Using,
Understanding, and
Unraveling

The OCaml Language

From Practice to Theory
and vice versa

Didier Rémy

These notes have also been published in Lectures Notes in Computer Science. A preliminary version was written for the Appsem 2000 summer school held in Camina, Portugal on September 2000.

Abstract

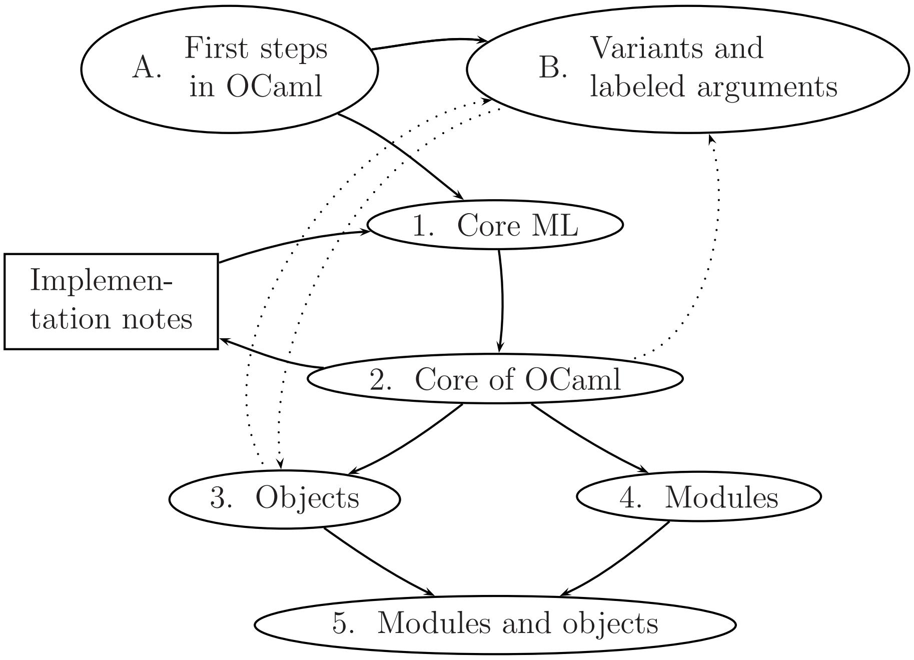
These course notes are addressed to a wide audience of people interested in modern programming languages in general, ML-like languages in particular, or simply in OCaml, whether they are programmers or language designers, beginners or knowledgeable readers —little prerequiresite is actually assumed.

They provide a formal description of the operational semantics (evaluation) and statics semantics (type checking) of core ML and of several extensions starting from small variations on the core language to end up with the OCaml language —one of the most popular incarnation of ML— including its object-oriented layer.

The tight connection between theory and practice is a constant goal: formal definitions are often accompanied by OCaml programs: an interpreter for the operational semantics and an algorithm for type reconstruction are included. Conversely, some practical programming situations taken from modular or object-oriented programming patterns are considered, compared with one another, and explained in terms of type-checking problems.

Many exercises with different level of difficulties are proposed all along the way, so that the reader can continuously checks his understanding and trains his skills manipulating the new concepts; soon, he will feel invited to select more advanced exercises and pursue the exploration deeper so as to reach a stage where he can be left on his own.

Figure 1: **Road map**



Legend of arrows (from *A* to *B*)

⟶ *A* strongly depends on *B*

⇢ Some part of *A* weakly depends on some part of *B*

Legend of nodes

- Oval nodes are physical units.
- Rectangular nodes are cross-Chapter topics.

Contents

Introduction

OCaml is a language of the ML family that inherits a lot from several decades of research in type theory, language design, and implementation of functional languages. Moreover, the language is quite mature, its compiler produces efficient code and comes with a large set of general purpose as well as domain-specific libraries. Thus, OCaml is well-suited for teaching and academic projects, and is simultaneously used in the industry, in particular in several high-tech software companies.

This document is a multi-dimensional presentation of the OCaml language that combines an informal and intuitive approach to the language with a rigorous definition and a formal semantics of a large subset of the language, including ML. All along this presentation, we explain the underlying design principles, highlight the numerous interactions between various facets of the language, and emphasize the close relationship between theory and practice.

