*⟩ | ⟨*id*⟩ **(** ⟨*id*⟩ **,** ... **,** ⟨*id*⟩ **)**

For instance, we can create a pair and extract its two components again using the following definitions. (The arguments a and b in the definition of the constructor P are only used to specify the arity. Later on when we add a type system, these parameters will specify the types of the constructor arguments.)

```
1  type int_pair = | P(int, int);
2
3  let make_pair(x,y) { P(x,y) };
4
5  let fst(p) {
6    case p
7    | P(x,y) => x
8  };
9
10 let snd(p) {
11   case p
```

```
type int_pair = [ x : int, y : int ];

let make_pair(x,y) { [ x = x, y = y ] };

let fst(p) { p.x };

let snd(p) { p.y };
```

```
12      | P(x,y) => y
13    };
```

Similarly, we can define the following functions to create and traverse lists.

```
1   let empty_list         = Nil;
2   let add_to_list(x,lst) = Cons(x,lst);
3
4   let is_nil(lst)  { case lst | Nil        => True | else => False };
5   let is_cons(lst) { case lst | Cons(x,xs) => True | else => False };
6   let head(lst)    { case lst | Cons(x,xs) => x  };
7   let tail(lst)    { case lst | Cons(x,xs) => xs };
```

The implementation of the support for these kinds of data structures is rather straightforward, but the code is a bit lenghty, in particular the evaluation function for **case** statements. We will therefore only highlight a few interesting points. The full implementation can be found in the accompanying source code. First of all, we need to add two new kinds of values: constructors and records.

```
1   data Value =
2       VNum Integer
3     | VCtor Symbol Int [Value]
4     | VRec (Table.T Symbol Value)
5     | VFun Symbol Environment Expr
```

For constructors we store (i) its name, (ii) its arity, and (iii) the value of all the arguments that were already provided. So, the expressions

```
Pair                Pair(4)              Pair(4,7)
```

evaluate to the values

```
VCtor "Pair" 2 []    VCtor "Pair" 1 [4]    VCtor "Pair" 0 [4, 7]
```

respectively. For records, we simply store a table mapping field names to their values.

We also need to add a few more cases to the data type for expressions.

```
6   data Expr =
7       ...
8   | ERecord [(Symbol, Expr)]
9   | EMember Expr Symbol
10  | ETypeVar [(Symbol, Int)] Expr
11  | ECase Expr [(Pattern, Expr)]
```

The implementation of the `eval` function is straightforward. The only interesting point is that, for function applications, we know have to distinguish whether the first expression denotes a function or a constructor.

```
12  eval env (EApp f a)      =
13    let val = eval env a in
14    case eval env f of
15      VFun p e b         -> eval (bind_variable e p val) b
16      VCtor ctor 0  args -> error "constructor applied to too many arguments"
17      VCtor ctor ar args -> VCtor ctor (ar-1) (args ++ [val])
18      _                  -> error "application of non-function"
```

With the case-construct we can now implement if- and (non-recursive) let-statements as syntactic sugar.

```
if c₀ == c₁ then t else e   ⟹   case c₀ - c₁ | 0 => t | else => e
let x = e; e'               ⟹   case e | x    => e'
```

In fact, we can now define the equality predicate explicitly and introduce a version of the if-statement that uses an arbitrary predicate.

```
e == e'                ⟹   case e - e' | 0 => True | else => False
if c then t else e     ⟹   case c      | True => t | False => e
```

Exercise Use case statements to define syntactic sugar for `and` and `or` operations that evaluate their arguments only as needed (short-circuit evaluation).

```
e₁ and e₂    ->   case e₁ | True => e₂    | False => False
e₁ or e₂     ->   case e₁ | True => True | False => e₂
```

Exercise Introduce syntactic sugar for lists.

```
[e₁,...,eₙ]    ->  Cons(e₁, Cons(..., Cons(eₙ, Nil)...))
[e₁,...,eₙ|e]  ->  Cons(e₁, Cons(..., Cons(eₙ, e)...))
```

Introduction and elimination forms Let us conclude this section with a remark on *introduction* and *elimination* constructs. In many aspects of a programming language, we can observe a duality between constructs introducing a certain object and ones eliminating it again. For instance, with data types we have (I) constructors to assemble a structure and (E) the case-statement, the array indexing operation, etc. to disassemble it into its components again. Similarly, for functions we have (I) lambda abstractions which create new functions and (E) function applications which turn functions into their return value.

2.9 Recursion

Every serious programming language needs a mechanism for *unbounded recursion* or iteration. For instance, we would like to define recursive functions like the following one.

```
let fac(n) { if n == 0 then 1 else n * fac(n-1) };
```

Recursion. Defining a name in terms of itself.

(Which is, in fact, a *bounded* (by n) recursion.) Implementing recursion is rather straightforward: in let-bindings `let x = e; e'` we just have to extend the scope of x to include e.

Note that, while straightforward, the addition of recursion *does* change the language considerably. In particular, it is now very easy to write *non-terminating programs*. (Strictly speaking, this is also possible in our old language with non-recursive let-bindings, but it requires some tricks and a lot of effort to do so, see below.)

From a theoretical perspective, this addition is much more involved, and books on programming language theory usually devote quite some space to the topic. The problem is how to implement recursion without using recursion. (We cheated in our implementation by using the built-in recursion of Haskell.) There are two ways to get around this problem.

The first one requires mutable state. When defining a recursive function f, we first allocate a variable for it (initialised with some dummy value), then we write the actual function into the variable using an assignment.

```
1  let f  = fun (x) { x };        // dummy value
2  let f' = fun (x) { ... body using f ... }
3  f := f'
```

This is what most real language implementations do.

The second solution is much cleaner from a theoretical point of view. We add a *recursion operator* (also called a *fixed-point operator*) to the language which is defined by

```
rec(f) = f(rec(f))
```

(Of course, this is a recursive definition itself.) Then we can write

```
1  let fac_body(f) {
2    fun (n) { if n == 0 then 1 else n * f(n-1) }
3  };
4  let fac = rec(fac_body);
```

`fac_body(f)` is the body of the factorial function where we have replaced the recursive call by a call to the function `f`. Then we tie the knot by defining

```
fac = rec(fac_body) = fac_body(rec(fac_body)) = fac_body(fac)
```

Intuitively, the `rec` operator provides a marker indicating that 'at this position there is a recursive call'. Whenever such a marker is evaluated, we insert the body of the corresponding function (where all recursive calls are marked by `rec` again).

If our language is untyped (or if the type system supports recursive types), we can actually define `rec` as syntactic sugar.

```
let rec(f) = (fun (x) { f(x(x)) })(fun (x) { f(x(x)) });
```

Then `rec(f)` evaluates to `f(rec(f))` (try it).

Simultaneous recursion Our let-construct only allows the definition of a single recursive function. Sometimes one would like to define several mutually recursive functions like

```
let f(x) { if x = 0 then 1 else g(x-1) };
let g(x) { if x = 0 then 1 else 1+f(x-1) };
```

There are three ways to implement such definitions. The first one is to extend the syntax of let-bindings to allow for the simultaneous definition of several identifiers. This is the most practical solution and implemented in all serious programming languages. In our toy language we will not take this approach (just to make our life easier, at the cost of making the programmers life harder). We can do so because simultaneous recursion can be implemented using single recursion. Suppose we have a definition like

```
let x = f(x,y) and y = g(x,y);
h(x,y);
```

We can either transform it into

```
let x = f(x, (let y = g(x,y); y));
let y = g(x,y);
h(x,y)
```

or, if the language supports tuples or records (see the next section), we can use them to write

```
let (x,y) = (f(x,y), g(x,y));
h(x,y);
```

The first solution duplicates some code (which is then executed twice), the second one has to allocate memory for the tuple. In most cases this overhead is negligible.

Recursive data structures Since we are already talking about data structures, let us also mention the related problem of creating mutually recursive data structures. The most practical solution is again to use mutable data structures. Then we can (i) first allocate all the memory and then (ii) initialise it. For instance, to create two pairs

```
let p = (1, q) and q = (2, p);
```

we can write

```
let p = (1, 0);
let q = (2, 0);
p.2 := q;
q.2 := p;
```

Tail calls Finally, let us mention an important implementation detail. In a programming language where the only mechanism for unbounded iteration is recursion, it is essential that this feature is usable even if the number of iterations is large. For every recursive call, we have to allocate memory to store parameters and local variables. In a naive implementation this memory will only be freed once all recursive calls have returned. This leads to a memory consumption that

is linear in the number of recursive calls, which is problematic if this number is large. There is an important situation where we can free this memory earlier: if the recursive call is the last expression of our function, i.e., the return value of the function is the value returned by the recursive call.

```
let find_next_prime(n) {
  if n is prime then
    n
  else
    find_next_prime(n+1)
};
```

```
let fac(n) {
if n == 0 then
    1
  else
    n * fac(n-1)
};
```

In the situation on the left, we will not need the parameters and local variables after the recursive call has returned. Hence, we can free the memory containing them before the call to `find_next_prime` instead of after it. This is called a *tail-call optimisation*. It amounts to replacing the recursive call by a jump to the beginning of the function.

```
let find_next_prime(n) {
  label start;
  if n is prime then
    n
  else
    (n := n+1; goto start)
};
```

After this transformation the function will use a constant amount of memory and is as efficient as an imperative solution using a while-loop.

Frequently, it is possible to transform a recursive definition that uses non-tail calls into a tail-call one by using an *accumulator*. For instance, we can define the factorial function as

```
let fac(n) {
  let multiply(n, acc) {
    if n == 0 then
      acc
    else
      multiply(n-1, n*acc)
  };
  multiply(n,1);
};
```

After tail-call optimisation, this looks like

```
let fac(n) {
  let multiply(n, acc) {
    label start;
    if n == 0 then
      acc
    else (
```

Eager evaluation. The evaluation of an expression proceeds from the left-most, inner-most operation.

Lazy evaluation. The evaluation of an expression proceeds from the left-most, outer-most operation.

```
7         new_n   := n-1;
8         new_acc := n*acc;
9         n       := new_n;
10        acc     := new_acc;
11        goto start;
12    };
13    acc := 1;
14    goto start;
15  };
```

which (after some trivial optimisations) is equivalent to the imperative code

```
1   let fac(n) {
2     let acc = 1;
3     while n > 0 {
4       acc := n * acc;
5       n   := n - 1;
6     };
7     return acc;
8   };
```

2.10 Lazy evaluation

Since our language does not support side-effects, the order in which we evaluate expressions does not matter. Any order we choose produces the same result.

```
fun (x) {1+x*x} (1+1)
=> fun (x) {1+x*x} 2
=> 1+2*2
=> 1+4
=> 5
```

```
fun (x) {1+x*x} (1+1)
=> 1+(1+1)*(1+1)
=> 1+2*(1+1)
=> 1+2*2
=> 1+4
=> 5
```

There are two canonical orders in which we can evaluate expressions:

- *eager evaluation* evaluates an expression starting with the left-most, inner-most operation, while
- *lazy evaluation* starts with the left-most, outer-most operation.

Advantages of lazy evaluation It can be shown that lazy evaluation is more powerful than eager evaluation in the following sense: every computation that terminates using an *arbitrary* evaluation order also terminates with lazy evaluation and produces the same result. On the other hand, there are expressions that terminate with a result using lazy evaluation but not with eager evaluation.

It has turned out that there are two main areas where this property of lazy evaluation makes it superior to eager evaluation:

(1) processing of infinite data structures and

(2) evaluations of (mutually) recursive definitions.

(1) Using data constructors with lazy evaluation, it is very simple to define and process infinite data structures like infinite lists.

```
let ones = [1 | ones];
ones

let numbers i = [i | number(i+1)];
numbers

let add(x,y) { x+y };
let fib = [0, 1 | map2(add, fib, tail(fib))];
fib
```

But note that this means that there are no inductive lazy datatypes. For example

```
type nat = Zero | Succ(nat)
```

does not define the natural numbers since

```
let omega = Succ(omega);
```

defines the infinite number `Succ(Succ(Succ(...)))`. Hence, in order to be able to define inductive datatypes like natural numbers, finite lists, or finite trees, a lazy language must have support for eagerly evaluated data constructors.

(2) The definition of the Fibonacci sequence above is also an example of a recursive definition, that is very easy to write down using lazy evaluation, but much more involved when using eager evaluation. For instance, the following code computes the list of the first n Fibonacci numbers.

```
// lazy
let fib =
  [0, 1 |
    map2(add, fib, tail(fib))];
let fib_list(n) = take(n, fib);
```

```
// eager
let fib_list(n) {
  let iter(i,a,b) {
    if i == n then
      []
    else
      [a+b | iter (i+1,b,a+b)]
  };
  [0, 1 | iter(2,0,1)]
};
```

Disadvantages of lazy evaluation On the flip side, lazy evaluation has also severe disadvantages. The most prominent one is that it cannot be combined with side-effects as it obscures the order in which expressions are evaluated, which is of paramount importance in computations with side-effects.

Furthermore, it turned out that it is very hard to predict the memory consumption of programs using lazy data structures since one is never quite sure when a structure will be constructed and when the program is done processing it, so the garbage collector can free it again.

2.11 Programming examples

Let us conclude this chapter with several examples of programs in a functional style. We concentrate on functions for list processing.

```
1  let compose(f,g) {
2    fun (x) { f(g(x)) }
3  };
4  let iterate(f,n) {
5    if n == 0 then
6      fun (x) { x }
7    else
8      compose(f, iterate(f,n-1))
9  };
10
11 type list =
12   | Nil
13   | Cons(a, b);
14
15 let nth(lst,i) {
16   if i == 0 then
17     head(lst)
18   else
19     nth(tail(lst), i-1)
20 };
21
22 let length(lst) {
23   case lst
24   | Nil        => 0
25   | Cons(x,xs) => 1 + length(xs)
26 };
27
28 let sum(lst) {
29   case lst
30   | Nil        => 0
31   | Cons(x,xs) => x + sum(xs)
```

```
32  };
33
34  let map(f, lst) {
35    case lst
36    | Nil        => Nil
37    | Cons(x,xs) => Cons(f(x), map(f, xs))
38  };
39
40  let fold(f, acc, lst) {
41    case lst
42    | Nil        => acc
43    | Cons(x,xs) => fold(f, f(acc, x), xs)
44  };
45
46  let foldr(f, acc, lst) {
47    case lst
48    | Nil        => acc
49    | Cons(x,xs) => f(x, foldr(f, acc, xs))
50  };
51
52  let reverse(lst) {
53    let iter(lst, result) {
54      case lst
55      | Nil        => result
56      | Cons(x,xs) => iter(xs, Cons(x,result))
57    };
58    iter(lst, Nil)
59  };
60
61  // tail recursive version of foldr
62
63  let foldr(f, acc, lst) {
64    let g(x,y) { f(y,x) };
65    fold(g, acc, reverse(lst))
66  };
```

Exercise Write an implementation of balanced binary search trees in the toy language as it is defined so far.

3 Types

3.1 Static and dynamic typing

In most languages there are operations that cannot be performed on every kind of input. For instance, division might be defined for numbers, but not for strings. For this reason one distinguishes several *types* of data. In some languages such as Haskell, Scala, or Rust, the type system is extremely sophisticated and subject to active research, other languages make do with rather impoverished type systems. For instance, the original Fortran had only two types: integers and floating point numbers.

Traditionally, there are two radically different ways of implementing types: *static typing* and *dynamic typing*. In static typing, every identifier of the program is associated with some type and the compiler ensures that the value of the identifier will always be of that type. In dynamic typing on the other hand, the types are not associated with the identifiers but with the values themselves. That means that every value in memory is *tagged* with its type and these tags are checked by all operations performed on the value. Each choice has its advantages and disadvantages.

Dynamic typing

- is very slow: every operation performs runtime checks of the types,
- catches only type errors in those parts of the program that are executed,
- is more permissive and more convenient: no type annotations or other kinds of red tape.

For these reasons, dynamic typing is mainly useful in scripting languages, but not for writing non-trivial programs.

Static typing

- is stricter and catches more errors,
- the compiler can *prove* that the program is free of type errors,
- there is no runtime overhead,
- it can sometimes be inconvenient: the programmer has to write additional code in order to make correct code actually compile,
- not all properties can be checked statically (e.g., array bounds),

Static typing. Types are checked at compile-time.

Dynamic typing. Types are checked at run-time.

- with sophisticated type systems, the error messages from the type checker can be hard to understand,
- type annotations help document code,
- types help with refactoring,
- static type information can provide implicit context that changes the behaviour of a piece of code (e.g., with overloading).

Good static type systems try not to get in the way of the programmer. For instance, ML-like languages provide static type checking without requiring any kind of type annotations. Unfortunately, other languages are much less successful in this respect, think for example of template code in C++.

For serious software development, static type checking has turned out to be indispensable. First of all, we can use it as a means for the compiler to automatically *prove* that the program does not contains certain kinds of errors. The more expressive the type system is, the more kinds of errors we can catch.

Secondly, types also help with program design. When tasked with writing a certain submodule of a program, many programmers first design the types and data structures of the data involved. Then they use these types as a guide to write the actual code.

Thirdly, experience has shown that a good type system helps with refactoring large programs: after a change in one place of the program, the type checker can tell you all the other places you have to change as well.

Finally, let us note that the advantages of type checking apply much more to symbolic computations, than to numeric code (e.g., it doesn't catch sign errors).

3.2 Type annotations

To implement static typing in our toy language, we add type annotations to every declaration. (This is not strictly necessary as there exist algorithms to *automatically infer* the types from a program without annotations. We will discuss such algorithms below.)

$$\begin{aligned}\langle expr\rangle \ ::= \ldots \mid\ & \texttt{let}\ \langle id\rangle\ \texttt{:}\ \langle type\rangle\ \texttt{=}\ \langle expr\rangle\ \texttt{;}\ \langle expr\rangle \\ \mid\ & \texttt{let}\ \langle id\rangle\ \texttt{(}\ \langle id\rangle\ \texttt{:}\ \langle type\rangle\ \texttt{)}\ \texttt{:}\ \langle type\rangle\ \texttt{\{}\ \langle expr\rangle\ \texttt{\};}\ \langle expr\rangle \\ \mid\ & \texttt{fun(}\ \langle id\rangle\ \texttt{:}\ \langle type\rangle\ \texttt{)}\ \texttt{:}\ \langle type\rangle\ \texttt{\{}\ \langle expr\rangle\ \texttt{\}}\end{aligned}$$

We also need to define which types the language provides. In our case we have the base type `int` for integers, function types *a* `->` *b*, and one type `foo` for every type declaration

```
type foo = | A(a,b,...) | ... | Z(c,d,...);
```

in the program. So far, the parameters `a,b,c,d,...` in constructor declarations served only to denote the arity of the constructor. Now we require them to be type expressions specifying the type of the constructors arguments. For instance, if we want `A` to take an integer and a boolean, we write `A(int, bool)`.

Basic type. An atomic type without parts.

Composite type. A type built up from several other types.

Examples

```
let fac(n: int) : int {
  if n == 0 then 1 else n * fac(n-1)
};

let compose(f: int -> int, g: int -> int): int -> int {
  fun (x: int) { f(g(x)) }
};
let twice(f : int -> int): int -> int {
  compose(f, f)
};
let apply : (int -> int) -> int -> int =
  fun (f: int -> int) {
    fun (x: int) {
      f(x)
    }
  };

type int_list = IntNil | IntCons(int, int_list);

let sum(l : int_list) : int { ... };
```

Exercise What could be the type of the following function?

```
let f = fun (x) { x(x) };
```

(Note that `f(f)` evaluates to `f(f)`.)