Indeed, theory and practice should often cross their paths. Sometimes, the theory is deliberately weakened to keep the practice simple. Conversely, several related features may suggest a generalization and be merged, leading to a more expressive and regular design. We hope that the reader will follow us in this attempt of putting a little theory into practice or, conversely, of rebuilding bits of theory from practical examples and intuitions. However, we maintain that the underlying mathematics should always remain simple.

The introspection of OCaml is made even more meaningful by the fact that the language is boot-strapped, that is, its compilation chain is written in OCaml itself, and only parts of the runtime are written in C. Hence, some of the implementation notes, in particular those on typechecking, could be scaled up to be actually very close to the typechecker of OCaml itself.

The material presented here is divided into three categories. On the practical side, the course contains a short presentation of OCaml. Although this presentation is not at all exhaustive and certainly not a reference manual for the language, it is a self-contained introduction to the language: all facets of the language are covered; however, most of the details are omitted. A sample of programming exercises with different levels of difficulty have been included, and for most of them, solutions can be found in Appendix C. The knowledge and the practice of at least one dialect of ML may help getting the most from the other aspects. This is not mandatory though, and beginners can learn their first steps in OCaml by starting with Appendix A. Conversely, advanced OCaml programmers can learn from the inlined OCaml implementations of some of the algorithms. Implementation notes can always be skipped, at least in a first reading when the core of OCaml is not mastered yet —other parts never depend on them. However, we left implementation notes as well as some more advanced exercises inlined in the text to emphasize the closeness of the implementation to the formalization. Moreover, this permits to people who already know the OCaml language, to read all material continuously, making it altogether a more advanced course.

On the theoretical side —the mathematics remain rather elementary, we give a formal definition of a large subset of the OCaml language, including its dynamic and static semantics, and soundness results relating them. The proofs, however, are omitted. We also describe type inference in detail. Indeed, this is one of the most specific facets of ML.

A lot of the material actually lies in between theory and practice: we put an emphasis on the design principles, the modularity of the language constructs (their presentation is often incremental), as well as their dependencies. Some constructions that are theoretically independent end up being complementary in practice, so that one can hardly go without the other: it is often their combination that provides both flexibility and expressive power.

The document is organized in four parts (see the road maps in figure 1). Each of the first three parts addresses a different layer of OCaml:

the core language (Chapters 1 and 2), objects and classes (Chapter 3), and modules (Chapter 4); the last part (Chapter 5) focuses on the combination of objects and modules, and discusses a few perspectives. The style of presentation is different for each part. While the introduction of the core language is more formal and more complete, the emphasis is put on typechecking for the Chapter on objects and classes, the presentation of the modules system remains informal, and the last part is mostly based on examples. This is a deliberate choice, due to the limited space, but also based on the relative importance of the different parts and interest of their formalization. We then refer to other works for a more formal presentation or simply for further reading, both at the end of each Chapter for rather technical references, and at the end of the manuscript, Page 123 for a more general overview of related work.

This document is thus addressed to a wide audience. With several entry points, it can be read in parts or following different directions (see the road maps in figure 1). People interested in the semantics of programming languages may read Chapters 1 and 2 only. Conversely, people interested in the object-oriented layer of OCaml may skip these Chapters and start at Chapter 3. Beginners or people interested mostly in learning the programming language may start with appendix A, then grab examples and exercises in the first Chapters, and end with the Chapters on objects and modules; they can always come back to the first Chapters after mastering programming in OCaml, and attack the implementation of a typechecker as a project, either following or ignoring the relevant implementation notes.

Programming languages are rigorous but incomplete approximations of the language of mathematics. General purpose languages are Turing complete. That is, they allow to write all algorithms. (Thus, termination and many other useful properties of programs are undecidable.) However, programming languages are not all equivalent, since they differ by their ability to describe certain kinds of algorithms succinctly. This leads to an —endless?— research for new programming structures that are more expressive and allow shorter and safer descriptions of algorithms. Of course, expressiveness is not the ultimate goal. In particular, the safety of program execution should not be given up for expressiveness. We usually limit ourselves to a relatively small subset of programs that are well-typed and guaranteed to run safely. We also search for a small set of simple, essential, and orthogonal constructs.