3.3 Common Types

Let us give a short overview of types that are commonly found in real programming languages. Generally, we distinguish between *basic* and *composite* types. The former are atomic and built into the language, while that latter are composed out of one or several simpler types. Hence, basic types are types such as

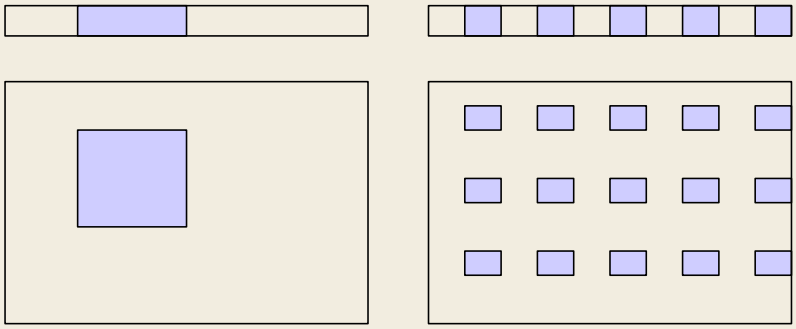
- integers, signed and unsigned, of various precisions, including arbitrary precision integers,
- floating point numbers of various precisions, decimal numbers (`0000.00`), and arbitrary precision rational numbers,

- integer ranges (`1..100`),
- enumerations (`enum colours { Red, Green, Blue, Yellow }`),
- booleans,
- characters,
- strings,
- the empty type and the unit type,

while the composite types include

- arrays,
- pointers and references,
- functions and procedures,
- records and tuples,
- unions and variants,
- lists and maps or dictionaries.

Arrays Arrays are homogeneous (all elements have the same type) collections of values. Some languages come with a very elaborate support for arrays. The language Fortran shines in this area, as it was specifically designed for numeric computations where arrays play an important rôle. In particular, Fortran supports higher-dimensional arrays and efficient array *slices*, which are (not necessarily contiguous) subsets of an array. For instance, one can define a slice consisting of the first 16 elements of every other row of an array. The important aspect of a slice is that it does not make a copy of the array, but only provides a new way to index the elements of the old array.



From the point of view of the type system, a point of concern is the fact that it is not possible to statically check that all array accesses are within bounds. This would be a very desirable thing to have, as array overflows are a very common source of bugs and security problems. Therefore, modern languages usually add dynamic bounds checks to each array access. (Usually these can be turned off selectively at places where efficiency matters.)

Product type. Several components are laid out in memory next to each other.

Sum type. All components share the same memory. Only one is usable at the same time.

Product types Products are similar to arrays, but they are inhomogeneous (the elements can have different types) and their size is fixed. Set-theoretically they correspond to a cartesian product. Commonly the components of a product are labelled and can be referred to by their name. In this case such types are usually called *records* or *structures.*

```
type triple = int * int * int;
type vector = [ x : float, y : float, z : float ];
```

Languages supporting product types come in two flavours depending on how the components of a product are accessed. If the language has *first-order fielding* the component is fixed at compile time, while a language with *first-class fielding* allows the runtime computation of the component. For instance, if we write `r.x` to access the field named `x` of a record `r`, we know the field at compile time. But if we can write `r.(`*e*`)` with an arbitrary expression *e*, it is only known at compile time which field we are accessing. Clearly, first-class fielding is much more expressive than first-order fielding, but it is unfortunately not possible to combine it with static type checking. For example, in

```
type foo = [ x : int, y : bool ];

r.(if i = 0 then x else y)
```

we cannot say, whether the expression evaluates to an integer or a boolean. For these reasons, first-class fielding is usually found only in dynamically typed scripting languages. One example, where it is rather useful is in writing serialisation and deserialisation code.

Sum types Sum types (also called *tagged unions*) are dual to products. Instead of storing several values at the same time, a sum type contains only a single value whose type may be one of a given list of types. Set-theoretically they correspond to a disjoint union.

```
type int_list = | Nil | Cons(int, int_list);
type expr = | Num(int) | Plus(expr, expr) | Times(expr, expr);
type nat = | Zero | Suc(nat)
```

In languages with sum types, one usually combines them with products, i.e., one allows the user to specify a type as a sum of products as in the example above. In this case one speaks of *variant types* or *algebraic types.*

Variant types are frequent in ML-like languages, but not well-supported by C-based or Pascal-based ones. C++ has *enums* which can be seen as sum types where the constructors have no arguments. It also has *untagged unions,* which can simulate sum types by adding the tag manually. Pascal supports a case-statement inside records which serves the same purpose as a sum type.

Note that sum types add a dynamic component to a type system. For instance, if we have a value of type

Unit type. A type with a single value.

Void type. A type without any values.

```
type either = | Left(int) | Right(bool);
```

it is unknown at compile time whether it is an integer or a boolean. Hence, we have to tag the value with its variant (`Left` or `Right`). Note that this is the same thing we do in set theory, where a disjoint union is usually defined as

$$A + B := \{0\} \times A \cup \{1\} \times B\,.$$

Here the first component (0 and 1) serves as a tag distinguishing the elements of A and B.

Unit and void type One has to distinguish between a *unit* type which has exactly one value, usually the empty tuple,

```
type unit = | Nothing;
```

and the *void* type which has no values at all.

```
type void = ;
```

If we want to treat procedures as functions with a special return value, this value must be of a unit type, since a function must return a value but we do not care which one it is. A function whose return type is void cannot return at all as it would have to produce a value of void type to do so.

Recursive types Most programming languages have at least some support for recursively defined types such as

```
type expr = | Num(int) | Plus(expr, expr) | Times(expr, expr);
```

Note that a value of the form, say, `Plus(`e_1`,`e_2`)` is not stored in memory by having a memory segment consisting of a tag and two copies of the value e_1 and e_2 (which can be arbitrarily large). Instead, the memory segment contains the tag and *pointers* to the two argument values. In many languages one is only allowed to define recursive types if the recursion is via such pointers. Some languages have full support for recursive types by allowing arbitrary recursive definitions. Unfolding such definitions produces a possibly infinite type expression. For instance,

```
type t = t -> t
```

is the type of all functions from `t` to `t`. It unfolds to

```
type t = (... -> ...) -> (... -> ... )
```

This is the type of the self-application function.

```
let f(x : t): t = { x(x) };
```

This means that with full support for recursive types, we can type the recursion operator as

Name equivalence. Two types are only equivalent if they have the same name.

Structural equivalence. Two types are equivalent if they have the same definition.

```
type b = b -> a;
let rec(f : a -> a) : a =
  (fun (x : b) : a { f(x(x)) })
  (fun (x : b) : a { f(x(x)) });
```

3.4 Type checking

Type equivalence Before being able to type check, we have to decide when we allow an argument of a given type *a* to be passed to a function expecting arguments of some, possibly different, type *b*. Clearly, this should be the case if the two types are *equivalent*. But what does being equivalent mean? There are basically two choices.

- With *name equivalence* two types are considered to be the same if they have the same name. Examples of languages using name equivalence are Pascal, C and their descendants.
- With *structural equivalence* two types are considered to be the same if they have the same structure, even if their names might be different. Languages in the ML-family typically use this kind of equivalence.

Example In C, which uses name equivalence for structures, all of the following types are considered to be distinct since they have different names.

```
type vector = [ x : int, y : int ];
type pair   = [ x : int, y : int ];
type pair2  = [ y : int, x : int ];
```

In ML the corresponding definitions would all define the same type, so we could pass a `pair2` to a function expecting a `pair`.

Example Suppose we want to use types to distinguish between measurements in metric units and in imperial units. How to do so depends on which kind of equivalence the type system uses.

```
// name equivalence
type metric   = float;
type imperial = float;

let f(x: metric): metric {
  ... 2*x ...
};
let x : imperial = 10;
f(x)  // error
```

```
// structural equivalence
type metric   = M(float);
type imperial = I(float);

let f(x: metric) : metric {
   ... 2 * unpack(x) ...
};
let x : imperial = I(10);
f(x)  // error
```

Type coercion. An implicit type conversion.

Type cast. An explicit type conversion.

Type conversions There are cases where we can allow passing arguments to a function even if the types are not equivalent. For instance, this is the case when we can *convert* the argument to the expected type. For example, in C one can pass an integer to a function expecting a floating point argument and it will automatically converted into a floating point number. When such conversions are done automatically by the compiler, we speak of *type coercions*. Some languages like C, Perl, or JavaScript are very liberal with regard to type coercions, while other languages, like ML and Haskell do not allow coercions at all. Except for scripting languages, modern programming languages usually try to reduce the amount of coercions.

On the one hand, coercions are convenient since they make the code shorter and cleaner. On the other hand, they make the code harder to understand (implicit behaviour) and can hide type errors. This is the usual trade-off between an implicit effect and an explicit one. In moderation they can make the life of the programmer easier, but when overdone they easily create a mess.

Some languages with a permissive type system also allow *type casts*. A type cast is a command telling the compiler to regard a value as having a user-specified type instead of its real one.

There are several kinds of conversion between types (either in a coercion or a type cast). If every value of the first type has the same memory representation as the corresponding value of the second type, we can just change the type and there is no run-time overhead. If the memory representations differ (e.g., if we convert an integer to a floating-point number), we have to insert code that does the conversion. Some languages also support *non-converting type casts*. Such casts never change the memory representation, even if this does not make sense semantically. This feature makes the type system *unsound,* but it can be useful for system programming. For instance, in C one can cast from any pointer-type to any other in this way.

An additional complication arises if not every value of one type can be converted to the other type. In such cases one has to add a runtime check ensuring that the conversion is possible. For example, in object-oriented languages one sometimes wants to cast an object of a superclass to one of its subclasses. In this case a runtime check is needed to make sure that the object is in fact of the required class.

Type checking After these preliminary remarks, let us finally turn to type checking itself. For the simple type system we have chosen for our toy language, which is basically what the older mainstream languages like C and Pascal did provide, this is straightforward.

```
1  let fac(n : int) : int {
2    if n == 0 then 1 else n * fac(n-1)
3  };
```

Polymorphism. The phenomenon where a piece of code can be used with several types.

Ad-hoc polymorphism. Providing several definitions for the same identifier, each with its own type

3.5 Polymorphism

In the typing examples above, we have seen that, when adding type annotations to a program, we sometimes have to make arbitrary choices since some functions could be used with different types. For instance, the identity function

```
fun (x) { x }
```

can be given the type `int -> int`, or `bool -> bool`, or `(int -> bool) -> (int -> bool)`, and so on. It would be desirable, to use the same function definition for all suitable types instead of requiring a separate definition (with the same code) for each of them. This phenomenon is called *polymorphism*. Most modern programming languages support it in one form or other. One can broadly distinguish three different forms of polymorphism.

(i) *ad-hoc polymorphism*, also called *overloading*,

(ii) *parametric polymorphism* as can be found in ML-like languages, and

(iii) *subtyping polymorphism* as is present in object-oriented languages.

Ad-hoc polymorphism In ad-hoc polymorphism the programmer can define several versions of a function. Which of them will be selected when the function is called will depend on the types of the supplied arguments. A typical example are arithmetic operations which in many languages are defined both for integers and floating point numbers.

```
1  + : int -> int -> int
2  + : float -> float -> float
3  + : string -> string -> string
```

Ad-hoc polymorphism is the most flexible form of polymorphism since it allows the programmer complete freedom. The disadvantage is that one has to write several different versions of each function which can become quite a chore. Furthermore, if ad-hoc polymorphism is used extensively the program can become hard to understand as it will be difficult to figure out which version of the function will be called at each call site.

Parametric polymorphism In parametric polymorphism we allow type expressions to contain *type variables*. For instance, we can specify the type of the `map` function as

Parametric polymorphism. Type expressions may contain parameters. Each identifier has a single definition that works for all types.

Subtyping polymorphism. A subtyping relation is defined between values. Every value is implicitly converted to its supertypes.

Type inspection. Making decisions based on types at compile-time or runtime.

```
map : (a -> b) -> list(a) -> list(b)
```

with two variables a and b. This is a simple and quite clean extension of the type system with few drawbacks. But is is less flexible than ad-hoc polymorphism. Most of the functional programming languages have adopted this version of polymorphism.

Subtyping polymorphism This kind of polymorphism is based on the *subtyping relation*. We say that a type *a* is a *subtype* of *b* if every value of type *a* can be used where a value of type *b* was expected. This is a situation that commonly arises in object-oriented languages where objects of a subclass automatically also belong to their superclasses. We will discuss subtyping polymorphism in more detail in Chapter 8. As far as the expressive power is concerned, there are things that subtyping can express, which parametric polymorphism cannot, and vice versa. Both approaches have their merits, but they have a very different *feel* to them. While parametric polymorphism is conceptually quite simple, subtyping makes a type system very complex.

Type inspection Some languages provide mechanisms to inspect the type of a polymorphic value either at compile-time or at runtime. In this way a polymorphic function can behave differently depending on which type is supplied. A prominent example is serialisation, where an arbitrary value is converted to a string.

```
let serialise(value) {
  case type_of(value)
  | int    => int_to_string(value)
  | bool   => bool_to_string(value)
  | string => sanitise_string(value)
  | cons   => "cons(" ++ serialise(fst(value)) ++ ","
                      ++ serialise(snd(value)) ++ ")"
  | ...
};
```

Type inspection is a way to add the power of ad-hoc polymorphism back to a system based on parametric or subtyping polymorphism. It makes the type system much more powerful, but also less uniform and more complex.

Data polymorphism So far, we have looked at polymorphic functions. Data structures can also be polymorphic. For instance, the general list type and the types of the two constructors are

```
type list(a) = | Nil | Cons(a, list(a));
Nil  : list(a)
Cons : a -> list(a) -> list(a)
```

Type inference. Automatically computing the types at compile-time.

3.6 Type inference

Writing explicit type annotations at every declaration can become quite tedious, in particular, if we use a sophisticated type system where the type expressions are quite large (see, for instance, template code in C++). Many modern languages therefore implement a form of *type inference* where the types of expressions are automatically derived from the code without the help of type annotations. The amount to which this is possible strongly depends on the type system. In ML-like systems, type inference is possible without restrictions. In more complicated type systems, we need some type annotations but can infer others. The original type inference algorithm for ML was developed by Damas, Hindley, and Milner. Therefore, one frequently speaks of Hindley-Milner type inference. Given an expression the algorithm looks at every subexpression and extracts a list of equations between the types involved and solves them.

Example

```
let twice(x) { 2 * x };
```

We start by associating a type variable with every subexpression.

$2 : \alpha$
$x : \beta$
$* : \gamma$
$2*x : \delta$
$\texttt{fun (x) \{ 2*x \}} : \varepsilon$

We already know that

$\alpha = \texttt{int}$ and $\gamma = \texttt{int * int -> int}$.

By looking at each subexpression in turn, we obtain the following additional equations.

$2*x :$ $\gamma = \alpha * \beta \texttt{ -> } \delta$
$\texttt{fun (x) \{ 2*x \}} :$ $\varepsilon = \beta \texttt{ -> } \delta$

Solving them we obtain.

$\alpha = \texttt{int}$, $\beta = \texttt{int}$, $\gamma = \texttt{int * int -> int}$, $\delta = \texttt{int}$, $\varepsilon = \texttt{int -> int}$.

Example

```
let compose(f,g) { fun (x) { f(g(x)) } }
```

Unification. Solving an equation between terms.

Again we start by associating type variables with subexpressions.

```
x : α
f : β
g : γ
g(x) : δ
f(g(x)) : ε
fun (x) { f(g(x)) } : ξ
compose : η
```

The equations are

$$\gamma = \alpha \to \delta$$
$$\beta = \delta \to \varepsilon$$
$$\xi = \alpha \to \varepsilon$$
$$\eta = \beta \to \gamma \to \xi$$

which lead to the solution

$$\eta = \beta \to \gamma \to \xi = (\alpha \to \delta) \to (\delta \to \varepsilon) \to \alpha \to \varepsilon\,.$$

Unification The process of solving a single type equation $s = t$ is called *unification*. Conceptually, the algorithm is very simple. If s or t is a type variable, we can set its value to be the other term. Otherwise, we check that the outermost operator of both type expressions is the same and recursively unify the arguments.

$$\begin{aligned}
x = t &\quad\rightsquigarrow\quad x := t \\
s = x &\quad\rightsquigarrow\quad x := s \\
s \to s' = t \to t' &\quad\rightsquigarrow\quad s = t \wedge s' = t' \\
c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n) &\quad\rightsquigarrow\quad s_1 = t_1 \wedge \cdots \wedge s_n = t_n \\
s = t &\quad\rightsquigarrow\quad \text{failure}
\end{aligned}$$

Type inference has its advantages and disadvantages. On the one hand, it is very convenient, relieving the programmer of the burden of having to annotate every declaration with a type. Furthermore, it will find the *most general type* for an expression and automatically introduce polymorphism. On the other hand, having explicit type annotations serves as a kind of documentation and improves the readability of the code (explicit vs. implicit information). Furthermore, error messages from a type checker with type inference are usually more complicated and harder to read. One reason for this is that the equation-based approach of type inference obscures the *location* of the type error. The algorithm only determines that some equations are inconsistent, but it cannot deduce *which* of them is the cause.

3.7 Advanced topics

Much of the current research on programming languages focuses on type theory. In this last section we briefly discuss a few of the more experimental usages of type systems.

Linear types Type systems with linear types keep track of how many ways there are to access a given value. This information can be used in several ways.

- It can be used for resource management: if a certain resource is no longer accessible, it can safely be deallocated.
- It can be used to manage mutable state: a mutable value can only be modified if it cannot be accessed by another part of the program.
- It can be used as a synchronisation mechanism for shared-memory concurrency: no mutable value should be accessible from different threads.

A prominent example of a language with a linear type system is Rust, which uses linear types for all of these tasks.

Dependent types Dependent types are parameteric types where the parameters are not necessarily other types, but might also be other values. For instance, one could have a type `array(n)` for arrays of length `n`, where the parameter `n` ranges over integers. An example of a language supporting dependent types is Idris. Contrary to many other such languages, the focus is here not on theorem proving, but on programming.

Gradual types Languages with gradual typing provide a mixture of static and dynamic typing. The idea is to use dynamic typing in the early prototyping phase of a project and to incrementally transition to static typing later on.

4 State and Side-Effects

4.1 Assignments

In its current state our toy language is purely functional, that is, running a program amounts to evaluating a mathematical expression that produce some value and nothing more. In this chapter we will add *side effects* to the picture. With side effects expressions do not only produce a value, but they can also modify the state of the world in certain ways, say, changing the contents of memory cells, drawing on the screen, or reading keystrokes from the keyboard. These are all essential features no serious general purpose programming language can do without. Even so-called purely functional languages must therefore support side effects, but they do it in a way which is separated from the rest of the program. For instance, a Haskell program consists of two phases. The first phase is pure and does not allow side effects. It computes a list of commands that *do* have side effects. This list is then executed in the second phase.

We start by extending our toy language with two commands providing different kinds of side effects: an assignment statement to alter the contents of a memory cell and a print statement to produce screen output.

$$\begin{aligned}\langle expr\rangle ::= \ldots &\mid \textbf{skip} \mid \textbf{print}\ \langle msg\rangle\ \langle expr\rangle \mid \langle expr\rangle\ \textbf{;}\ \langle expr\rangle \\ &\mid \langle id\rangle\ \textbf{:=}\ \langle expr\rangle\end{aligned}$$

```
1  let x = 1;
2  print "x has value: " x;
3  x := 2;
4  print "now x has value: " x;
```

With these new statements we cannot regard an expression e anymore as a mathematical function $env \to val$ that, given values for the free identifiers in e, produces a value. Instead, we also have to specify its effect on the state of the world. That is, an expression now determines a function $env \times state \to val \times state$. In our case the state must contain the memory contents and also the produced output.

Note that, with assignments identifiers no longer represent constant values but *variables* instead. A variable in this context is an identifier associated with a *location* in memory which contains the value stored in the variable. This means that the notion of an environment is changed from a function mapping names to values to one mapping names to memory locations.

Side effects. Additional effects of an expression besides returning a value.

Referential transparency. The property that expressions are indistinguishable from their values.

We have seen that in a language with assignments we must distinguish between expressions denoting values and those denoting memory locations. Only the latter can appear on the left-hand-side of an assignment, while the right-hand-side can contain both kinds of expressions. One frequently uses the terminology of *l-values* and *r-values* for locations and values, respectively. Here, the *l* and the *r* specify on which side of an assignment the expression can appear.

In our toy language, the only l-values are variables and expressions for structure access `r.m`. In real imperative languages like C several other kinds of expression can be l-values, for instance, expressions for array indexing `a[i]`.

4.2 Ramifications

The support for side effects has a drastic influence on all aspects of a programming language. Let us mention a few aspects.

Evaluation order First of all, with side effects the *order of evaluation* becomes important. Until now we could not have cared less about in which order subexpressions were evaluated (if we ignore termination issues for the moment), but with assignments and IO the order matters. For instance,

```
1  let x = 0;
2  let y = (x := 1; 3) + (x := 2; 4);
3  x + y
```

returns either 8 or 9 depending on which term in the definition of *y* is evaluated first. This means that with side effects we have to define an evaluation order, preferably one that can easily be read off from the syntactic structure of the code. This rules out lazy evaluation, where it is very hard for the programmer to determine in which order expressions are evaluated.

Failure of referential transparency Referential transparency refers to the property that an expression can be replaced by its value without changing the behaviour of a program. This makes reasoning about programs much simpler. But it clearly fails for an expression with side-effects (values have no side-effects). For instance, a `print` statement has type `unit`, but we cannot replace it by `()`.

Uninitialised data structures Another new effect is that assignments allow for uninitialised or partially initialised data structures. Such things are not possible in a purely functional language since there is no way to retrospectively initialise objects. Partial initialisation is very convenient when creating mutually recursive data structures. We can first allocate the memory for all the structures and then fill in the pointers between them. Of course, having uninitialised data structures is also a source of bugs when such a structure escapes into a part of the program that expects

Aliasing. A situation where two different expressions refer to the same memory location.

Deep copying. Copying a data structure and all structures reachable from it via pointers.

its inputs to be fully initialised. (This is a common problem when writing constructors in, say, C++ as constructor code is executed while the objects in question is not yet fully initialised. So all methods called inside a constructor have to work with a partially initialised object.)

Aliasing With assignments we have to distinguish two notions of equality. Two objects can have *the same value* or *the same memory location.* We can tell these two apart by changing the value of one object. If the value of the second object also changes, they share the same memory location, otherwise their locations are distinct. Having the same memory location accessible through several variables or expressions is called *aliasing.* For instance, consider the following code.

```
1  let x = 1;
2  let y = x;
3  x := 2;
4  y
```

Depending on the semantics of our programming language `y` will or will not alias `x` and the code will return either `1` or `2`.

When working with mutable data structures, aliasing has to be strictly controlled. If a piece of code wants to modify a data structure and it does not know whether there is aliasing involved, it has to make a copy of the structure before modification. In big programs written by a large team of programmers, it is not always clear at which places aliasing can occur. Therefore, one commonly makes copies of data 'just to be sure'. This leads to a lot of unnecessary copying. For instance, in one version of the Chrome web browser, profiling revealed that every single keystroke in the URL field the browser resulted in several thousand memory allocations. This copying does not only slow down the program it also wastes space. When working with immutable data structures, one can have them share those parts they agree on. For instance, we can have two lists share a common tail or two search trees share common subtrees. This is a common situations for data structures in functional languages.

In light of the aliasing effect, a language designer has to decide what to do if a data structure gets assigned to a variable. The most efficient solution is to just let the variable point to the same object without making a copy. As we have discussed, this creates aliasing. If one wants to avoid aliasing, one has to make a copy of the data structure and, recursively, of all data structures reachable from the given one via pointers. This approach is called *deep copying.* It is quite slow and memory inefficient. There is also a compromise where only the first structure is copied, but not the pointed to structures. This approach, called *shallow copying,* is clearly inferior to the other two: it is less efficient than the first one, does not avoid aliasing, and it is also more complicated for the programmer. We will discuss these different strategies more below in the section on parameter

Shallow copying. Copying a data structure without duplicating other structures pointed to.

passing.

Cleanup code Finally, since our code can now affect the state of the system, it needs to clean up when it is done by freeing the allocated resources like, say, memory, file handles, or windows. This means that we have to make sure that every code path leaving this part of the program calls the cleanup code. In practice, this can be a lot of work and rather a nuisance. It is also quite error prone as it is easy to forget to free one or two of the resources. Note that, in addition to direct returns we also have to check indirect ones like exceptions.

```
...
let a = allocate_a();
if error then
  return
...
let b = allocate_b();
if error then {
  free(a);
  return
}
...
let c = allocate_c();
if error then {
  free(b);
  free(a);
  return
}
...
```

Many languages have added special constructs to help with cleanup. For instance, in Java a block can be annotated with a finally-statement which contains code that is always executed when control leaves the block.

```
let a = nil;
let b = nil;
let c = nil;
{
  ...
  a := allocate_a();
  if error then return;
  ...
  b := allocate_b();
  if error then return;
  ...
  c := allocate_c();
  if error then return;
```

```
    ...
  }
  finally {
    if c then free(c);
    if b then free(b);
    if a then free(a);
  }
```

A similar idea is to have a defer-statement which specifies commands to be executed when leaving the current block.

```
  let a = nil;
  let b = nil;
  let c = nil;
  {
    ...
    a := allocate_a();
    if error then return;
    defer free(a);
    ...
    b := allocate_b();
    if error then return;
    defer free(b);
    ...
    c := allocate_c();
    if error then return;
    defer free(c);
    ...
  }
```

Discussion Side effects drastically increase the power of a language. There are algorithmic problems that have very simple solutions using side effects, but where the corresponding side-effect free solutions are extremely cumbersome or inefficient. Furthermore, every serious language must support some form of IO, which is not possible without side effects.

On the flip side, side effects make the code much more complicated to reason about. They add implicit interactions between different parts of a program, for instance, via mutable global variables. This reduces encapsulation, makes the program harder to understand (non-local reasoning), and the coding more error prone.

So side effects are necessary but dangerous. Therefore it is desirable for a language to have some sort of separation between pure and impure code. This separation was already present in Algol which distinguishes between expressions and commands. A modern example is Haskell, which is particularly strict in this regard. Other languages are much more relaxed. For instance in ML or C++, one can declare variables to be constant (the default in ML) or mutable (the default in C++). This can be used to limiting side effects. So far, none of the solutions are really satisfactory. Either

the separation is too lenient to offer real protection against side effects in places where they are not needed; or it is too strict making it very cumbersome. For instance, if during development one discovers that some part of a Haskell program would profit from a use of side effects, it is frequently necessary to rewrite large (and mostly unrelated) parts of the program to make the type system happy.

4.3 Parameter passing

Having introduced assignments and mutable state, we have to decide how it interacts with parameter passing. When we change a variable inside a function, does this effect become visible on the outside?

```
1  let f(x) { x := 1; };
2  let y = 0;
3  f(y);
4  y
```

Some languages allow the programmer to annotate function definitions with the desired behaviour for the parameters. For instance, Ada distinguishes between *in-mode, out-mode,* and *in/out-mode* parameters. In-mode parameters allow the passage of value from the call site to the function, out-mode parameters allow the passage in the opposite direction, and in/out-mode parameters can be used for both. Most other languages only provide in-mode parameters. Let us take a closer look at the various parameter passing mechanisms.

Call-by-value is the standard mechanism for in-mode parameters. When calling a function, the argument values are passed as copies to the function body. Modifications of the copies do not affect the originals. This is a very safe method that avoids any confusion cause by unexpected modifications. The disadvantage is that it can be inefficient if large objects are passed in this way.

Call-by-result is the analog of call-by-value for out-mode parameters. No value is passed during the function call. Instead, when the function returns, the current contents of the variable corresponding to the parameter is copied back to the argument, which *must be an l-value*. Call-by-result has the same advantages and disadvantage as call-by-value. There are two additional problems that need to be addressed.

(i) What happens if the same variable is passed to two different out-mode parameters?

```
1  f(in x, out y, out z) {
2    y := x+1;
3    z := x+2;
4  };
5  let u = 0;
6  f(u,u,u);
```

(ii) At what time is the l-value passed as argument evaluated?

```
f(in x, out y, out z) {
  y := x+1;
  z := x+2;
};
let i = 0;
f(i, array[i], i);
```

Call-by-value-result/call-by-copy/call-by-copy-result combines call-by-value and call-by-result for in/out-mode parameters. The argument value is copied to the parameter when the function is called and copied back, when it returns.

```
let u = 1;
let f(x) {
  print "u is " u;
  x := 2;
  print "u is now " u;
};
f(u);
print "u is now " u;
```

Call-by-reference is a more efficient version of call-by-value-result. Instead of copying the value back-and-forth, its address is passed to the function. Every Modification inside the function directly affects on the original l-value. This is very efficient, but can create aliasing problems.

```
let u = 1;
let v = 0;
f(x, y) {
  x := x + u - v;
  y := y + u - v;
};
f(u, v)
```

```
f(x, y) { x := 1; y := 2; };
g(x, y) { y := 2; x := 1; };
let u = 0;
f(u, u); print "after f:" u;
g(u, u); print "after g:" u;
```

Call-by-name is a radically different calling convention invented in Algol. Here the expression given as argument is substituted for the formal parameter in the function body using a *capture-avoiding substitution,* i.e., all local variables in the function will be renamed to avoid name clashes. In an implementation this amounts to passing the argument expression as a *thunk* (a suspended computation). This calling convention is the basis for implementing lazy evaluation. For code

without side-effects, we have seen that call-by-name is nearly indistinguishable from call-by-value (except for issues of termination). In combination with side-effects, call-by-name is radically different from call-by-value.

```
let i = 0;
let array = [1,2];
let p(x) {
  i := x;
  x := 0;
};
p(array[i]);
print i array[0] array[1];
```

A famous example of call-by-name is what is called *Jensen's device*. The function

```
let sum(k, l, u, expr) {
  let s = 0;
  for k = l .. u {
    s := s + expr;
  };
  s;
};
```

computes $\sum_{k=l}^{u} expr$ where the expression can be passed as an argument.

- `sum(i, 0, 99, array[i])` sums the first 100 entries of an array.
- `sum(i, 1, 100, i*i)` sums the first 100 square numbers.
- `sum(i, 0, 3, sum(j, 0, 3, m[i,j]))` sums the entries of a 4×4 matrix.

Call-by-need is an optimised version of call-by-name useful for in-mode parameters. It is the standard calling convention used in lazy functional languages like Haskell. Here, after the first evaluation of a passed argument expression, the result is stored, so subsequent uses of the parameter do not need to evaluate the expression again. Of course, this only works if the argument expression has no side effects.

Call-by-macro-expansion is also similar to call-by-name but uses textual substitutions instead of capture-avoiding ones. Hence, the function works like a macro. This calling convention has its uses in a few limited cases, but it is clearly unsuited as the main calling convention of a language. Besides it being hard to implement efficiently (in particular, if recursion is involved), it also introduced non-local effects via unintended variable capturing. In particular, renaming local variables can change the behaviour of a program.

More examples

```
let f(x,y) { x := 2; y := 3; x };
let u = 1;
```

```
3   let v = 1;
4   let w = f(u,v);
5   print "u is now: " u;
6   print "v is now: " v;
7   print "w is now: " w;
8   let w = f(u,u);
9   print "u is now: " u;
10  print "v is now: " v;
11  print "w is now: " w;
12
13  let swap(x,y) { let tmp := x; x := y; y := tmp };
14  let a = 1;
15  let b = 2;
16  swap(a,b);
17  print "a is now: " a;
18  print "b is now: " b;
```

Discussion The consensus today is that one does want to have call-by-value in languages with side effects and call-by-need in languages without. The reason for call-by-value is to avoid *aliasing*, in particular *variable aliasing* where writing to one variable can change the contents of another one. There is also *data structure aliasing* where part of a data structure is accessible from different variables. If one wants to avoid this as well, we have to copy entire data structure when passing them to a function. This process is called *deep copying* as it involves following all pointers in the structure and recursively copying the memory pointed to. Since deep copying is very inefficient, it is implemented by only a few languages. Some languages prove a compromise where only the structure directly pointed to is copied, but no recursive copying occurs. This is called *shallow copying*. Shallow copying has fallen out of favour, as it does not really solve the problem of aliasing and it is still more inefficient than call-by-value. Therefore, most languages today use call-by-value where non-scalars, i.e., composite data structures, are passed by pointer. Some allow the simulation of call-by-reference by using explicit *reference* or *pointer types*. There is one exception: in a logical language call-by-reference is more natural, as it better fits the semantics expected by the user.

```
head([X|XS], X).

p(L) :- head(L, X), q(X).
```

In such languages, the problems of call-by-reference is reduced considerably as variables usually only support single-assignments (see Chapter 7), not multiple ones.

4.4 Memory management

When adding assignments we introduced the notion of a store. Our naive implementation added values to the store but never removed them again. In a real implementation this is of course

Reference couting. Each memory block contains a count of how many pointers point to it.

Garbage collection. The heap is scanned periodically to free unreachable memory blocks.

unacceptable. Programs would run out of memory. So every real programming language must have some form of memory management that frees unused values in memory. There are three forms of memory management.

- In *manual memory management* the programmer is responsible for (nearly all) allocations and deallocations of memory blocks. (The exception is memory for local variables, which is usually managed automatically on a stack.)
- In *automatic memory management* the runtime system of the language performs allocations and deallocations automatically.
- In *type based memory management* the type system tells the compiler at which places it has to allocate and deallocate memory.

There are two types of problems memory management has to address.

- *Dangling pointers,* that is, pointers to already deallocated memory block. These can lead to program crashes and other undefined behaviour.
- *Unreachable objects,* that is, objects that are still allocated, but no longer reachable via pointers. These waste memory but are otherwise harmless.

Manual memory management For manual memory management, the language provides two operations to the programmer: one to allocate a certain amount of memory and one to deallocate it again. It is the responsibility of the programmer to make sure that memory that is not needed anymore is actually freed. Of course, this is quite tedious and error prone. It is easy to either forget to free memory, or to free it too soon. Both kinds of errors are hard to debug as the place where the error is made is usually not the place where its effects manifest.

Most implementations of manual memory management use a list (or several) of free memory blocks. If a certain amount of memory is to be allocated, this list is traversed until a block of suitable size is found. If later on the memory is freed again, it is simply added to the list. In actual implementations the picture is a bit more complicated as several techniques are added to increase efficiency. In particular, one should note that, in this scheme, allocating and freeing memory are both operations which take a non-negligible amount of time.

Automatic memory management With automatic memory management the programmer is relieved of the burden of managing memory herself. There are two main approaches. The first one is called *reference counting.* Here, every memory object has a counter which stores the number of pointer to this object. If, at some point, this counter reaches zero, the object is automatically deleted. The other approach is based on *garbage collection.* Here, objects are not freed right away. Instead, the program continues to allocate memory until the remaining amount of free memory

drops below a certain threshold. Then the memory manager determines which part of the allocated memory is actually in use and frees the rest.

Reference counting is easy to implement, but very slow and it cannot deal with cyclic data structures. Garbage collection on the other hand, is very hard to implement well. But it has the advantage that allocations are very fast (usually just a pointer increment and a compare) and deallocations do not take any time at all. Of course there is also the collection phase, which can take quite some time. How much depends on the kind of collection being performed. We distinguish the following cases.

- During collection the whole program is stopped. This is the easiest to implement, but it causes latency problems.
- A collection is split into several pieces, which are interleaved with the program execution. This somewhat reduces the latency problem.
- The garbage collector and the main program run in parallel. This is very hard to implement well, but it completely eliminates the latency problem at the cost of further increasing the garbage collection overhead.

Type based memory management This is a novel approach to memory management and still experimental. The only mainstream language currently implementing it is Rust. Here, one uses the type system to encode information about the lifetime of objects. Objects are deallocated when the type system says that they are dead. For instance, if an object is locally defined in some scope and no references to the object are passed out of the scope, we know that we can safely delete the object when the scope terminates. This approach tries to retain the safety guarantees of automatic memory management while avoiding its overhead. It remains to be seen how practical it will turn out to be.

Discussion Automatic memory management has clear advantages over manual management. It guarantees the absence of certain kinds of memory errors which historically have been the cause of many program crashes and security breaches. It also makes the code shorter and cleaner as the programmer does not need to write cleanup code. Finally, there are scenarios where it is even faster than manual memory management.

On the other hand, it also has several disadvantages. First of all, it is quite complex and hard to implement. In particular, if it is to be parallelised or if one wants to address real-time requirements. Furthermore, many of the faster garbage collection algorithms waste quite a bit of memory (frequently only half of the real memory is usable). And finally, even with all optimisations, there is still a considerable overhead associated with garbage collection. This makes it unusable for certain applications with strict performance requirements like, say, computer games.

4.5 Loops

The imperative analogue of recursion is a loop. We distinguish two kinds of loops: *bounded* and *unbounded* ones. A loop is bounded if the number of iterations is known beforehand. So for-loops

are bounded and while-loops unbounded.

$$\langle expr \rangle \ ::= \ldots \mid \textbf{while}\ \langle expr \rangle\ \textbf{\{}\ \langle expr \rangle\ \textbf{\}} \mid \textbf{for}\ \langle id \rangle\ \textbf{=}\ \langle expr \rangle\ \textbf{..}\ \langle expr \rangle\ \textbf{\{}\ \langle expr \rangle\ \textbf{\}}$$

There is a more fundamental primitive that can be used to implement loops: the goto-statement. A goto is an unconditional jump that transfers the program execution to the specified location.

$$\langle expr \rangle \ ::= \ldots \mid \textbf{label}\ \langle id \rangle \mid \textbf{goto}\ \langle id \rangle$$

Using gotos we can replace a while-loop

```
while cond { expr }
```

by the following code:

```
label start;
if cond then (
  expr;
  goto start
)
else
  skip
```

Similarly, a for-loop

```
for i = first to last { expr }
```

can be translated to

```
let i = first;
let l = last;
label start;
if i == l then
  skip
else (
  expr;
  i := i + 1;
  goto start
)
```

Although goto is more expressive than for- and while-loops, it has the disadvantage that it can easily lead to unreadable code jumping willy-nilly from one location to another. The nesting imposed by loops prevents this kind of spaghetti code. There are several guidelines for the clean use of goto-statements. The simplest one is to only allow forward jumps in the code, but no backward ones. It can be shown that, if the language supports while-loops and if-statements, we can eliminate every goto by restructuring the code. For these reasons many modern programming languages have no goto-statements.

There are situations where the lack of a goto-statement leads to rather cumbersome code. The most common one is when one wants to jump out of the middle of a loop. Here a solution using an if-statement is rather inelegant, in particular if several such jumps are needed.

```
let terminate = False;

while ... and not(terminate) {
  ...
  if ... then
    terminate := True
  else {
    ... rest of the loop ...
  }
}
```

As this situation arises quite frequently, most languages provide specialised statements for them. A **break** statement terminates the innermost loop, a **continue** statement skips the rest of the loop's body and directly continues with the next iteration, and a **return** statement terminates the current function and returns to the caller.

⟨*expr*⟩ ::= ... | **break** | **continue** | **return** ⟨*expr*⟩

In some languages, it is also possible to jump out of nested loops by adding a label to the break- or continue-statement.

```
for i = 0 to 10 {
  for k = 0 to 10 {
    ...
    continue i;
    ...
  }
}
```

4.6 Programming Examples

We have argued above that the use of side-effects can be problematic as it can make a program much harder to understand. On the other hand, judicial use of side-effects can also greatly simplify a program. Let us give some examples.

Recursive data structures As already explained in the section on recursion, we can use side-effects to create truly recursive data structures: first, we allocate all the memory needed for the various parts of the structure, then we initialise it and create all references between them.