Learning programming languages Learning a programming language is a combination of understanding the language constructs and practicing. Certainly, a programming language should have a clear semantics, whether it is given formally, *i.e.* using mathematical notation, as for Standard ML [51], or informally, using words, as for OCaml. Understanding the semantics and design principles, is a prerequisite to good programming habits, but good programming is also the result of practicing. Thus, using the manual, the tutorials, and on-line helps is normal practice. One may quickly learn all functions of the core library, but even fluent programmers may sometimes have to check specifications of some standard-library functions that are not so frequently used.

Copying (good) examples may save time at any stage of programming. This includes cut and paste from solutions to exercises, especially at the beginning. Sharing experience with others may also be helpful: the first problems you face are likely to be “Frequently Asked Questions” and the libraries you miss may already be available electronically in the “OCaml hump”. For books on ML see “Further reading”, Page 123.

A brief history of OCaml The current definition and implementation of the OCaml language is the result of continuous and still ongoing research over the last two decades. The OCaml language belongs to the ML family. The language ML was invented in 1975 by Robin Milner to serve as a “meta-language”, *i.e.* a control language or a scripting language, for programming proof-search strategies in the LCF proof assistant. The language quickly appeared to be a full-fledged programming language. The first implementations of ML were realized around 1981 in Lisp. Soon, several dialects of ML appeared: Standard ML at Edinburgh, Caml at INRIA, Standard ML of New-Jersey, Lazy ML developed at Chalmers, or Haskell at Glasgow. The two last dialects slightly differ from the previous ones by relying on a lazy evaluation strategy (they are called lazy languages) while all others have a strict evaluation strategy (and are called strict languages). Traditional languages, such as C, Pascal, Ada are also strict languages. Standard ML and Caml are relatively close to one another. The main differences are their implementations and their superficial —sometimes annoying— syntactic differences. Another minor difference is their module systems. However, SML does not have an object layer.

Continuing the history of Caml, Xavier Leroy and Damien Doligez designed a new implementation in 1990 called Caml-Light, freeing the previous implementation from too many experimental high-level features, and more importantly, from the old Le_Lisp back-end.

The addition of a native-code compiler and a powerful module system in 1995 and of the object and class layer in 1996 made OCaml a very mature and attractive programming language. The language is still under development: for instance, in 2000, labeled and optional arguments on the one hand and anonymous variants on the other hand were added to the language by Jacques Garrigue.

In the last decade, other dialects of ML have also evolved independently. Hereafter, we use the name ML to refer to features of the core language that are common to most dialects and we speak of OCaml, mostly in the examples, to refer to this particular implementation. Most of the examples, except those with object and classes, could easily be translated to Standard ML. However, only few of them could be straightforwardly translated to Haskell, mainly because of both languages have different evaluation strategy, but also due to many other differences in their designs.

Resemblances and differences in a few key words All dialects of ML are functional. That is, functions are taken seriously. In particular, they are first-class values: they can be arguments to other functions and returned as results. All dialects of ML are also strongly typed. This implies that well-typed programs cannot go wrong. By this, we mean that assuming no compiler bugs, programs will never execute erroneous access to memory nor other kind of abnormal execution step and programs that do not loop will always terminate normally. Of course, this does not ensure that the program executes what the programmer had in mind!

Another common property to all dialects of ML is type inference, that is, types of expressions are optional and are inferred by the system. As most modern languages, ML has automatic memory management, as well.

Additionally, the language OCaml is not purely functional: imperative programming with mutable values and side effects is also possible. OCaml is also object-oriented (aside from prototype designs, OCaml is still the only object-oriented dialect of ML). OCaml also features a powerful module system inspired by the one of Standard ML.

Acknowledgments

Many thanks to Jacques Garrigue, Xavier Leroy, and Brian Rogoff for their careful reading of parts of the notes.

Chapter 1

Core ML

We first present a few examples, insisting on the functional aspect of the language. Then, we formalize an extremely small subset of the language, which, surprisingly, contains in itself the essence of ML. Last, we show how to derive other constructs remaining in core ML whenever possible, or making small extensions when necessary.

1.1 Discovering Core ML

Core ML is a small *functional* language. This means that functions are taken seriously, *e.g.* they can be passed as arguments to other functions or returned as results. We also say that functions are *first-class* values.