Optimisation In certain cases using mutable variables makes an implementation more efficient. If we update some value and do not need the old value anymore, we can store the new value at the same memory location instead of allocating new memory. A typical example are accumulator variables used in loops. For instance, the list functions of Section 2.11 can be written using mutable variables in the following way.

```
let length(lst) {
  let len = 0;
  while is_cons(lst) {
    len := len+1;
    lst := snd(lst);
  };
  len
};

let sum(lst) {
  let s = 0;
  while is_cons(lst) {
    s   := s + head(lst);
    lst := snd(lst);
  };
  s
};

let map(f, lst) {
  while is_cons(lst) {
    fst lst := f(fst(lst));
    lst     := snd(lst);
  }
};

let fold(f, acc, lst) {
  while is_cons(lst) {
    acc := f(acc, fst(lst));
    lst := snd(lst);
  };
  acc
};
```

Another common example are mutable data structures such as hash tables, search trees, etc. When programming in a functional style we have to create a new copy of the data structure whenever we update it. (Frequently, we do not need to copy the *whole* structure though, since we can share those parts that do not need to be modified with the old copy.) If we allow mutation, we can change the structure in place, which is usually more efficient. Of course, if we do so and we still need the old version of the structure, we have to manually make a copy first (which is less efficient as the functional implementation since in this case we usually cannot use sharing of parts of the structure).

Communication We can use mutable data structures to communicate between parts of the code. For example, if we want to implement a random number generator, we have to pass its state

from one invocation to the next. In a functional implementation, the generator takes the form

```
random : state -> (state, int)
```

Hence, we have to pass the current state of the generator to every place where we want call this function and we have to pass the new state back to the next invocation. This is very tedious and decreases the readability of the code quite a bit. In an implementation with side-effect, we can store the current state in a mutable variable.

```
1  let state = ... some initial value ...;
2
3  let random(): int {
4    state := (1103515245 * state + 12345) mod 2147483647;
5    state
6  };
```

The problem with this use of side-effects is that it can make the program much harder to understand. Instead of explicitly passing values between the program parts in question, we do so implicitly by storing them in some shared variable. Hence, the programmer cannot understand one part of the program without the other, which violates the principle of local reasoning.

5 Modules

5.1 Simple modules

As programs grow larger it is necessary to divide them into manageable units commonly called *modules, packages,* or *program units.* A module is a part of a program with a well-defined interface that lists all the identifiers (and their types) defined within.

$$\begin{aligned}\langle \textit{expr} \rangle &::= \ldots \mid \textbf{module}\ \langle \textit{id} \rangle\ \textbf{\{}\ \langle \textit{declarations} \rangle\ \textbf{\}} \mid \textbf{module}\ \langle \textit{id} \rangle\ \textbf{=}\ \langle \textit{module-expr} \rangle \\ &\quad \mid \langle \textit{module-expr} \rangle\ \textbf{.}\ \langle \textit{id} \rangle \mid \textbf{import}\ \langle \textit{module-expr} \rangle \\ \langle \textit{module-expr} \rangle &::= \langle \textit{id} \rangle \mid \langle \textit{module-expr} \rangle\ \textbf{.}\ \langle \textit{id} \rangle\end{aligned}$$

Every module creates its own namespace. To access its elements, other parts of the program must prefix the identifier with the module name. Alternatively, one can use an **import**-statement to include the namespace of the module in the current one.

```
1   module Stack {
2     type stack(a) = list(a);
3
4     let empty = [];
5
6     let top(s)     { head(s) };
7     let pop(s)     { tail(s) };
8     let push(s, x) { [x|s]   };
9   };
10
11  ...                                   import Stack;
12  let s = Stack.empty;                  let s = empty;
13  ...                                   ...
14  Stack.push(s, 13);                    push(s, 13);
15  ...                                   ...
```

5.2 Encapsulation

The module mechanism addresses two ergonomic issues. Firstly, they help us manage namespaces and avoid name clashes between identifiers. Note that this could also be solved by adopting a strict coding style where, for instance, all identifier names in a given program unit start with a prefix indicating that unit. But this manual solution is cumbersome for the programmer (for instance, the convenience of import statements is lost) and not enforced by the compiler.

Encapsulation. Allowing access to part of the program only through a well-defined interface.

Secondly, they help to decompose the program into smaller, easier to understand parts (which themselves might be divided further into submodules). To understand such a program we only need to understand each component separately and then look at the way they interact. The second part is the easier the more limited the interaction between modules is. This is where declarative programming styles shine. If the modules are written declaratively, they can only interact via their specified inputs and outputs. We do not have to take further interactions into account, say, via mutable global state as is the case when using side effects.

This second use of modules is an example of a mechanism called *encapsulation*. Generally, encapsulation is the process of separating part of the program from the rest and allowing access only via a specified *interface*. This has several advantages.

First of all, as we have already explained above, it makes the program easier to understand since it reduces the amount of code a programmer must be aware of when looking at some part of the program. In particular, users of a module only need to know its interface, not the actual implementation. This is called *information hiding* and is the main way encapsulation contributes to program readability.

But note that when dividing a program into modules one needs to strike a balance. The smaller the modules are, the easier each of them is to understand in isolation. At the same time, as the size of modules decreases their number increases, and so does the interaction between modules, which makes the program as a whole harder to understand again. A good rule of thumb seems to be to organise the division into modules by themes. Put types and functions with a common purpose into one module. As a slogan: for every module one should be able to give a single-sentence description of what it contains.

Secondly, encapsulation can be used to guarantee the *integrity* of data maintained by a module, since only code within the module is allowed to directly access the inner representation the data. In this way a module can enforce certain invariants a data structure must satisfy. (For instance, the requirement on red and black nodes in a red-black tree.)

Finally, it helps with program maintainability as one is free to change the inner representation of a modules data without affecting the rest of the program.

Encapsulation is the single most important mechanism to make programs more readable, with no associated runtime overhead. When used correctly it has no drawbacks. But note that, when used incorrectly, it can both make a program harder to understand and decrease its performance significantly. The problem is that we have to choose the division into modules carefully and to come up with good interfaces. Both of these can easily be gotten wrong and require a lot of experience to do well. A bad division makes a program harder to understand as it becomes unclear what the task of each module is. A bad interface makes it cumbersome to use a module and can increase both runtime and code size considerably. Let us give some examples.

(a) If an interface does not make certain information accessible, it might force its users to repeatedly recompute this information.

(b) If the interfaces of some modules use different types for the same kind of data (e.g., old style C-strings and `std::string` in C++), it might be necessary to frequently convert between

the various representations.

(c) If the interface of a module is stateful, we might be required to make frequent copies of data structures to prevent the module from modifying them.

As an example, some years ago a Chrome developer reported: "In the course of optimizing SyzyASan performance, the Syzygy team discovered that nearly 25000 (!!) allocations are made for every keystroke in the Omnibox. We've since built some rudimentary memory profiling tools and have found a few issues:

- strings being passed as `char*` (using `c_str`) and then converted back to string
- Using wrappers to `StringAppendV`, such as `StringPrintf`, to concatenate strings or for simple number conversions (see note below)
- Using a temporary set to search for a specific value, which creates $O(n)$ allocations and runs in $O(n \cdot \log n)$ time, only to call `find` on it to return `true`/`false` if the value is present. Ouch!
- Not reserving space in a vector when the size is known"

The simple module system defined in the previous section supports the separating part of encapsulation, but not the interface part. For full encapsulation, we need to add a mechanism to restrict the access to the names defined in a module. There are basically two ways to do so. One is to allow definitions to be declared as either *public* or *private*. Only the public definitions are accessible from the outside. This is the method chosen by C++ and by Java for class definitions.

An alternative method, used in ML and also in C header files, for example, is to provide every module with a separate interface specification containing declarations of all identifiers visible to the outside. It requires more typing from the programmer, but it spacially separates the interface from the implementation. This makes it easier to read and also allows some more advanced mechanisms for module handling which we shall introduce below.

5.3 Abstract Data Types

An *abstract data type* is what we get when we apply the concept of encapsulation to the implementation of a data type. More concretely, an abstract data type is a data structure (usually defined inside a module) where the *representation* of the data, i.e., the *concrete implementation,* is hidden from the rest of the program (*information hiding*). The only access is via the operations defined in its *interface* (*encapsulation*). For instance, note that in most languages, built in types can be considered abstract, although this is a somewhat degenerate case.

Example Let us take a look at an abstract data type for stacks. We start with a functional version. The interface is

```
1  module Stack {
2    type stack(a);
3    let empty : stack(a)
4    let push  : stack(a) * a -> stack(a);
5    let top   : stack(a) -> a;
```

```
    let pop   : stack(a) -> stack(a);
};
```

and the implementation is

```
module Stack {
  type stack(a) = list(a);
  let empty : stack(a) = nil;
  let push(st : stack(a), x : a) : stack(a) {
    pair(x, st)
  };
  let top(st : stack(a)) : a {
    case st | pair(x,xs) => x
  };
  let pop(st : stack(a)) : stack(a) {
    case st | nil => nil | pair(x,xs) => xs
  };
};
```

The next version is imperative. The interface is

```
module Stack {
  type stack(a);
  let create : unit -> stack(a);
  let empty : stack(a) -> bool;
  let push  : stack(a) * a -> unit;
  let top   : stack(a) -> a;
  let pop   : stack(a) -> unit;
};
```

and the implementation is

```
module Stack {
  let create() : stack(a) {
    [ elements = nil ]
  };
  let empty(st : stack(a)) : bool {
    is_nil(st.elements)
  };
  let push(st : stack(a), x : a) : unit {
    st.elements := [x|st.elements]
  };
  let top(st : stack(a)) : a {
    head(st.elements)
  };
  let pop(st : stack(a)) : unit{
    st.elements := tail(st.elements)
```

```
16    };
17  };
```

5.4 Parametrised modules

Most programming languages only offer a simple module system like the one presented above. A notable exception is ML where one can parametrise modules by other modules.

$$
\begin{aligned}
\langle \textit{expr} \rangle &::= \ldots \mid \textbf{module}\ \langle \textit{id} \rangle\ \textbf{(}\ \langle \textit{id} \rangle\ \textbf{,} \ldots \textbf{,}\ \langle \textit{id} \rangle\ \textbf{)}\ \textbf{\{}\ldots\textbf{\}} \\
&\quad \mid \textbf{module}\ \langle \textit{id} \rangle\ \textbf{=}\ \langle \textit{module-expr} \rangle \\
&\quad \mid \langle \textit{module-expr} \rangle\ \textbf{.}\ \langle \textit{id} \rangle \mid \textbf{import}\ \langle \textit{module-expr} \rangle \\
\langle \textit{module-expr} \rangle &::= \langle \textit{id} \rangle \mid \langle \textit{module-expr} \rangle\ \textbf{.}\ \langle \textit{id} \rangle \\
&\quad \mid \langle \textit{module-expr} \rangle\ \textbf{(}\ \langle \textit{module-expr} \rangle\ \textbf{,} \ldots \textbf{,}\ \langle \textit{module-expr} \rangle\ \textbf{)}
\end{aligned}
$$

For instance, one way to define a map data type parametrised by the key type is as follows.

```
1   interface KEY {
2     type t;
3     type ord = | LT | EQ | GT;
4     let compare : t * t -> ord;
5   };
6   module Map(Key : KEY) {
7     type map(a) =
8     | Leaf
9     | Node(Key.t, a, map(a), map(a));
10
11    let empty : map(a) = Leaf;
12
13    let add(m : map(a), k : Key.t, v : a) : map(a) {
14      case m
15      | Leaf               => Node(k, v, Leaf, Leaf)
16      | Node(k2, v2, l, r) => case compare(k, k2)
17          | LT => Node(k2, v2, add(l, k, v), r)
18          | EQ => Node(k2, v,  l, r)
19          | GT => Node(k2, v2, l, add(r, k, v))
20    };
21    ...
22  };
```

First-class modules We can make the module system even more expressive by supporting *first-class modules,* i.e., adding the ability to pass modules around just like other values.

```
1   let add_two(M,x,y) {                  let add_two(M,x,y) {
```

```
    let m = M.make();
    M.add(m,x);
    M.add(m,y);
};
```

```
    import M;
    let m = make();
    add(m,x);
    add(m,y);
};
```

One can implement first-class modules by treating every module as a record containing the values of all identifiers defined within. Of course this means that referencing an element of a module now requires a memory lookup and cannot be done statically anymore (in general, the lookup can of course be optimised away in certain cases).

6 Control-Flow

6.1 Continuation passing style

In order to introduce the notion of a continuation let us take a look at the following example. Suppose we have a program that prompts the user for two inputs and then computes some result.

```
1  let f () {
2    let u = input("first: ");
3    let v = input("second: ");
4    process(u,v)
5  };
```

When we want to adapt this program to use a web-interface we face a problem with the way web-servers operate. Web-servers have the ability to call external programs to generate web-pages. But these programs are immediately terminated by the server after a web-page is produced. In our example we need three pages, two containing forms for the user to fill in the values of `u` and `v`, and one to display the computed result. As the program is terminated after each page, we have to figure out some way to pass the program state from one invocation to the next. Of course, web-sites with internal state are quite common, so web-servers do provide several mechanisms for doing so (cookies, hidden form fields, URL query string,…). What remains for us to do is to figure out, which information precisely to pass along. We need (i) a data structure storing at what place in the program we are and (ii) a way to use this data to resume the program at that point.

To resume the computation of the program from an arbitrary point we need to know

- where in the program we are, i.e., what the last evaluated expression was,
- what the result of this expression was, and
- what the values of the local variables were.

We can store this information as a function that, given the result of the last expression, continues the program from this point. Such a function is called a *continuation*. For instance, in the above example the continuation after having read the first input is

```
1  fun (u) {
2    let v = input("second: ");
3    process(u,v)
4  };
```

Continuation. A function containing the remainder of a computation.

Continuation passing style. A programming style where functions never return but call an explicitly passed continuation instead.

The continuation after the second input is

```
fun (v) {
  process(u,v)
};
```

In order to prepare our program for usage with a web-server, it is useful to translate it into a form that makes these continuations explicit. This form is call *continuation passing style,* CPS for short. In this form, every function takes an additional argument `k` that takes the continuation to be called when the function wants to return. Our example now looks as follows.

```
let f (k) {
  input("first: ",
        fun (u) {
          input("second: ",
                fun (v) {
                  process(u,v,k);
                })
        })
};
```

As a second example, let us take a look at the factorial function.

```
let fac(n) { if n == 0 then 1 else n * fac(n-1) };
```

We present two versions using continuation passing style. The first one is rather relaxed in the sense that we do not convert primitive operations.

```
let fac_cps(n,k) {
  if n == 0 then
    k(1)
  else
    fac_cps(n-1, fun (x) { k(n*x) })
};
```

If we also use continuation passing style for primitive operations like equality, subtraction, and multiplication, the code looks as follows.

```
let fac_cps(n,k) {
  equal(n,0,
    fun (c) {
      if c then
        k(1)
      else
        minus(n,1,
```

```
8             fun (a) {
9               fac_cps(a,
10                fun (b) { times(n,b,k) })
11            })
12      })
13  };
```

As we see in CPS a function never really returns, instead it calls its continuation. We can see CPS as a programming style where instead of using a call stack we manually handle return addresses by storing them in function closures, i.e., on the heap. This is of course a bit less efficient, since we removed the optimisation of using a stack, but as we will see below it offers more flexibility and allows for certain programming constructs not possible (or at least much harder to implement) with a stack discipline.

Let us conclude this section with a more involved example: a parsing function for regular expressions.

```
1   type regex =
2     | Char(char)
3     | Plus(regex, regex)
4     | Concat(regex, regex)
5     | Star(regex);
6
7   let parse_cps(str   : list(char),
8                 regex : regex,
9                 succ  : list(char) -> bool,
10                fail  : unit -> bool) : bool {
11    case regex
12    | Char(c) => if head(str) == c then
13                   succ(tail(str))
14                 else
15                   fail()
16    | Plus(r,s) =>
17        parse(str, r, succ,
18              fun () { parse(str, s, succ, fail) })
19    | Concat(r,s) =>
20        parse(str, r,
21              fun (str) { parse(str, s, succ, fail) },
22              fail)
23    | Star(r) =>
24        parse(str, r,
25              fun (str) { parse(str, Star(r), succ,
26                                fun () { succ(str) }) },
27              fun () { succ(str) })
28  };
29  let parse(str, regex) {
```

```
30      parse_cps(str, regex,
31                fun (s) { s == "" },
32                fun ()  { False })
33
34  };
```

6.2 Continuations

The problem with continuation passing style is that, in order to use it at one place in the program, we have to convert all functions used in that part to CPS. This is rather inconvenient and makes modifications of a program unnecessarily complicated. To avoid this overhead we can introduce a new construct into our language that makes the continuation at the current position available to the programmer.

⟨*expr*⟩ ::= ... | **letcc** ⟨*id*⟩ **{** ⟨*expr*⟩ **}**

The statement

letcc *k* { *expr* }

evaluates the given expression after binding the current continuation to the identifier *k*. So when calling $k(a)$, the program behaves as if *expr* returned the value *a*.

Examples

```
1   letcc k { 1 }
2   letcc k { k(1) }
3   letcc k { k(1+2) }
4   1 + letcc k { k(1) }
5   1 + letcc k { k(1+1) }
6   letcc k { (3 + k(1)) }
7   1 + letcc k { (3 + k(1)) }
```

There are two ways we can use the continuation *k*. We can call it within the expression *expr*, or we can store it somewhere and call it after the evaluation of the **letcc** statement is already finished. In the first case it acts like a return statement or an exception: we abort the evaluation of the *expr* prematurely and return the specified result. In the second case, we perform some kind of backtracking: we restart the computation following the **letcc** statement with an alternative value for *expr*. We will see several examples where this can be used to good effect.

Discussion Continuations increase the power of a language by allowing the user to define her own control-flow operations. As usual such power comes with costs.

First of all, there is a performance cost as continuations invalidate the usual stack regime for function calls. The two common ways to implement them involve either (i) replacing the call stack by a more general data structure, which is bad for cache locality, or (ii) copying parts of the stack when a continuation is created or called.

Generator.	A coroutine generating a sequence of values.

The second issue concerns program readability. A continuation can be seen as a functional analogue of a goto statement. The only difference is that with continuations we can only jump to places we have already been, while a goto also allows jumps to unvisited program locations. As with gotos, this flexibility can be misused. Many languages therefore try to replace arbitrary continuations with restricted versions, like exception mechanisms (see below).

6.3 Generators

As a first application of continuations, let us implement what is sometimes called a generator. For-loops in our language are rather restricted as we can only iterate over the numbers between two given bounds. Many language designers tried to generalise loops to an imperative analogue of a fold function. For instance, there are languages where for-loops can also iterate over the elements of container types like arrays and lists. Instead of having built in support for a handful of such types, recent languages like Python found a way to allow the user to define her own iterators for for-loops. Such a definition is called a *generator*. It is a function that produces, one after the other, all the values the loop should iterate over. The question is how we can pass these values to the loop construct while at the same time remembering where to continue for the next iteration. Using the return value of the function is cumbersome since we can return only one value at a time. So these languages introduced a new language construct **`yield`** that stops the evaluation of the current function, returns a value to the caller, and allows the caller to later resume the function at the point it was interrupted. For instance,

```
let gen() {
  let n = 0;
  while True {
    yield n;
    n := n+1;
  }
};
```

generates the sequence `0,1,2,3,4,...`. Similarly,

```
let gen(lst) {
  while is_cons(lst) {
    yield head(lst);
    lst := tail(lst);
  }
};
```

generates the sequence of all elements in the given list.

Looking at the definition of **`yield`**, we see that it looks a lot like a continuation, and we can in fact use continuations for the implementation.

Exception. A control-flow mechanism for returning out of nested function calls.

```
1  let gen_return(x) { () };
2
3  let gen_helper() {
4    let n = 0;
5    while True {
6      letcc k {
7        gen_helper := k;
8        gen_return(n)
9      };
10     n := n+1;
11   };
12   0
13 };
14
15 let gen() {
16   letcc k {
17     gen_return := k;
18     gen_helper()
19   }
20 };
```

6.4 Exceptions

Continuations can also be used to good effect for error handling. The problem with error handling is that, when an error occurs, we need to abort the current computation and go to an outer context where we can sensibly react to the failure. If we are using side effects, we also have to do the required clean up work required by the aborted computation. In the traditional way of error handling, error conditions are communicated via the return value of functions. This has the disadvantage that we have to surround every function call by an if-statement to test for the occurrence of an error, which is quite cumbersome, error prone (easy to forget), and clutters the code. Therefore programming languages have introduce a mechanism making the error checking implicit. This *exception* mechanism works similarly to the break-statement of imperative languages. But instead of jumping out of loops, i.e., out of a nested static scope, it allows the program to jump out of nested function calls.

$$\begin{aligned}\langle expr\rangle \ ::= \ & \ldots \mid \textbf{try}\ \langle expr\rangle\ \textbf{catch}\ \langle var\rangle\ \textbf{=>}\ \langle expr\rangle \\ & \mid \textbf{throw}\ \langle expr\rangle\end{aligned}$$

```
1  try 2                          try 2 + throw 4
2  catch x => x + 1               catch x => x + 1
3  => 2                           => 5
```

```
type error = | EmptyList;

let head(lst) {
  case lst
  | []     => throw EmptyList
  | [x|xs] => x
};

try head([])
catch x => 0
```

```
type error   = | NotFound;
type key_val = [ key : a, val : b ];

let lookup(lst : list(key_val), k : a) : b {
  case lst
  | []     => throw NotFound
  | [x|xs] => if x.key == k then
                x.val
              else
                lookup(xs, k)
};
```

Exceptions can be implemented using continuations. Every function gets as an additional argument a continuation to call when raising an exception. A **catch** statement uses **letcc** to create such a continuation.

```
try e catch x => handler    ⟹    letcc k { e(fun (x) { k(handler) }) }
throw e k                   ⟹    k(e)
```

Exercise Implement exceptions using this translation.

Exceptions are not without problems. Although they can be considered as a generalised break-statement, the destination of an exception is determined dynamically by the sequence of function calls and not statically by the syntactic structure of the program. This makes reasoning about exceptions non-local.

In a purely functional program, the main kind of bug caused by improper use of exceptions is forgetting to catch them. This can lead to program termination and thus to possible data loss and reduced availability. When programming with side effects it is much harder to get code with exceptions right. (There are whole books written about how to write exception-save C++ code. In Python it is actually *impossible* to write exception-save code as every statement can throw exceptions.) The problem is that, in the presence of side effects, it is essential to know which function calls can throw exceptions, since we might need to perform some cleanup tasks if an exception occurs. Some languages therefore require programmers to annotate functions with a list of all exceptions they can throw. In practice, this has not turned out to be very successful, as many programmers consider this tedious and simply specify that there are no restrictions on the exceptions

a function can throw. (Java and most other object oriented languages make this very easy as there usually is some general `Exception` class that a programmer can use to circumvent the need for precise exception specifications.)

For these reasons it is better to use exceptions sparingly. They should not be used as the main mechanism for error handling. Instead, it is better to think of exceptions as a control flow mechanism. An exception is an efficient way to return from a deeply nested function call that can be used to avoid having to unwind the stack one function at a time and to check the return value for an error each time. Code that uses the return value for error reporting is usually much easier to understand. Such a coding style is therefore recommended in cases where function calls are not nested that deeply and the cost associated with checking the return value is thus not that high.

A programming style based on returning error codes is not without its own problems though.

(1) It is easy to forget checking for errors, if a function does not use a variant type that distinguishes between regular return values and error codes, but uses some 'magic values' to indicate errors. For instance, in old C code it is usual for a function returning an integer to use negatives values as error codes and other values for regular returns. Similarly, for functions returning pointers it is common to indicate failures by returning `null`.

(2) If a language does not support variant types or functions with multiple return values, it is cumbersome to use return values both for error reporting and for returning the regular result.

(3) Return values containing error codes can seriously degrade performance. One needs to pass more data to and from functions and after every function call one needs additional code to check for errors. Besides the runtime cost this also slows down the compiler as it has to deal with this additional code. On the other hand, throwing exceptions also has associated costs: for C++, a recent report names a cost of 8000 to 20000 CPU cycles (depending on which compiler is used) for throwing an exception.

6.5 Algebraic effects

Algebraic effects are a generalisation of exceptions that can be used to extend the language by user-defined control-flow constructs. As with exceptions, algebraic effects consist of two parts: the execution of an effect and its handling.

⟨*expr*⟩ ::= ... | **effect** ⟨*name*⟩ **:** ⟨*type*⟩ **;**
| **try** ⟨*expr*⟩ **catch** ⟨*effect*⟩ **=>** ⟨*expr*⟩
| ⟨*effect*⟩ ⟨*arguments*⟩
| **abort** ⟨*arguments*⟩
| **resume** ⟨*arguments*⟩

We can define a new effect using the **effect** statement.

```
1  effect break    : unit;
2  effect continue : unit;
3  effect throw    : a -> b;
```

After such a definition the name of an effect can be used in an expression to call the corresponding handler. Handlers work just like exception handlers except for the fact that, in addition to aborting the expression containing the effect, they can also resume it at the point after the effect. To do so an effect handler can use the statements **abort** and **resume**. For instance,

```
effect bar : int;

try 3 + bar catch bar => 1 + abort 5
```

works just like an exception where the handler returns the value 5, i.e.,

```
try 3 + throw bar catch x => 5
```

But

```
effect bar : int;

try 3 + bar catch bar => resume 5
```

resumes the expression `3 + bar` by assuming that the value of `bar` is `5`. Thus this code is equivalent to

```
3 + 5
```

Hence, one can think of **resume** `5` as calling the continuation of `bar` in `3 + bar` with the argument `5`. Similarly, **abort** `5` is a call to the continuation of **try** `...` **catch** `bar => ...` with argument `5`. Instead of calling these continuations, the handler also has to ability to return them or to store them in some data structure.

Let's take a look at a few examples. We start by implementing **break** and **continue** statements as algebraic effects.

```
                                  effect break    : unit;
                                  effect continue : unit;

for x in ... {                    try
  ...                               for x in ... {
  if ... then break;                  try
  ...                       ===>        ...
  if ... then continue;                 if ... then break;
  ...                                   ...
}                                       if ... then continue;
                                        ...
                                    catch continue => abort ()
                                  }
                                  catch break => abort ()
```

Similarly, exceptions can be defined using algebraic effects.

```
                                  effect throw : a -> b;

```

```
try                                    try
  ...                          ===>      ...
  throw 4                                throw 4
  ...                                    ...
catch x => ... x ...                   catch throw(x) => abort (... x ...)
```

As usual, the more powerful a language feature the easier it is to misuse. In particular our remarks on the issues associated with exceptions also apply to algebraic effects.

7 Constraints

The idea of declarative programming is to just describe *what* you want to compute, but not *how* to compute it. It is then the responsibility of the compiler to figure this out on its own. In particular this means that declarative programs do not use side-effects or state. The advantage is that each part of such a program can be understood on its own. Furthermore, the various components of declarative programs can be written separately and then combined. This makes programs written in this style usually easier to understand and reason about.

There are two main declarative programming paradigms: functional programming and logical programming. We have already seen the former in Chapter 2. Here, we will focus on the latter. In logic programming one tries to formulate the problem as a set of constraints and then use a built in constraint solver to search for a solution.

7.1 Single-assignment variables

To support logic programming, we have to extend our language with two new constructs. The first is the concept of a *single-assignment variable*. Such variables may start uninitialised, but once a value is assigned it cannot be changed anymore. The only change in syntax to the previous chapter is that we allow to omit the value from let-bindings to introduced uninitialised variables.

⟨*expr*⟩ ::= ... | **let** ⟨*id*⟩ **;** ⟨*expr*⟩

But the semantics change. Assigning a value to a variable is only allowed if the variable is either uninitialised, or it already has the same value we are assigning.

```
1  let x;
2  let y;
3  x := 1;
4  x := 1; // ok
5  x := 2; // error
6  y := x+1;
```

Furthermore, parameter-passing is now by reference.

```
1  let add(x,y,z) {
2    z := x+y;
3  };
4  let u;
5  add(1,2,u);
```

Single-assignment variable. A variable that can we written to only once.

There are several choices of what should happen if we read from an uninitialised variable. In a concurrent program, the most sensible thing to do is to block until some other fibre initialises the variable. Without concurrency the only possibility is to abort the program with an error message.

```
let reverse(lst, ret) {
  case lst
  | []     => ret := []
  | [x|xs] => {
      let rev;
      ret := [x|rev];
      reverse(xs, rev);
    }
};
```

Note that we made the function `reverse` tail recursive by putting the assignment to `ret` before the recursive call.

We can also use uninitialised variables in data structures. For instance, to implement lists that allow adding elements at the end.

```
let make() {
  let empty;
  Pair(empty, empty)
};
let add_first(lst, x, ret) {
  case lst
  | Pair(first, last) =>
      ret := Pair([x|first], last)
};
let add_tail(lst, x, ret) {
  case lst
  | Pair(first, last) =>
    ( let empty;
      last := [x|empty];
      ret  := Pair(first, empty) )
};
```

We can also use single-assignment variables to create cyclic data structures.

```
x := [1,2,3|x]
```

7.2 Unification

An assignment statement `x := e` is asymmetric as we can only use l-values on the left-hand side, while arbitrary r-values are allowed on the right-hand side. When using single-assignment variables, we can define a symmetric version of the assignment statement which is called *unification*.

Unification. Solving an equation between terms.

$\langle expr \rangle$ ::= ... | $\langle expr \rangle$ `:=:` $\langle expr \rangle$

When unifying two values u and v, we try to assign values to all undefined variables in u and v in such a way that the resulting values become equal. Hence, a unification `u :=: v` can be seen as solving the equation $u = v$ by substituting values for the variables.

```
1  1 :=: x            x := 1
2  x :=: y            identifies x and y
3  [x,2] :=: [1,y]    x := 1 and y := 2
```

Implementation We can solve equations of the form u :=: v recursively as follows.

- If u is an uninitialised variable, we set it to v.
- If v is an uninitialised variable, we set it to u.
- If $u = m$ and $v = n$ are both numbers, we check that $m = n$. If this is not the case, unification fails.
- If $u = c(s_0, \ldots, s_{m-1})$ and $v = d(t_0, \ldots, t_{n-1})$ are both constructors, we check that $c = d$, $m = n$, and s_i :=: t_i, for all i.
- If $u = [l_0 = s_0, \ldots, l_{m-1} = s_{m-1}]$ and $v = [k_0 = t_0, \ldots, k_{n-1} = t_{n-1}]$ are both records, we check that there is some bijection $\varphi : m \to n$ such that $l_i = k_{\varphi(i)}$ and s_i :=: $t_{\varphi(i)}$, for all i.
- In all other cases, unification fails.

(Note in particular that we cannot unify function values.) There are a few things one has to keep in mind when implementing this procedure.

(1) We have to distinguish two kinds of uninitialised values. If we have just introduced an uninitialised variable x, we know nothing at all about its value. After a unification with another uninitialised variable y, we still do not know the value of x, but we already know that it is equal to that of y. So we need to distinguish between values for completely undefined variables and values for variables that are equal to other variables.

(2) A naïve recursive implementation of unification can go into an infinite loop if we unify cyclic data structures. For instance, the last unification in

```
1  let x; let y;
2  x :=: [1,x];
3  y :=: [1,1,y];
4  x :=: y;
```

might not terminate. To prevent this, we need to remember during unification which equations we have already checked. If we try to check an equation `u :=: v` for the second time, we do not need to recursively call the unification procedure, we can simply skip it and assume that it was successful.

Backtracking. Reverting previous choices.

7.3 Backtracking

What do we do if we use single-assignment variables and discover that we have assigned them the wrong value? *Backtracking* is a mechanism for reverting such choices by reverting the whole program to an earlier state. To implement it we add a nondeterministic choice operator to our language.

⟨*expr*⟩ ::= ... | **choose|** ⟨*expr*⟩ ... **|** ⟨*expr*⟩ | **fail**

Abstractly, a choice operator selects one of the given expressions that does not cause a failure and executes it. The actual implementation of course does not know which of the expressions will fail. What the operator therefore does is to create a kind of checkpoint and then executes the first expression. If, later on, a failure occurs, the program state saved at the last checkpoint is restored and the next alternative is tried instead. If all alternatives fail, the checkpoint is deleted again and the choose-statement itself fails. This means that only this last alternative executed (the one that succeeded) will have an effect on the program, those that have failed will not. It is if they never were executed. Not that the failure does not need to occur inside the expressions themselves, it may happen later on in the program.

```
1  choose
2  | { x := 1; y := 1; }
3  | { x := 2; y := 2; }
```

tries first to set two variables to 1. If one of them already has a different value, the corresponding assignment fails and we try to set the variables to 2. If this fails as well, the whole choose-statement fails. In this case, none of the variables is modified. Note that a transaction can only undo memory changes, not other kinds of side-effects. So

```
1  choose
2  | { print "start..." 1; fail; print "stop..." 1; }
3  | print "start..." 2
```

will print out `"start... 1start... 2"` even though the first expression is aborted.

How do we implement the choose construct? We use two primitive operations `checkpoint k` and `rewind`. The first one takes a continuation as argument and creates a checkpoint storing the current machine state. If later on a `rewind` command is executed, we

- fetch the continuation associated with the last checkpoint,
- restore the machine state to its previous state (which deletes the last checkpoint),
- and call the fetched continuation.

Using these two operations we can translate a choose statement as follows.

```
1  choose | e1                  ⟹   e1
2  choose | e1 | e2 ... | en    ⟹   letcc k {
```

```
3                                        checkpoint
4                                          fun () { k(choose | e2 ... | en) };
5                                        e1
6                                      }
7   fail                        ⟹     rewind
```

Of course, now we have to figure out how to implement `checkpoint` and `rewind`. Saving the whole machine state is very inefficient. What we will do instead is to record all memory changes and undo them when we rewind. As we only use single-assignment variables the only changes we need to undo are assignments of values to uninitialised variables. This can be done by simply marking those variables as uninitialised again. Hence, what we need to do is to store a list of all variables whose value has changed since the last checkpoint. Then rewind can traverse the list and undo those changes again.

This means our implementation looks as follows. We maintain a stack of checkpoints. Each entry of the stack contains a stored continuation and the list of variables whose value has changed. A `checkpoint k` command simply pushes a new entry on the stack consisting of `k` and the empty list. Each variable assignment now has to add the variable in question to the list in the top stack entry. Finally, a `rewind` command, retrieves the continuation from the top stack entry, walks the list of variables to mark them as uninitialised again, and then calls the retrieved continuation.

With single-assignment variables and backtracking, we can translate most Prolog programs (which do not use advanced features) into our kernel language. For instance,

```
edge(a,b).
edge(b,c).
trans(X,Y) :- edge(X,Y).
trans(X,Y) :- edge(X,Z), trans(Z,Y).
```

turns into

```
let edge(x,y) {
  choose
  | { x := a; y := b; }
  | { x := b; y := c; }
}
let trans(x,y) {
  choose
  | edge(x,y)
  | { let z; edge(x,z); trans(z,y); }
}
```

7.4 Programming examples

Let us write our standard list processing examples using single-assignment variables.

```
let nth(lst,n,z) {
  choose
```

```
    | { let t;
        n :=: 0;
        lst :=: [z|t] }
    | { let h; let t;
        lst :=: [h|t];
        nth(t,n-1,z) }
  };
  let length(lst, n) {
    choose
    | { lst :=: []; n :=: 0 }
    | { let h; let t; let m;
        lst :=: [h|t];
        length(t, m);
        n :=: m+1 }
  };
  let sum(lst, n) {
    choose
    | { lst :=: []; n :=: 0 }
    | { let h; let t; let m;
        lst :=: [h|t];
        sum(t,m);
        n :=: m+h; }
  };
  let map(f, lst, z) {
    choose
    | { lst :=: []; z :=: [] }
    | { let h; let t; let y;
        lst :=: [h|t];
        z :=: [f(h)|y];
        map(f, t, y); }
  };
  let fold(f, acc, lst, z) {
    choose
    | { lst :=: []; z :=: acc }
    | { let h; let t;
        lst :=: [h|t];
        fold(f, f(acc, h), t, z) }
  };
  let foldr(f, acc, lst, z) {
    choose
    | { lst :=: []; z :=: acc }
    | { let h; let t; let y;
        lst :=: [h|t];
        foldr(f, acc, t, y);
```

```
47          z :=: f(h, y) }
48    };
49    let append(x,y,z) {
50      choose
51      | { x :=: []; y :=: z }
52      | { let h; let t; let r;
53          x :=: [h|t];
54          z :=: [h|r];
55          append(t,y,r); }
56    };
57    let reverse(lst, z) {
58      let iter(lst, y, z) {
59        choose
60        | { lst :=: []; z :=: y }
61        | { let h; let t;
62            lst :=: [h|t]; iter(t, [h|y], z) }
63      };
64      iter(lst, [], z)
65    };
```

If we use a more Prolog-like syntax, the code becomes extremely clean.

```
1   nth([x|xs], 0, x).
2   nth([x|xs], i, y) :- nth(xs, i-1, y).
3
4   length([],     0).
5   length([x|xs], n) :- length(xs, n-1).
6
7   sum([],     0).
8   sum([x|xs], n) :- sum(xs, n-s).
9
10  map(f, [],     []).
11  map(f, [x|xs], [f(x)|ys]) :- map(f, xs, ys);
12
13  fold(f, acc, [],     acc).
14  fold(f, acc, [x|xs], z)  :- foldr(f, f(acc, x), xs, z).
15
16  foldr(f, acc, [],     acc).
17  foldr(f, acc, [x|xs], f(acc, z)) :- foldr(f, acc, xs, z).
18
19  append([], y, y).
20  append([x|xs], y, [x|z]) :- append(xs, y, z).
21
22  reverse(lst, rev) :- reverse_helper(lst, [], rev).
23
```

```
reverse_helper([],     z, z).
reverse_helper([x|xs], y, z) :- reverse(xs, [x|y], z).
```

We can use lists terminated by an unbound variable to efficiently implement queues.

```
type queue = | Queue(int,list(a),list(a));

let empty () {
  let t;
  Queue(0, t, t);
};
let is_empty(queue) {
  case queue
  | Queue(n,q,t) => n == 0
};
let insert(queue,x) {
  case queue
  | Queue(n,q,t) => { let s; t := [x|s]; Queue(n+1,q,s) }
};
let first(queue) {
  case check(queue)
  | Queue(n,q,t) => head(q)
};
let remove(queue) {
  case check(queue)
  | Queue(n,q,t) => Queue(n-1, tail(q), t)
};
```

8 Objects

Object-oriented programming has created quite a hype after it became mainstream with the release of the first C++ compilers. It was seen as a panacea for all kinds of program design issues, mainly because of its clear advantages over the other mainstream languages of the time, most notably C. Fortunately, this hype is slowly fading over the last years, so a rational discussion of object-oriented programming is now possible.

Unfortunately, there is no standard definition of object-orientation as everybody uses his or her own version. The initial idea was to make the global state of a program more manageable by breaking it into smaller parts called *objects*. Each of these objects has its own local state which is not accessible to the outside. To communicate objects can pass *messages* between them. Thus, as a slogan we could say that object-orientation combines *encapsulated state* plus *message passing*.

At least that was the initial idea. Over time the meaning has changed slightly. Nowadays when introducing object-oriented programming one usually mentions as a key idea the concept of combining data and functions operating on it into a single data structure. According to this newer definition, an object is simply a record containing both functions and non-function values. In addition one usually considers a certain set of additional languages features (such as inheritance) to be an essential part of the definition. Which exactly depends on the person.