In principle, the notion of a function relates as closely as possible to the one that can be found in mathematics. However, there are also important differences, because objects manipulated by programs are always countable (and finite in practice). In fact, core ML is based on the lambda-calculus, which has been invented by Church to model computation.

Syntactically, expressions of the lambda-calculus (written with letter a) are of three possible forms: variables x, which are given as elements of a countable set, functions $\lambda x.a$, or applications $a_1\ a_2$. In addition, core ML has a distinguished construction $\mathbf{let}\ x = a_1\ \mathbf{in}\ a_2$ used to bind an expression a_1 to a variable x within an expression a_2 (this construction is also used to introduce polymorphism, as we will see below). Furthermore, the language ML comes with primitive values, such as integers, floats, strings, *etc.* (written with letter c) and functions over these values.

Finally, a program is composed of a sequence of sentences that can optionally be separated by double semi-colon "`;;`". A sentence is a single expression or the binding, written **let** $x = a$, of an expression a to a variable x.

In normal mode, programs can be written in one or more files, separately compiled, and linked together to form an executable machine code (see Section 4.1.1). However, in the core language, we may assume that all sentences are written in a single file; furthermore, we may replace `;;` by `in` turning the sequence of sentences into a single expression. The language OCaml also offers an interactive loop in which sentences entered by the user are compiled and executed immediately; then, their results are printed on the terminal.

Note *We use the interactive mode to illustrate most of the examples. The input sentences are closed with a double semi-colons "*`;;`*". The output of the interpreter is only displayed when useful. Then, it appears in a smaller font and preceded by a double vertical bar "‖". Error messages may sometimes be verbose, thus we won't always display them in full. Instead, we use "✗" to mark an input sentence that will be rejected by the compiler. Some larger examples, called* implementation notes*, are delimited by horizontal braces as illustrated right below:*

Implementation notes, file `README`

Implementation notes are delimited as this one. They contain explanations in English (not in OCaml comments) and several OCaml phrases.

```
let readme = "lisez-moi";;
```

All phrases of a note belong to the same file (this one belong to `README`*) and are meant to be compiled (rather than interpreted).*

As an example, here are a couple of phrases evaluated in the interactive loop.

```
print_string "Hello\n";;
```

```
Hello
- : unit = ()
```

```
let pi = 4.0 *. atan 1.0;;
```

```
val pi : float = 3.141593
```

```
let square x = x *. x;;
```

```
val square : float -> float = <fun>
```

The execution of the first phrase prints the string `"Hello\n"` to the terminal. The system indicates that the result of the evaluation is of type `unit`. The evaluation of the second phrase binds the intermediate result of the evaluation of the expression `4.0 * atan 1.0`, that is the float `3.14...`, to the variable `pi`. This execution does not produce any output; the system only prints the type information and the value that is bound to `pi`. The last phrase defines a function that takes a parameter `x` and returns the product of `x` and itself. Because of the type of the binary primitive operation `*.`, which is `float -> float -> float`, the system infers that both `x` and the the result `square x` must be of type `float`. A mismatch between types, which often reveals a programmer's error, is detected and reported:

```
square "pi";;
```

```
Characters 7-11:
This expression has type string but is here used with type float
```

Function definitions may be recursive, provided this is requested explicitly, using the keyword `rec`:

```
let rec fib n = if n < 2 then 1 else fib(n-1) + fib(n-2);;
```

```
val fib : int -> int = <fun>
```

```
fib 10;;
```

```
- : int = 89
```

Functions can be passed to other functions as argument, or received as results, leading to higher-functions also called *functionals*. For instance, the composition of two functions can be defined exactly as in mathematics:

```
let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

The best illustration OCaml of the power of functions might be the function "power" itself!

```
let rec power f n =
  if n <= 0 then (fun x -> x) else compose f (power f (n−1));;
```

```
val power : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Here, the expression `(fun x -> x)` is the anonymous identity function. Extending the parallel with mathematics, we may define the derivative of an arbitrary function `f`. Since we use numerical rather than formal computation, the derivative is parameterized by the increment step `dx`:

```
let derivative dx f = function x -> (f(x +. dx) −. f(x)) /. dx;;
```

```
val derivative : float -> (float -> float) -> float -> float = <fun>
```

Then, the third derivative `sin'''` of the sinus function can be obtained by computing the cubic power of the derivative function and applying it to the sinus function. Last, we calculate its value for the real `pi`.

```
let sin''' = (power (derivative 1e−5) 3) sin in sin''' pi;;
```

```
− : float = 0.999999
```

This capability of functions to manipulate other functions as one would do in mathematics is almost unlimited... modulo the running time and the rounding errors.