Still, the original definition is quite useful as it tells us *how to use* object-orientation, whereas the newer one simply tells us *how it is implemented*. In the following sections we will present several language features that can be used to implement object-oriented programming, or to make it more useful. In particular, we will consider

- dynamic dispatch,
- subtyping,
- encapsulated state,
- inheritance,
- open recursion.

8.1 Dynamic dispatch

We will implement the features of object-oriented programming step by step starting with dynamic dispatch. If we want to send a certain message to an object, we do not know statically which function to call. Therefore, we have to store the function with the object and look it up at

Dynamic dispatch.	Method calls are resolved at runtime, not at compile-time.

runtime. An easy way to do so is to represent the object as a record of functions, one for each message. For instance, a list object supporting the messages

```
next   : unit -> object
get    : unit -> int
iter   : (int -> unit) -> unit
length : unit -> int
```

would be represented by a record of type

```
[ next   : unit -> object,
  get    : unit -> int,
  iter   : (int -> unit) -> unit,
  length : unit -> int ];
```

To send a message, we just call the corresponding function.

```
let new_empty() {
  let n =
    [ next   = fun ()  { n },
      get    = fun ()  { error },
      iter   = fun (f) { () },
      length = fun()   { 0 } ];
  n
};
let new_node(val, next) {
  [ next   = fun ()  { next },
    get    = fun ()  { val },
    iter   = fun (f) { f(val); next.iter(f) },
    length = fun ()  { 1 + next.length() } ]
};

let n1 = new_empty();
let n2 = new_node(1,n1);
let n3 = new_node(2,n2);
n3.iter(fun (x) { print "value is " x });
```

This direct encoding via records quickly becomes cumbersome, but the right kind of syntactic sugar makes it usable.

```
object { m_1 : t_1 ; ... ; m_k : t_k ; }
⟹ [ m_1 : t_1 , ... , m_k : t_k ]

object {
  m_1 ( a_1 ) { b_1 } ;
  ...
  m_k ( a_k ) { b_k } ;
}
```

```
⟹
[ m₁ = fun ( a₁ ) { b₁ },
  ...
  mₖ = fun ( aₖ ) { bₖ } ]
```

With this notation we can write the above code as

```
type olist =
  object {
    next   : unit -> olist;
    get    : unit -> int;
    iter   : (int -> unit) -> unit;
    length : unit -> int
  };

let new_empty() {
  let n =
    object {
      next()   { n };
      get()    { error };
      iter(f)  { () };
      length() { 0 };
    };
  n
};
let new_node(val, next) {
  object {
    next()   { next };
    get()    { val };
    iter(f)  { f(val); next.iter(f) };
    length() { 1 + next.length() };
  }
};

let n1 = new_empty();
let n2 = new_node(1,n1);
let n3 = new_node(2,n2);
n3.iter(fun (x) { print "value is " x });
```

Objects that are implemented as records of functions as above are sometimes called *functional objects* since they have no internal state. Such functional objects can be regarded as a generalisation of a function that has multiple entry points.

Note that the above implementation is based on structural type equivalence. Two objects (like `empty` and `node` above) have the same type if they support the same set of methods. Most object-oriented languages use name equivalence instead and would consider `empty` and `node` to have different types. In languages such as C++ there is nothing corresponding to object types like the

type `olist` in the above example. But many of the modern object-oriented languages that are based on name equivalence have added such types as an additional concept. For instance, in Java they are called *interfaces*.

Example Our running example in this chapter will be a class hierarchy for geometric shapes. This is still simple enough to (mostly) fit on a single page but shares many properties with the more complicated hierarchies one finds in real-world programs, like class hierarchies for graphical user interfaces.

```
1  type shape = object {
2    draw       : unit -> unit;
3    move       : int -> int -> shape;
4    dimensions : unit -> [ min_x : int, min_y : int,
5                           max_x : int, max_y : int ];
6  };
7
8  let new_point(x : int, y : int) : shape =
9    object {
10     draw()       { draw_point(x,y)           };
11     move(dx, dy) { new_point(x + dx, y + dy) };
12     dimensions() { [ min_x = x, min_y = y, max_x = x, max_y = y ] };
13   };
14
15 let new_circle(x : int, y : int, r : int) : shape =
16   object {
17     draw()       { draw_circle(x,y,r)            };
18     move(dx, dy) { new_circle(x + dx, y + dy, r) };
19     dimensions() { [ min_x = x - r, min_y = y - r,
20                      max_x = x + r, max_y = y + r ] };
21   };
22
23 let new_rectangle(x : int, y : int, w : int, h : int) : shape =
24   object {
25     draw()       { draw_rectangle(x,y,w,h)             };
26     move(dx, dy) { new_rectangle(x + dx, y + dy, w, h) };
27     dimensions() { [ min_x = x,     min_y = y,
28                      max_x = x + w, max_y = y + h ] };
29   };
30
31 let new_group(shapes : list(shape)) {
32   object {
33     draw()       { iter(fun (s) { s.draw() }, shapes) };
34     move(dx, dy) { new_group(map( fun (s) { s.move(dx, dy) }, shapes)) };
35     dimensions() {
```

```
36        case shapes
37        | []     => [ x = 0, y = 0, w = 0, h = 0 ]
38        | [s|ss] => let d = s.dimensions();
39                    fold(fun(d,s) {
40                           let d2 = s.dimensions();
41                           [ min_x = min(d.min_x, d2.min_x),
42                             min_y = min(d.min_y, d2.min_y),
43                             max_x = max(d.max_x, d2.max_x),
44                             max_y = max(d.max_y, d2.max_y) ]
45                         },
46                         s.dimensions(),
47                         ss)
48      };
49    };
50  };
```

Multi-methods One problem with dynamic dispatch as defined above is that it is asymmetric with respect to its arguments. The object we dispatch on is treated differently than the other arguments. Some languages have introduced the possibility to dispatch on the types of all arguments. This is called *multi-methods*.

```
1  let intersect(x : circle,    y : circle)    : shape { ... }
2  let intersect(x : circle,    y : rectangle) : shape { ... }
3  let intersect(x : rectangle, y : circle)    : shape { ... }
4  let intersect(x : rectangle, y : rectangle) : shape { ... }
```

The problem with multi-methods is that, as the number of classes grows, defining functions for all combinations quickly becomes unmanageable. While there are languages that support multi-methods, the approach has never really become popular.

Type classes An alternative approach to dynamic dispatch is provided by Haskell's *type classes*. A type class consists of a collection of functions types associated with one or more parameter types. For each choice of parameter types, we can defined an *instance* of the type class by providing an implementation of the required functions.

```
1  typeclass Shape(a) {
2    draw       : a -> unit;
3    move       : a -> int -> int -> a;
4    dimensions : a -> [ min_x : int, min_y : int,
5                        max_x : int, max_y : int ];
6  };
7
8  type point = Point(int,int);
9
10 instance Shape(point) {
```

```
    draw(Point(x,y))         { draw_point(x,y)       };
    move(Point(x,y), dx, dy) { Point(x + dx, y + dy) };
    dimensions(Point(x,y))   { [ min_x = x, min_y = y,
                                 max_x = x, max_y = y ] };
  };

  type circle = Circle(int,int,int);

  instance Shape(circle) {
    draw(Circle(x,y,r))         { draw_circle(x,y,r)        };
    move(Circle(x,y,r), dx, dy) { Circle(x + dx, y + dy, r) };
    dimensions(Circle(x,y,r))   { [ min_x = x-r, min_y = y-r,
                                    max_x = x+r, max_y = y+r ] };
  };
```

Type classes are a controlled form of ad-hoc polymorphism. When comparing them with general overloading we notice the following advantages and disadvantages.

- Type classes are more restrictive as every instance needs to fit to the given parametric type. E.g. it is not possible to have instances with a different number of arguments.
- Type classes provide a mental framework that prevents overloading to cause too much chaos. For instance, in every instance of the `Eq` typeclass the operator == should check for equality, and not do something completely different.
- When using type classes one can type check code without having to know all the instances. With overloading on the other hand, all instances need to be known before one is able to check that a function call is well-typed.

Comparison with variant types There is an alternative solution based on variant types instead of objects. We could have defined

```
type shape =
  | Point(int,int)
  | Circle(int,int,int)
  | Rectangle(int,int,int,int)
  | Group(list(shape));

let draw(sh) {
  case sh
  | Point(x,y)         => draw_point(x,y)
  | Circle(x,y,r)      => draw_circle(x,y,r)
  | Rectangle(x,y,w,h) => draw_rectangle(x,y,w,h)
  | Group(shapes)      => iter(draw, shapes)
};
let move(sh, dx, dy) {
  case sh
```

Subtype. *s* is a subtype of *t* if values of type *s* can be used everywhere a value of type *t* is expected.

```
16      | Point(x,y)          => Point(x + dx, y + dy)
17      | Circle(x,y,r)       => Circle(x + dx, y + dy, r)
18      | Rectangle(x,y,w,h) => Rectangle(x + dy, y + dy, w, h)
19      | Group(shapes)       => Group(map(fun (s) { move(s, dx, dy) }, shapes))
20    };
21    let dimensions(sh) {
22      case sh
23      | Point(x,y)          => ...
24      | Circle(x,y,r)       => ...
25      | Rectangle(x,y,w,h) => ...
26      | Group(shapes)       => ...
27    };
```

The only difference is the way we have grouped the code. In the object-based solution we collect all code pertaining to a given shape in one place, whereas when using variant types we collect all code pertaining to a given operation on shapes in one place. The difference becomes noticeable if we want to extend the program. If we add a new shape, say a triangle, the object-based approach is more convenient, we only need to define a new class. In the variant-type-based solution we have to modify every operation to add a new case. If, on the other hand, we add a new operation, like rotation, then the solution using variant types is more convenient. In the object-based approach we have to modify every class definition.

8.2 Subtyping

A type *s* is a *subtype* of a type *t* if values of type *s* can be used everywhere a value of type *t* is expected. This means that *s* is more specialised than *t*, or *t* more general than *s*. We write $s \leq: t$ to denote this fact. As with type equivalence there are two different approaches to implement subtyping: *structural* and by *name*. In languages like Java where subtyping is defined by name, the programmer has to explicitly declare if one object type is to be a subtype of another. In languages with structural subtyping on the other hand, a type *s* is automatically a subtype of all types that are more general than *s*. This means that, if *s* and *t* both are object types, then *s* will be a subtype of *t* if *s* supports all the methods of *t*. For instance, if we have defined a class of shapes with methods `draw`, `move`, and `box` and a subclass of rectangles with an additional method `area`, then the rectangle class is a subtype of the shape class.

Programming languages have a certain choice in how exactly to define the subtyping relation. Let us discuss the possibilities for some of the usual types. It does make a difference whether we have *immutable* or *mutable* values. We start with the case where all values are immutable. It is possible to already define subtyping relations between basic types. For instance, we could have

`int16` $\leq:$ `int32` $\leq:$ `int64` or `uint16` $\leq:$ `int32`

Contravariant. The order is reversed.

Covariant. The order is preserved.

What about

`uint32` $\leq:$ `int32` or `int32` $\leq:$ `float32` ?

For records we have

$$[l_1 : s_1, \ldots, l_m : s_m, k_1 : t_1, \ldots, k_n : t_n] \leq: [l_1 : u_1, \ldots, l_m : u_m]$$

if $s_i \leq: u_i$ for all i, that is, if every label appearing in the second record is also present in the first one with a subtype of the corresponding type on the right-hand side.

Example

```
type shape     = [ x : int, y : int ];
type circle    = [ x : int, y : int, r : int ];
type rectangle = [ x : int, y : int, w : int, h : int ];

circle <: shape    and    rectangle <: shape
```

Exercise What is the subtype ordering for variant types?

What about functions? Suppose we have a function of type $f : a \to b$. When can we use it at a place where a function of type $c \to d$ is expected? f will get passed a value of type c (so $c <: a$) and it will return a value of type b where one of type d is expected (so $b <: d$).

```
let g(f : c -> d) = {
  ...
  let x : c = ...;
  let y : d = f(x);
  ...
};
```

This means that

$$a \to b <: c \to d \quad \text{iff} \quad c <: a \quad \text{and} \quad b <: d\,.$$

Note that the orders for the parameter and the return value are different. We say that function types are *contravariant* (the order is reversed) in the parameter position and *covariant* (the order is the same) in the result position. In general a type constructor is contravariant in all types used as inputs and covariant in all types used as outputs. If an argument type is used both as input and output, the constructor is *invariant*.

Invariant. No order relation for different parameters.

Example

```
type shape     = [ x : int, y : int ];
type circle    = [ x : int, y : int, r : int ];
type rectangle = [ x : int, y : int, w : int, h : int ];

shape -> circle <: rectangle -> circle <: rectangle -> shape
```

Example The most important example of invariant constructors is the case of mutable data structures.

```
type box(a) = [ data : a ];

let get(box : box(a)) : a {
  box.data
};
let set(box : box(a), x : a) : unit {
  box.data := x
};
```

When is `box(a) <: box(b)`? Suppose that `box(a) <: box(b)`. Then applying `get : box(b) -> b` to a value of type `box(a)`, we need to get a value of some subtype of `b`. Hence, we must have `a <: b`. Furthermore, if we call `set : box(b) -> b -> unit` with a box of type `box(a)` and an element of type `b`, and then apply `get : box(a) -> a` to that box, we need to get an element of type `a`. Hence, we also must have `b <: a`.

```
x : a,  y : b,  u : box(a)
...
// conversion a -> b
set(u,x);      // set : box(a) -> a -> unit
y := get(u);   // get : box(b) -> b
...
// conversion b -> a
set(u,y);      // set : box(b) -> b -> unit
x := get(u);   // get : box(a) -> a
```

Subtyping for objects Many languages define simpler subtyping relations than the most general one we have described above. In particular, when determining whether some class is a subclass of another one, object-oriented languages frequently require the types of methods to match exactly instead of one being just a subtype of the other one. This makes type checking faster and, more importantly, the type system less complex.

In fact, this form of subtyping is simple enough that it can be emulated by a certain variant of parametric polymorphism called *row polymorphism*. The idea is to allow parameters in record (and object) types of the form

$$[l_0 : t_0, \ldots, l_{n-1} : t_{n-1}, \alpha]$$

which can be instantiated with further label declarations. For instance, instantiating the α in the above record type with the value $k_0 : s_0, k_1 : s_1, \beta$ yields the record type

$$[l_0 : t_0, \ldots, l_{n-1} : t_{n-1}, k_0 : s_0, k_1 : s_1, \beta]$$

Then we have a subtyping relation

$$[l_0 : t_0, \ldots, \alpha] <: [k_0 : s_0, \ldots, \beta]$$

between two such types if we can obtain the later by a suitable instantiation of the parameter α in the former. Hence, we can simulate object types with subtyping by identifying an object type

$$\texttt{object}\ m_0 : t_0, \ldots, m_{n-1} : t_{n-1}\ \texttt{end}$$

with the record type

$$[m_0 : t_0, \ldots, m_{n-1} : t_{n-1}, \alpha]$$

In this context of subtyping for objects let us also mention the language Eiffel, where the definition allows subtypes when comparing methods. But the designer of Eiffel consciously chose to define subtyping for functions to be covariant in *both* types. This leads to an unsound type system since the programmer is allowed to pass arguments of unsupported types to a function (in which case Eiffel generates a runtime-exception). The reason for this decision was that it was felt that contravariance was too confusing for the programmer. But it is questionable whether this solution is any less confusing.

Let us conclude this section with an example showing the advantages of subtyping. One area where it can be superior to parametric polymorphism is one wants to use *heterogeneous data structures*. For instance, using subtyping it is possible to have a list containing both circles and rectangles, whereas when using parametric polymorphism we have to decide which of the two kinds of objects we want to put into the list.

8.3 Encapsulated state

We have shown above how to implement purely functional objects. Now it is time to add mutable state. We can do so by simply combining dynamic dispatch with side-effects.

```
1  type account = object {
2    deposit  : int -> unit;
3    withdraw : int -> unit;
4  };
```

```
let new_account(balance) {
  object {
    deposit(amount)  { balance := balance + amount };
    withdraw(amount) { balance := balance - amount };
  }
};
```

As are more involved example let us give a version of the shape class with internal state.

```
type shape = object {
  draw        : unit -> unit;
  move        : int -> int -> unit;
  dimensions  : unit -> [ min_x : int, min_y : int, max_x : int, max_y : int ];
};

let new_point(x : int, y : int) : shape {
  object {
    draw()       { draw_point(x,y)           };
    move(dx, dy) { x := x + dx; y := y + dy; };
    dimensions() { [ min_x = x, min_y = y, max_x = x, max_y = y ] };
  }
};

let new_circle(x : int, y : int, r : int) : shape {
  object {
    draw()       { draw_circle(x,y,r)        };
    move(dx, dy) { x := x + dx; y := y + dy; };
    dimensions() { [ min_x = x - r, min_y = y - r,
                     max_x = x + r, max_y = y + r ] };
  }
};

let new_rectangle(x : int, y : int, w : int, h : int) : shape {
  object {
    draw()       { draw_rectangle(x,y,w,h)   };
    move(dx, dy) { x := x + dx; y := y + dy; };
    dimensions() { [ min_x = x,     min_y = y,
                     max_x = x + w, max_y = y + h ] };
  }
};

let new_group(shapes : list(shape)) {
  object {
    draw()       { iter(fun (s) { s.draw()       }, shapes) };
    move(dx, dy) { iter(fun (s) { s.move(dx, dy) }, shapes) };
```

Inheritance. A mechanism to share code between classes.

```
37      dimensions() { ... };
38    };
39  };
```

Capabilities Modules and abstract data types provide encapsulation via controlling the visibility of identifiers. A more powerful method uses an object to guard the access to certain data. An outside function can only access the guarded data if it has the object guarding it. Furthermore, the object can ensure data integrity and other invariants by executing suitable code before or after the access. Such objects are called *capabilities* or *proxies.*

As an example, we can use objects to guard file operations.

```
1  let prox = object {
2    public:
3      read(str) { ... };
4      write()   { ... };
5    private:
6      file : file_handle;
7  };
```

Such an object could, for instance, handling the locking of the file, check permissions, or ensure that the client uses the correct file format.

8.4 Inheritance

With the object framework introduced so far, we have to write every class from scratch. It would be desirable to share common code between classes. Besides requiring less typing, this also increases code maintainability as changes in the code do not have to be repeated for every class. On the negative side, one has to note that such sharing reduces code locality, as the implementation of a class becomes distributed over several parts of the program. This is particularly problematic for objects with encapsulated state as it can be quite easy to lose track of all the places where this state is modified. We will call the mechanism for code sharing within a class framework *inheritance,* although strictly speaking this term only refers to using code from a parent class in a subclass. There are several ways to support inheritance, some more problematic than others.

Delegates Suppose we want to add classes for coloured shapes to the class hierarchy defined above. A coloured shape has two more methods to access the colour of the shape.

```
1  type coloured_shape = object {
2    draw       : unit -> unit;
3    move       : int -> int -> shape;
4    dimensions : unit -> [ min_x : int, min_y : int,
5                           max_x : int, max_y : int ];
```

Delegate. An object used as part of another object.

```
6     colour     : colour;
7     set_colour : colour -> unit;
8   };
```

One easy way to create objects for such a class is to use an object of the parent class as is and implement the methods of the new class using those of the old one. An object used as part of another one in this way is called a *delegate*.

```
1   let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
2     let p = new_point(x,y);
3     object {
4       draw()          { set_colour(col); p.draw() };
5       move()          { p.move() };
6       dimensions()    { p.dimensions() };
7       colour()        { c };
8       set_colour(col) { c := col };
9     };
```

Adding methods In the above example we directly called the methods of the delegate without any changes. In this case we can simplify the code slightly as follows.

```
1   let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
2     let p = new_point(x,y);
3     [ draw       = p.draw,
4       move       = p.move,
5       dimensions = p.dimensions,
6       colour     = fun () { c },
7       set_colour = fun (col) { c := col } ];
```

Adding syntactic sugar we can neaten up the code further and obtain something like the following.

```
1   let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
2     let p = new_point(x,y);
3     object {
4       include p;
5       colour()        { c },
6       set_colour(col) { c := col };
7     }
```

Replacing methods Just adding new methods is not always enough. Sometimes we also want to change existing ones. Let us first see how to replace an old method with a completely new one.

```
1   let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
```

```
    let p = new_point(x,y);
    [ draw       = fun () { set_colour(c); draw_point(x,y); },
      move       = p.move,
      dimensions = p.dimensions,
      colour     = fun () { c },
      set_colour = fun (col) { c := col } ];
```

There is one issue one has to be aware of when doing such a replacement: in a simple implementation like the one above, if an old method tries to call the replaced one, it will use the original version, not the new one. If that is not the desired behaviour, one has to implement what is called *open recursion* (see below).