1.2 The syntax of Core ML

Before continuing with more features of OCaml, let us see how a very simple subset of the language can be formalized.

In general, when giving a formal presentation of a language, we tend to keep the number of constructs small by factoring similar constructs as much as possible and explaining derived constructs by means of simple translations, such as syntactic sugar.

For instance, in the core language, we can omit phrases. That is, we transform sequences of bindings such as **let** $x_1 = a_1;;$ **let** $x_2 = a_2;; a$ into expressions of the form **let** $x_1 = a_1$ **in let** $x_2 = a_2$ **in** a. Similarly, numbers, strings, but also lists, pairs, *etc.* as well as operations on those values can all be treated as constants and applications of constants to values.

Formally, we assume a collection of constants $c \in \mathcal{C}$ that are partitioned into constructors $C \in \mathcal{C}^+$ and primitives $f \in \mathcal{C}^-$. Constants also come with an arity, that is, we assume a mapping *arity* from $\mathcal{C}$ to $I\!N$. For instance, integers and booleans are constructors of arity 0, pair is a constructor of arity 2, arithmetic operations, such as + or × are primitives of arity 2, and `not` is a primitive of arity 1. Intuitively, constructors are passive: they may take arguments, but should ignore their shape and simply build up larger values with their arguments embedded. On the opposite, primitives are active: they may examine the shape of their arguments, operate on inner embedded values, and transform them. This difference between constants and primitives will appear more clearly below, when we define their semantics. In summary, the syntax of expressions is given below:

$$a ::= \underbrace{x \mid \lambda x.a \mid a\ a}_{\lambda\text{-}calculus} \mid c \mid \textbf{let}\ x = a\ \textbf{in}\ a \qquad c ::= \underbrace{C}_{constructors} \mid \overbrace{f}^{primitives}$$

Implementation notes, file `syntax.ml`

Expressions can be represented in OCaml by their abstract-syntax trees, which are elements of the following data-type `expr`:

```
type name = Name of string | Int of int;;
type constant = { name : name; constr : bool; arity : int}
type var = string
type expr =
  | Var of var
  | Const of constant
  | Fun of var * expr
  | App of expr * expr
  | Let of var * expr * expr;;
```

For convenience, we define auxiliary functions to build constants.

```
let plus = Const {name = Name ”+”; arity = 2; constr = false}
let times = Const {name = Name ”*”; arity = 2; constr = false}
```

```
let int n = Const {name = Int n; arity = 0; constr = true};;
```

Here is a sample program.

```
let e =
  let plus_x n = App (App (plus, Var ”x”), n) in
  App (Fun (”x”, App (App (times, plus_x (int 1)), plus_x (int (−1)))),
       App (Fun (”x”, App (App (plus, Var ”x”), int 1)),
            int 2));;
```

Of course, a full implementation should also provide a lexer and a parser, so that the expression `e` could be entered using the concrete syntax $(\lambda x.x * x)\ ((\lambda x.x + 1)\ 2)$ and be automatically transformed into the abstract syntax tree above.

1.3 The dynamic semantics of Core ML

Giving the syntax of a programming language is a prerequisite to the definition of the language, but does not define the language itself. The syntax of a language describes the set of sentences that are well-formed expressions and programs that are acceptable inputs. However, the syntax of the language does not determine how these expressions are to be computed, nor what they *mean*. For that purpose, we need to define the *semantics* of the language.

(As a counter example, if one uses a sample of programs only as a pool of inputs to experiment with some pretty printing tool, it does not make sense to talk about the semantics of these programs.)

There are two main approaches to defining the semantics of programming languages: the simplest, more intuitive way is to give an *operational semantics*, which amounts to describing the computation process. It relates programs —as syntactic objects— between one another, closely following the evaluation steps. Usually, this models rather fairly the evaluation of programs on real computers. This level of description is both appropriate and convenient to prove properties about the evaluation, such as confluence or type soundness. However, it also contains many low-level details that makes other kinds of properties harder to prove.