In the following example, some methods of the superclass are mere stubs that are intended to be overwritten by each subclass. This is a common idiom in languages like C++ that use name equivalence for subtyping and that do not support object types (interfaces in Java's terminology). In such a language we can emulate object types in the following way via inheritance and subtyping.

```
class shape {
    draw()                  { () };
    move(dx : int, dy : int) { () };
    dimensions()            { [ min_x = 0, min_y = 0,
                                max_x = 0, max_y = 0 ] };
};

class point(x : int, y : int) extends shape {
  draw()       { draw_point(x,y)          };
  move(dx, dy) { x := x + dx; y := y + dy; };
  dimensions() { [ min_x = x, min_y = y, max_x = x, max_y = y ] };
};

class circle(x : int, y : int, r : int) extends shape {
  draw()       { draw_circle(x,y,r)       };
  move(dx, dy) { x := x + dx; y := y + dy; };
  dimensions() { [ min_x = x - r, min_y = y - r,
                   max_x = x + r, max_y = y + r ] };
};

class rectangle(x : int, y : int, w : int, h : int) extends shape {
  ...
};

class group(shapes : list(shape)) extends shape {
  ...
};
```

Modifying methods In the implementation of coloured points above we did repeat the code of the old methods in the definition of the new one. We can increase the amount of code reuse by using the old methods instead.

```
let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
  let super = new_point(x,y);
  [ draw       = fun () { set_colour(c); super.draw(); },
    move       = super.move,
    dimensions = super.dimensions,
    colour     = fun () { c },
    set_colour = fun (col) { c := col } ];
```

One question one has to address when designing an inheritance mechanism for a programming language is who is in command, the subclass or the superclass? That is, when invoking a method of a subclass, do we execute the function given in the subclass definition (which then may or may not call the function of the superclass), or do we execute the function of the superclass (which then can call the function of the subclass)? For instance, suppose we use a class hierarchy to model widgets in a graphical user interface. We might define a class for a general kind of text field and several subclasses for more special versions. Consider the method that gets called when the user presses a key. Do we want the superclass to first process this key press and then pass the keys it is not interested in to the subclass, or do we want it to be the other way round? The following examples illustrate the differences between these two approaches.

Example Let us discuss the various choices on how to use inheritance in the example of a class for buttons in a user interface. In most object-oriented GUI frameworks they are implemented using inheritance with modification.

```
type button = object {
    button_down : unit -> unit;
    button_up   : unit -> unit;
    ...
  };
let basic_button() {
  object {
    button_down(self) { ... draw button ... };
    button_up(self)   { ... draw button ... };
    ...
  };
};
let my_button() {
  let super = basic_button();
  object {
    include super;
    button_down(self) {
      super.button_down(self);
```

```
19          ... do something ...
20        };
21        ...
22      }
23    };
```

Note that in this solution, it is not obvious how a subclass is intended to call the button superclass. When should it call the `button_down` method of the superclass? At the beginning of its own method, at the end, somewhere in between? Should it call it at all? Here we see why it is sometimes better to be able to call subclass methods via `outer` instead of superclass methods via `super`.

We can clean this design up, by splitting the `button_down` method into two parts. One part to be overwritten by the superclass and one to be left alone.

```
1   type button = object {
2       button_down    : unit -> unit;
3       button_up      : unit -> unit;
4       button_pressed : unit -> unit;
5       ...
6     };
7   let basic_button() {
8     object {
9       button_down(self) {
10        ... draw button ...
11        self.button_pressed();
12      };
13      button_up(self) { ... draw button ... };
14      button_pressed(self) { () };
15      ...
16    }
17  };
18  let my_button() {
19    let super = basic_button();
20    object {
21      include super;
22      button_pressed(self) {
23        ... do something ...
24      };
25      ...
26    }
27  };
```

Finally, we can simplify our implementation further, by using a first-class function instead of inheritance.

```
1   type button = object {
2       button_down    : unit -> unit;
```

Open recursion. Methods get a pointer to the object, so they can use dynamic dispatch.

```
    button_up      : unit -> unit;
    ...
  };
let basic_button(pressed : unit -> unit) {
  object {
    button_down(self) {
      ... draw button ...
      pressed();
    },
    button_up(self) {
      ... draw button ...
    };
    ...
  }
};
```

In this case we do not need to define new classes at all. We can simply use the base class as is.

Open Recursion It is frequently the case that some method of the superclass calls another method of the superclass that is overridden in the subclass. In this case we have to decide which version of the method to execute, the one in the superclass or in the subclass.

```
type widget = object {
  draw   : unit -> unit;
  resize : int -> int -> unit;
  ...
};
let new_widget(width,height) {
  object {
    draw() { () }
    resize(w,h) { width := w; height := h; draw(); }
  }
};

type text_field = object { ... };
let new_text_field() {
  object {
    draw() { ... };
    ...
  };
};
```

Usually, we want to use the version in the subclass. In order to get access to this function, we need to have the object available during the method call. The standard solution is to pass the object as an additional, implicit parameter to every method.

```
type widget = object {
  draw   : unit -> unit;
  resize : int -> int -> unit;
  ...
};
let new_widget(width,height) {
  object {
    draw(obj) { () }
    resize(obj,w,h) { width := w; height := h; obj.draw(); }
  }
};

type text_field = object { ... };
let new_text_field() {
  object {
    draw(obj) { ... };
    ...
  };
};
```

Type classes Type classes also offer two of the forms of inheritance discussed above. Firstly, we can extend a given type class with new functions.

```
typeclass Eq(a) {
  equal      : a -> a -> bool;
  not_equal  : a -> a -> bool;
};
typeclass Eq(a) => Ord(a) {
  type cmp = | LT | EQ | GT;
  compare : a -> a -> cmp;
};

instance Eq(int) {
  equal(x,y)     { prim_equal_int(x,y) };
  not_equal(x,y) { not(equal(x,y)) };
};
instance Ord(int) {
  compare(x,y) { if x < y then LT else if x > y then GT else EQ };
};
```

Secondly, a type class can offer a default implementation that may be overwritten by the instance.

Mixin. Code sharing without subtyping.

```
typeclass Eq(a) {
  equal      : a -> a -> bool;
  not_equal  : a -> a -> bool;
  not_equal(x,y) { not(equal(x,y)) };
};

instance Eq(int) {
  equal(x,y) { prim_equal_int(x,y) };
};
```

Multiple inheritance Inheritance is mainly a mechanism to reuse code from existing objects. Sometimes one would like to use code of several objects at once. Therefore some languages (most notably C++) allow to define classes that extend several superclasses at the same time. This is called *multiple inheritance*. While adding more power to the language, multiple inheritance makes the object system also considerably more complicated. For instance, what happens if several of the superclasses have methods with the same name? Does this result in an error, or do we simply pick one of the methods for the subclass? Another problematic situation is the following one. Suppose we have two classes B and C that both inherit from some class A. What happens if we inherit a class D from both B and C? Do we get two copies of the class A or only one? For these reasons, many modern languages do not support multiple inheritance and try to provide alternative, cleaner ways to achieve the same effects. For instance, Java does only support single inheritance, but it allows classes to implement multiple interfaces.

Mixins The main reason why multiple inheritance is problematic, is the fact that in most languages inheritance is the only mechanism to define class hierarchies. A declaration like

```
class B extends A { ... }
```

both declares B as a subclass of A and lets B inherit the methods of A. If we provide separate mechanisms for inheritance and the declaration of subtyping relationships, the object system becomes much simpler and cleaner.

How could an inheritance mechanism look like that is decoupled from subtyping? One example of such a mechanism is called *mixins*. A mixin is a function $F : I \to J$ that takes a class A of a specified object type I and produces a new class $F(A)$ of type J. Hence, a mixin is very similar to the parametrised modules we have described in Section 5.4. As an example let us consider a mixin that turns shapes into coloured shapes.

```
type coloured_shape = object {
  draw       : unit -> unit,
  move       : int -> int -> shape,
  dimensions : unit -> [ min_x : int, min_y : int,
                         max_x : int, max_y : int ],
```

```
6     colour     : colour,
7     set_colour : colour -> unit
8   };
9
10  let make_coloured(s : shape, c : colour) : coloured_shape =
11    [ draw       = fun () { set_colour(c); s.draw() },
12      move       = s.move,
13      dimensions = s.dimensions,
14      colour     = fun () { c },
15      set_colour = fun (col) { c := col } ];
```

So, instead of using inheritance to extend a class A to a subclass B, we can use a mixin F such that $B = F(A)$. In some cases, we can use mixins also to simulate multiple inheritance. Suppose that A is a common superclass of both B and C, and D inherits from B and C. If we can write $B = F(A)$ and $C = G(A)$ with mixins F and G, then we can try to express $D = H(G(F(A)))$ as an extension of $G(F(A))$ via a third mixin H.

8.5 Discussion

The problem with many object-oriented languages is that they offer a single mechanism (class definitions) that combines all the object features. This makes the language very complex and has lead to much confusion about object-oriented design. A much cleaner and simpler design is to provide separate mechanisms for subtyping, dynamic dispatch, encapsulated state, and inheritance.

For instance, there is an old debate on whether subclasses should represent an 'is-a' or a 'has-a' relationship. Separating the aspects of object-oriented design, we see that an 'is-a' relationship is precisely modelled by the subtyping relation, whereas a 'has-a' relationship is more suitably modelled by some form of inheritance or the use of delegates.

Separation also allows one to only use those features necessary for a particular solution. For instance, if stateless objects are sufficient for the task at hand, we can avoid the added complexities involved with side-effects.

Let me conclude this chapter with a word of advice: while subtyping and object-oriented programming as a whole are quite powerful, they are also quite complex. They are not always the best way to solve a problem. Only use them if they make the resulting program simpler. If a purely functional solution, or a plain imperative one, works as well, there is no need to resort to objects.

Also one can easily get carried away with designing elaborate class hierarchies, instead of writing code that actually does something. For instance, if you are about to define several helper classes to perform a single task, you should ask yourself whether you really need that many classes or whether a different approach would not offer a simpler solution. For instance, if it is possible to use them, higher-order functions and parametric polymorphism are usually the better approach.

9 Concurrency

So far in our language the evaluation order is completely *deterministic*. If we run a program several times, we observe the same ordering each time. In this chapter we study language features that cause the evaluation order to be *non-deterministic*. In this case we say that the program is *concurrent*. The most common case of concurrency is when the program consists of several threads of processes that are executed in *parallel*. But there are also examples of concurrency without parallelism.

Of course, concurrency increases the complexity of programs and makes them harder to reason about, in particular, when combined with side-effects which, as we have seen, care about the evaluation order of expressions. Purely functional, concurrent programs are much better behaved.

On the language level concurrency manifests in having several independent 'paths of execution', that is, instead of having a single code location identifying the expression that is to be executed next, there can be several such locations. Each of the expressions pointed to will be executed eventually, but the ordering in which this happens is left unspecified. Such a path of execution is called a *fibre* of the program. If fibres are executed in parallel, they are also called *threads* or *processes*. Fibres that are not executed in parallel are called *coroutines*. Thus, a fibre is a part of the program with its own control flow. Within each fibre execution is linear, but the program execution can jump between fibres at certain points. Even without parallelism, fibres have the advantage that the program is no longer restricted to a single syntactic nesting and stack discipline. It can use several of them in parallel.

The main problem of concurrent programming is to organise the communication between different fibres. This is called *synchronisation*. There are two fundamentally different methods for synchronisation: *message passing* and *shared memory*. We will consider each one in turn.

9.1 Fibres

Before turning to the synchronisation problem let us first see how to implement fibres in our language. As real parallelism requires support from the operating system, our implementation will be non-parallel and based on a form of *cooperative multi-threading*. We start with the following functions.

```
1  make_ready : (unit -> unit) -> unit;
2  schedule   : unit -> unit;
3  spawn      : (unit -> unit) -> unit;
4  yield      : unit -> unit;
```

Concurrency. Non-deterministic order of execution.

Parallelism. Instructions are executed at the same time.

Fibre. A path of execution in the program.

The two low-level functions `make_ready` and `schedule` respectively add a fibre to the list of running fibres and execute the next fibre in the list. The high-level functions `spawn` and `yield` respectively create a new fibre and mark a place where we can switch from one fibre to another one. We can implement them as follows.

```
1  let ready_fibres = Queue.make ();
2
3  type terminate = | Terminated;
4
5  let make_ready(f) {
6    Queue.push(ready_fibres, f);
7  };
8
9  let schedule() {
10   if Queue.is_empty(ready_fibres) then
11     throw Terminated
12   else {
13     let f = Queue.pop(ready_fibres);
14     f();
15     schedule()
16   }
17 };
18
19 let start_scheduler() {
20   try
21     schedule()
22   catch e => case e
23   | Terminated => ()
24   | else       => print "uncaught exception" e
25 };
26
27 let spawn(f) {
28   letcc k {
29     make_ready(k);
30     make_ready(f);
31     schedule()
```

Thread. A parallel fibre.

Coroutine. A non-parallel fibre.

Synchronisation. Communication between fibres.

```
32      }
33    };
34
35    let yield() {
36      letcc k { make_ready(k); schedule() }
37    };
```

The module `Queue` contains a simple queue implementation where we can add elements at one end and remove them at the other one. A simple example of how to use fibres, consider the following program where two fibres print some numbers in an arbitrary ordering. (With the implementation above, the fibres alternate.)

```
1   let f1() {
2     for i = 1 .. 10 {
3       print "fibre1:" i;
4       yield();
5     }
6   };
7   let f2() {
8     for i = 1 .. 10 {
9       print "fibre2:" i;
10      yield();
11    }
12  };
13  spawn(f1);
14  spawn(f2);
15  start_scheduler()
```

With only these two operations fibres are not of much use. We also need some operations for synchronisation of and communication between fibres. We start by defining a few primitive operations that can be then used to implement more complex communication mechanisms. These operations are based on the notion of a *condition*. A fibre can wait on such a condition until it is woken up by some other fibre.

```
1   type condition(a);
2   new_condition : unit -> condition(a);
3   wait          : condition(a) -> a;
4   wait_multi    : list(condition(a)) -> a;
5   resume        : condition(a) -> a -> unit;
```

`new_condition` creates a new condition, `wait(c)` sends the current fibre to sleep, waiting on the condition `c`, and `resume(c)` wakes up all fibres waiting on `c`.

```
type trigger(a)   = [ cont : a -> void, triggered : bool ];
type condition(a) = [ waiting : list(trigger(a)) ];

let make_trigger(k) {
  [ cont = k, triggered = False ]
};

let resume_trigger(t,v) {
  if t.triggered then
    ()
  else {
    t.triggered := True;
    make_ready(fun () { t.cont(v) });
  }
};

let new_condition() {
  [ waiting = [] ]
};

let resume(c,v) {
  let waiting = c.waiting;
  c.waiting := [];
  List.iter(fun (k) { resume_trigger(k,v) },
            waiting);
};

let wait(c) {
  letcc k {
    let t = make_trigger(k);
    c.waiting := [t | c.waiting];
    schedule();
  }
};

let wait_multi(cs) {
  letcc k {
    let t = make_trigger(k);
    List.iter(fun (c) { c.waiting := [t | c.waiting] },
              cs);
    schedule();
  }
};
```

Mutual exclusion. Making sure that certain parts of the program are not executed in parallel.

Using conditions the example from above can be written as follows. This code strictly alternates between fibres irrespective of the way the scheduling algorithm works.

```
1  let c1 = new_condition();
2  let c2 = new_condition();
3
4  let f1() {
5    for i = 1 .. 10 {
6      print "fibre1:" i;
7      resume(c2, ());
8      wait(c1);
9    };
10   resume(c2, ());
11 };
12
13 let f2() {
14   for i = 1 .. 10 {
15     print "fibre2:" i;
16     resume(c1, ());
17     wait(c2);
18   };
19   resume(c1, ());
20 };
21
22 spawn(f1);
23 spawn(f2);
24 start_scheduler()
```

9.2 Ramifications

When adding concurrency to a language many new phenomena and problems arise. Let us discuss a few of them.

Mutual exclusion When two regions of code in different fibres want to modify the same data structure, it is usually required that the control flows of the two fibres do not enter these regions simultaneously. This is called the *mutual exclusion problem* and most synchronisation problems can be reduced to it. Many of the synchronisation mechanisms we will introduce below have specifically been invented to ensure mutual exclusion.

Deadlocks A *deadlock* is a situation where there are several fibres, each waiting on an action that can only be performed by one of the other waiting fibres.

Deadlock. A cycle of fibres waiting on each other.

Race condition. A bug that only occurs under a specific timing.

Race conditions A *race condition* is a bug that is caused by the particular choices and timing of the scheduler.

Starvation If a fibre is ready but it is never executed because there is always another fibre that goes first, we say the fibre is *starving*.

Lifeness Lifeness is the opposite of starvation: every ready fibre is executed eventually.

Fairness When several fibres compete for a certain resource, we ideally want them to get access to the resource in equal amounts. This is called *fairness*.

9.3 Message passing

Having a basic implementation of fibres we can turn to communication mechanisms. We start with message passing. The central concept of message passing are objects called *channels* or *ports*. A channel is a line of communication between processes that supports two operations: one process can write data, a *message,* to the channel and the other one can read it. Channels come in many variants. They might be

- *synchronous:* a writer waits until the other process reads the message, a reader waits until the other process has supplied a message;
- *asynchronous and buffered:* a writer can continue immediately after sending the message, a reader must still wait until a message is available; there can be several messages waiting for the reader;
- *asynchronous and unbuffered:* a writer can continue immediately after sending the message, a reader must still wait until a message is available; there can be at most one message waiting for the reader; if the writer wants to send a second message, he blocks until the reader has read the first one;
- *one-directional:* a channel is split into two parts: one for reading and one for writing, if a process has access to only one of the parts, it can perform only the corresponding operation.
- *bidirectional:* each end of a channel can be used both for reading and for writing.

Before giving an example, let us describe the library functions we need to implement.

Starvation. A ready fibre is never scheduled.

```
new_channel : unit -> channel(a)
send        : channel(a) -> a -> unit
receive     : channel(a) -> a
```

`new_channel()`	creates a new channel
`send(ch,x)`	sends the value `x` over the channel `ch`
`receive(ch)`	reads a value from the channel `ch`

In our implementation, channels are synchronous and bidirectional.

```
1  type channel_state(a) = | Free | Reading | Written(a);
2  type channel(a)       = [ state   : channel_state(a),
3                            readers : condition(a),
4                            writers : condition(unit) ];
5
6  let new_channel() {
7    [ state   = Free,
8      readers = new_condition(),
9      writers = new_condition() ]
10 };
11
12 let receive(ch) {
13   case ch.state
14   | Free       => { ch.state := Reading; wait(ch.readers) }
15   | Written(v) => { ch.state := Free;    resume(ch.writers, ()); v }
16   | Reading    => error
17 };
18
19 let send(ch,v) {
20   case ch.state
21   | Free       => { ch.state := Written(v); wait(ch.writers); }
22   | Written(v) => error
23   | Reading    => { ch.state := Free; resume(ch.readers,v) }
24 };
25
26 let merge(ch1,ch2) {
27   let merge_fibre(ch1,ch2,c) {
28     while True {
29       case ch1.state
30       | Written(v) => send(c,receive(ch1))
31       | else       => case ch2.state
32           | Written(v) => send(c,receive(ch2))
33           | else       => { ch1.state := Reading;
34                             ch2.state := Reading;
35                             wait_multi([ch1, ch2]); }
```

```
36        }
37      };
38      let c = new_channel();
39      spawn(fun () { merge_fibre(ch1,ch2,c) });
40      c
41    };
```

Example As an example of how to use message passing, let us take a look at the well-known producer–consumer problem.

```
1   let produce(ch) {
2     while True {
3       let x = get_next_item();
4       send(ch,x);
5     };
6   };
7
8   let ch = new_channel();
9   spawn(fun () { produce(ch) });
10  spawn(fun () { consume(ch) });
11  start_scheduler()
```

```
let consume(ch) {
  while True {
    let x = receive(ch);
    process_item(x);
  };
};
```

Example Suppose we have a user interface where we want to implement drag-and-drop. The usual GUI frameworks have an event-loop where the program can register call-backs for various events like mouse clicks. When using such a framework, we face the problem of how to remember the program state between user inputs. The typical solution looks as follows.

```
1   type state = | Idle | Dragging(obj);
2
3   let state = Idle;
4
5   let mouse_down(x,y) {
6     case state
7     | Idle          => case object_under_pointer(x,y)
8                        | None      => Nothing
9                        | Some(obj) => state := Dragging(obj)
10    | Dragging(obj) => Nothing
11  };
12  let mouse_up(x,y) {
13    case state
14    | Dragging(obj) => { move_object_to(obj,x,y); state := Idle; }
15    | Idle          => Nothing
16  };
17  let mouse_move(x,y) {
```

```
    case state
    | Dragging(obj) => move_object_to(obj,x,y);
    | Idle          => Nothing
  };

  register_call_back_mouse_down(mouse_down);
  register_call_back_mouse_up(mouse_up);
  register_call_back_mouse_move(mouse_move);
```

When having more states than 'idle' and 'dragging', this quickly become tedious. Using fibres we can avoid having to manage the program state explicitly.

```
  type event = | Start(object) | Move(int,int) | Drop;

  let drag_and_drop(ch) {
    case receive(ch)
    | Start obj =>
        while True {
          case receive(ch)
          | Move(x,y) => move_object_to(obj,x,y);
          | Drop      => break
        };
  };

  let mouse_down(ch,x,y) {
    case object_under_pointer(x,y)
    | None      => Nothing
    | Some(obj) => send(ch, Start(obj))
  };
  let mouse_up(ch,x,y) {
    send(ch, Drop(x,y));
  };
  let mouse_move(ch,x,y) {
    send(ch, Move(x,y));
  };

  let ch = make_channel();
  spawn(drag_and_drop(ch));
  register_callback_mouse_down(mouse_down(ch));
  register_callback_mouse_up(mouse_up(ch));
  register_callback_mouse_move(mouse_move(ch));
```

Inversion of control If we do not use fibres, communication between program units is asymmetric. One unit is in control and calls the functions provided by the other unit. In the above

Inversion of control. A code transformation that reverses the master/slave relationship.

example, the event loop was in control and invokes the callbacks provided by the main program. Alternatively, we could have the main program be in control and call a library function to get the next event. But we must choose between these two options. Going from one to the other is called *inversion of control.* The choice between these two versions requires careful consideration, as it has a big influence on the structure of the whole program.

The big advantage of using fibres is that we do not need to choose: both parts can be in control at the same time and communicate via channels. Thus communication is symmetric.

When compared to shared-memory communication, which we will describe next, message passing has some overhead as messages must be constructed and passed to another process. But it scales really well to any number of processes and it works equally well on a single computer or on a distributed system. Furthermore, it is conceptually really simple and easy to use for the programmer. In my opinion it is therefore clearly superior to approaches relying on shared-memory.

Futures Futures are a simple synchronisation mechanism where we can evaluate a given expression in parallel and wait for the result. They work like a channel that can only be used a single time. The implementation is straightforward using the tools we already have. For instance, we can use channels.

```
1  let future(e) {
2    let ch = new_channel();
3    spawn(fun () { send(ch,e) });
4    ch
5  };
6
7  let get(f) { receive(f) };
```

We can also use single-assignment variables (if we use the convention that reading from an uninitialised single-assignment variable blocks until some other fibre assigns a value to it.)

```
1  let future(e) {
2    let x;
3    spawn(fun () { x := e });
4    x
5  };
6
7  let get(f) { let y = f; y };
```

Reactive programming An example of a very pure form of message passing concurrency is given by a programming paradigm called reactive programming. In a reactive program the programmer uses streams of data to connect time dependent data sources with objects reacting to the data. This is usually done in a purely functional way. For instance, there can be data sources for the current mouse position, the state of the mouse buttons, or the current time. The streams

Atomic operation.	No intermediate states are visible from the outside.

created by these sources can be modified by operations like counting the number of events, filtering out uninteresting ones, or applying arbitrary functions to the data. Finally, such streams are connected to objects that react to the data, like, a text field that reads strings from a stream and displays them.

```
let a = ... mouse.x ... mouse.y ... ;
let b = ... time ... ;
let c = ... a ... b ... ;

let num_clicks = count(mouse_button);
let str = sprintf("The mouse is at (%d, %d). There have been %d clicks.",
                  mouse.x, mouse.y, num_clicks);
let fld = make_text_field(str);
```

Because of its declarative nature this programming style is very easy to use and programs are highly compositional. Furthermore, typical problems of concurrent programs like race conditions and deadlocks are automatically avoided.

On the other side, it is not suitable for every problem. It seems to work well for simple user interfaces and dynamical web-pages, but in more complex situations it quickly becomes cumbersome. In addition, reactive programming sacrifices performance for reliability and ease of use.

9.4 Shared-memory

When several fibres or threads run on the same processor they can use the shared memory to communicate. Of course this relies on side-effects and, as the evaluation order matters with side-effects, the non-determinism of this order inherent in concurrency makes such program even harder to understand. Over the years people have invented several mechanisms and constructs to make shared memory communication easier to use and less error prone.

Atomic operations Before presenting the various synchronisation mechanisms let us introduce the primitive operations we need to implement them. These are operations that are *atomic* which means that, when executing such an operation we cannot observe any intermediate state. Either we see the state before the operation or we see the resulting state, but we can never find the operation to be halfway executed.

Usually processors provide a few special atomic instructions to build concurrency mechanisms with. Typical examples are a *test-and-set* and a *compare-and-swap* operation.

```
test_and_set     : location -> a -> a;
compare_and_swap : location -> a -> a -> bool;
```

The operation `test_and_set(x,a)` stores the value `a` in the variable `x` and returns the old value of `x`. `compare_and_swap(x,a,b)` compares the value in the memory location `x` with the value `a`.

If they are the same, it sets the value of `x` to `b`. Otherwise, it leaves `x` unchanged. The return value indicates whether a change occurred.

Locks/Mutexes A *lock* (also called a *mutex*) is a data structure with two states. It can either be *locked* by a certain fibre (we say that the fibre *holds* the lock, or that it has *acquired* the lock) or it is *open*. There are two operations.

```
type lock;
lock   : lock -> unit;
unlock : lock -> unit;
```

When a fibre calls `lock` on an open lock, the lock changes state to *locked* and it is now held by the fibre. When called with a lock that is hold by another fibre, the operation blocks until that fibre unlocks it. What happens when a fibre calls `lock` on a lock that is held by itself depends on the particular implementation. Some implementations just block, which results in a deadlock (as the fibre waits on itself to release the lock). In this case, we call the locks *non-reentrant*. The more sensible solution is to allow a fibre to acquire a lock several times (of course, in this case it has to release the lock the same number of times before the lock is open again). Such locks are called *reentrant*.

For simplicities sake, we will implement non-reentrant locks.

```
type lock = [ locked : bool; waiting : condition() ];

let new_lock() {
  [ locked = False, waiting = new_condition() ]
};

let lock(l) {
  while test_and_set(l.locked, True) {
    wait(l.waiting)
  }
};

let unlock(l) {
  l.locked := False;
  resume(l.waiting, ());
};
```

Using locks manually is quite error prone. If several locks are involved it is important to lock and unlock them in the correct order. Also it is easy to forget some of the unlock calls. Some of these errors can be avoided if the language has built in support for locks, like the following

```
lock name {                    lock(name);
  ...                    ⇒       ...
}                              unlock(name);
```

Example: lock-free stack Sometimes it is not possible to meet the performance requirements when using higher-level constructs like locks. In these cases one is forced to directly use atomic operations for synchronisation. Let us give an example of how to implement a stack in this way. We start with a non-thread-safe version.

```
type node(a)  = [ next : node(a), data  : a   ];
type stack(a) = [ head : node(a) ];

let push(st : stack(a), x : a) : unit {
  let n = [ next = null, data = x ];
  let old_head = st.head;
  n.next := old_head;
  st.head := n;
};

let pop(st : stack(a)) : a {
  let old_head = st.head;
  if old_head == null then
    throw Empty;
  let new_head = old_head.next;
  st.head := new_head;
  return old_head.data;
};
```

When several threads call `push` or `pop` at the same time, this code can corrupt the data structure. For instance, it is possible to pop the same element several times, or to silently drop pushed elements. To fix this, we have to make sure when modifying the head pointer of the stack at the end of `push` and `pop`, that it was not modified by another thread in the mean time. This can be done with `compare_and_swap`.

```
type node(a)  = [ next : node(a), data  : a   ];
type stack(a) = [ head : node(a) ];

let push(st : stack(a), x : a) : unit {
  let n = [ next = null, data = x ];
  while true {
    let old_head = atomic_read(&st.head);
    n.next := old_head;
    if compare_and_swap(&st.head, old_head, n) then
      return;
  };
};
let pop(st : stack(a)) : a {
  while true {
    let old_head = atomic_read(&st.head);
    if old_head == null then
```

```
17          throw Empty;
18        let new_head = old_head.next;
19        if compare_and_swap(&st, old_head, new_head) then
20          return old_head.data;
21      };
22    };
```

This code looks thread-safe, but it still has a subtle threading bug: `compare_and_swap` only checks whether the head pointer of the stack has not changed. But it can be that some other thread popped the top element of the stack and then pushed new elements, and coincidently the last node added was allocated at the same memory address as the popped one. Then `compare_and_swap` thinks the stack was not modified, while in fact is was. This is called the *ABA problem*. The easiest fix is to add a serial number to the stack that is incremented each time some element is popped.

```
1   type node(a)  = [ next : node(a), data  : a   ];
2   type stack(a) = [ head : node(a), count : int ];
3
4   let push(st : stack(a), x : a) : unit {
5     let n = [ next = null, data = x ];
6
7     while true {
8       let old_head = atomic_read(&st.head);
9       n.next := old_head;
10      if compare_and_swap(&st.head, old_head, n) then
11        return;
12    };
13  };
14
15  let pop(st : stack(a)) : a {
16    while true {
17      let old_head = atomic_read(&st.head);
18
19      if old_head == null then
20        throw Empty;
21
22      let old_count = atomic_read(&st.count);
23      let new_head  = old_head.next;
24      let new_count = old_count + 1;
25
26      if compare_and_swap2(&st,
27           old_head, old_count,
28           new_head, new_count) then {
29        return old_head.data;
30      }
31    };
```

```
};
```

Here, `compare_and_swap2` is a version of `compare_and_swap` that swaps two values at the same time.

Condition variables A condition variable is a condition that is associated with a lock. Waiting on the condition also unlocks the lock and when it is woken up again it automatically acquires the lock again. This simplifies the common case where a fibre has to wait while holding a lock.

```
type cvar = [ cond : condition, lock : lock ];

let new_cvar(l) {
  [ cond = new_condition(), lock = l ];
};

let wait_cvar(c) {
  unlock(c.lock);
  wait(c.cond);
  lock(c.lock);
};

let resume_cvar(c) {
  resume(c.cond, ());
};
```

Semaphores A semaphore is a generalisation of a lock that has several degrees of being unlocked. It takes the form of an integer counter that cannot go below zero. If a fibre tries to decrease the counter when it already is zero, it blocks instead until another fibre increases the counter again. So, we have two operations: one to increment the counter and one to decrement it.

```
type semaphore;
increment : semaphore -> unit;
decrement : semaphore -> unit;
```

The implementation is as follows.

```
type semaphore = [
  count   : int,
  lock    : lock,
  waiting : cvar
];

let new_semaphore() {
  let l  = new_lock();
  let cv = new_cvar(l);
  [ count   = 0,
```

```
    lock    = l,
    waiting = cv ]
};

let increment(sem) {
  lock(sem.lock);
  sem.count := sem.count + 1;
  resume_cvar(sem.waiting);       // this automatically unlocks the lock
};

let decrement(sem) {
  lock(sem.lock);
  while sem.count == 0 {
    wait_cvar(sem.waiting);
  }
  sem.count := sem.count - 1;
  unlock(sem.lock);
};
```

Example The following producer–consumer implementation uses a buffer whose size is bounded by some constant `n`.

```
let lock   = new_semaphore();
let full   = new_semaphore();
let empty  = new_semaphore();
let n      = 10;
let buffer = new_buffer(n);

for i = 1 .. n {              // the initial value of empty should be n
  increment(empty)
};

increment(lock);

let producer() {
  for i = 0 .. 100 {
    let x = ... generate a value ...;
    decrement(empty);
    decrement(lock);
    put(buffer, x);
    increment(lock);
    increment(full);
  }
};
```

```
23
24  let consumer() {
25    for i = 0 .. 100 {
26      decrement(full);
27      decrement(lock);
28      let x = get(buffer);
29      increment(lock);
30      increment(empty);
31      ... process the value x ...
32    }
33  };
```

Monitors A *monitor* is an abstract data type that is protected by a lock. Each operation on the type first acquires the lock and releases it again upon return. This makes the operations atomic. For instance, a monitor for a queue implementation could look as follows.

```
1   type node(a) = [ ... ];
2
3   type queue(a) = [
4     lock  : lock,
5     front : node(a),
6     back  : node(a)
7   ];
8
9   let make() {
10    let node = [ ... ];
11    [ lock = new_lock(), front = node, back = node ]
12  };
13
14  let pop(q) {
15    lock(q.lock);
16    ... remove the first node from the list ...
17    unlock(q.lock);
18    ... return the data stored in the removed node ...
19  };
20
21  let push(q,x) {
22    lock(q.lock);
23    ... add a new node at the end of the list ...
24    unlock(q.lock);
25  };
```

Some languages, most notably Java, provide syntactic sugar for monitors. For instance, in Java you can declare methods as `synchronized` which automatically protects the body by `lock` and `unlock` statements.

The obvious advantage of monitors is that they provide a very easy way to write correct synchronisation code. Their main disadvantage is that, like most automatic mechanisms, they can lead to poor performance by taking locks in situations where it is not necessary.

Transactional memory Transactional memory is a general mechanism to make arbitrary operations atomic. A *transaction* is a piece of code that can either succeed or fail with its task. When it fails it has no effect on the program, it is as if the transaction was never executed. Thus, transactional memory can be seen as a form of backtracking.

We add the following constructs to our language.

$$\langle expr \rangle \ ::= \ldots \mid \texttt{atomic\{}\ \langle expr \rangle\ \texttt{\}} \mid \texttt{abort} \mid \texttt{retry}$$

An expression of the form `atomic { e }` evaluates the expression `e` as if it were atomic. That means that no other fibres can see intermediate states of the execution of `e`. They either see the state before its execution or the one after it.

In addition, the expression `e` can contain `abort` and `retry` statements to indicate, respectively, that the transaction has failed, or that it should be restarted from the beginning.

From a programmer's perspective, transactional memory is by far the easiest to use mechanism for shared memory concurrency. It automatically avoids the typical errors associated with this form of concurrency, like deadlocks and race conditions. Furthermore, the resulting code is much more composable and modular than code written with other primitives.

Of course this convenience comes at a significant cost. Transactional memory is very hard to implement and it comes with a considerable overhead. In particular, acceptable performance requires special hardware support, which most mainstream processors do not offer yet. The runtime cost is increased further by the fact that we sometimes need to execute a transaction several times for it to succeed. Finally, since transactional memory relies on backtracking, some operations cannot be used inside a transaction. In particular IO operations are not supported.

Discussion Shared-memory communication is very efficient, but it requires all processors to share memory. This becomes quickly impractical as their number increases. From a programmers perspective the main problem with shared-memory communication is that it is very error prone. The reason is that mechanisms for shared-memory communication require side-effects and we have seen that, when programming with side-effects, the order of execution becomes important. In a concurrent setting, this order is non-deterministic and it is therefore much harder to reason about. This added complexity makes it almost impossible to write correct code in this setting.

10 Various topics

10.1 Scripting languages

Scripting languages fill a niche that is very different from ordinary programming. Most non-toy programming languages are designed with large code bases in mind, while scripting languages specialise in very small programs, where the trade-offs involved are quite different.

Typical use cases for scripting languages are single-file programs with one or two hundred lines of code and, in particular, programs like shell scripts you run once and then discard. As a consequence such languages are optimised for convenience and ease of use, while readability and intelligibility take a back-seat. (It is much easier to understand code and to ensure its correctness, if the amount of code is that small.)

Let us mention a few examples where scripting languages can be used.

- shell scripts and a language for the command line;
- a macro language for an application like a word processor, a spread sheet program, etc.;
- providing a simple interface to libraries which allows also non-programmers (e.g., scientists) to write the glue code connecting such libraries. (For instance, there are Python frameworks that are very popular for numerical computations, data analysis, and machine learning.)

As mentioned, the trade-offs involved in designing a scripting language are quite different to other languages. Here are a few examples.

- An interpreter is frequently superior to a compiler, as compile times easily outnumber execution times.
- A REPL ('read-evaluate-print loop') can be useful.
- Scripting languages are the only case I can think of where dynamic typing actually makes sense.
- The syntax is frequently rather concise and it might supports many different ways to express the same thing.

Finally, let us mention a few typical consequences of such design choices.

- Scripting languages are usually easier to use for people that are not programmers.
- Most scripting languages are completely unusable for programs with more than, say, 400-500 lines of code (although a lot of people do so anyway).
- Performance is usually quite low (e.g., due to being interpreted or due to using dynamic typing).

10.2 Metaprogramming

Metaprogramming is the practice of using programs to write, modify, or check other programs. Typical use-cases include

- the automation of tedious tasks (e.g., keeping several parts of a program that depend on each other synchronised, generating bindings for a large library,…);
- checking for domain-specific errors;
- code instrumentation (e.g., adding profiling code or debugging checks);
- generation of highly-specialised code that is too long or too complicated to write manually.

The simplest way to add metaprogramming capabilities to a language is by implementing macros. This can be done in several ways, which differ in their degree of sophistication.

- The simplest way to implement macros is as textual substitutions that work independently of the syntactic structure. The C preprocessor (which used to be a separate program) is a prime example of this approach.
- Less powerful, but also easier to understand are macro systems working on the syntax level, i.e., macros that replace one syntax tree by another one.
- Finally, there are so-called *hygienic macros,* which also work on the syntax level, but that additionally know how to handle local variables. When applying such a macro, all variables local to the macro are renamed to prevent clashes with already existing identifiers. Thus, a hygienic macro is very similar to a function that is always inlined.

For more elaborate kinds of metaprogramming, the language needs to support *first-class code,* either built into the language itself, or via a library. At the very least there have to be data types for syntax trees and a way to get the syntax tree of a program. More advanced systems additionally provide a way to execute code given by such a syntax tree. The prime example here is Lisp, where expressions are just lists and where there exists a built in evaluation function.