This approach is somehow too concrete —it is sometimes said to be "too syntactic". In particular, it does not explain well what programs really are.

The alternative is to give a *denotational semantics* of programs. This amounts to building a mathematical structure whose objects, called *domains*, are used to represent the meanings of programs: every program is then mapped to one of these objects. The denotational semantics is much more abstract. In principle, it should not use any reference to the syntax of programs, not even to their evaluation process. However, it is often difficult to build the mathematical domains that are used as the meanings of programs. In return, this semantics may allow to prove difficult properties in an extremely concise way.

The denotational and operational approaches to semantics are actually complementary. Hereafter, we only consider operational semantics, because we will focus on the evaluation process and its correctness.

In general, operational semantics relates programs to answers describing the result of their evaluation. Values are the subset of answers expected from *normal* evaluations.

A particular case of operational semantics is called a *reduction* semantics. Here, answers are a subset of programs and the semantic relation is defined as the transitive closure of a small-step internal binary relation (called reduction) between programs.

The latter is often called *small*-step style of operational semantics, sometimes also called Structural Operational Semantics [61]. The former is *big-step* style, sometimes also called Natural Semantics [39].

1.3.1 Reduction semantics

The call-by-value reduction semantics for ML is defined as follows: values are either functions, constructed values, or partially applied constants; a constructed value is a constructor applied to as many values as the arity of the constructor; a partially applied constant is either a primitive or a constructor applied to fewer values than the arity of the constant. This is summarized below, writing v for values:

$$v ::= \lambda x.a \mid \underbrace{C^n\ v_1 \ldots v_n}_{\substack{\text{Constructed}\\ \text{values}}} \mid \underbrace{c^n\ v_1 \ldots v_k}_{\substack{\text{Partially applied}\\ \text{constants}}} \qquad k < n$$

In fact, a partially applied constant $c^n\ v_1 \ldots v_k$ behaves as the function $\lambda x_{k+1}. \ldots \lambda x_n.c^k\ v_1 \ldots\ v_k\ x_{k+1}\ \ldots\ x_n$, with $k < n$. Indeed, it is a value.

Implementation notes, file `reduce.ml`

Since values are subsets of programs, they can be characterized by a predicate `evaluated` defined on expressions:

```
let rec evaluated = function
    Fun (_,_) -> true
  | u -> partial_application 0 u
and partial_application n = function
    Const c -> (c.constr || c.arity > n)
  | App (u, v) -> (evaluated v && partial_application (n+1) u)
  | _ -> false;;
```

The small-step reduction is defined by a set of *redexes* and is closed by congruence with respect to *evaluations contexts*.

Redexes describe the reduction at the place where it occurs; they are the heart of the reduction semantics:

$$\begin{array}{rcll} (\lambda x.a)\ v & \longrightarrow & a[v/x] & (\beta_v) \\ \textbf{let } x = v \textbf{ in } a & \longrightarrow & a[v/x] & (Let_v) \\ f^n\ v_1\ \ldots v_n & \longrightarrow & a & (f^n\ v_1\ \ldots v_n, a) \in \delta_f \end{array}$$

Redexes of the latter form, which describe how to reduce primitives, are also called *delta rules*. We write δ for the union $\bigcup_{f \in \mathcal{C}^-}(\delta_f)$. For instance, the rule (δ_+) is the relation $\{(\overline{p} + \overline{q}, \overline{p+q}) \mid p, q \in I\!N\}$ where $\overline{n}$ is the constant representing the integer n.

Implementation notes, file `reduce.ml`

Redexes are partial functions from programs to programs. Hence, they can be represented as OCaml functions, raising an exception `Reduce` when there are applied to values outside of their domain. The δ-rules can be implemented straightforwardly.

```
exception Reduce;;
let delta_bin_arith op code = function
  | App (App (Const { name = Name _; arity = 2} as c,
              Const { name = Int x }), Const { name = Int y })
    when c = op -> int (code x y)
  | _ -> raise Reduce;;
let delta_plus = delta_bin_arith plus ( + );;
let delta_times = delta_bin_arith times ( * );;
let delta_rules = [ delta_plus; delta_times ];;
```

The union of partial function (with priority on the right) is

```
let union f g a = try g a with Reduce -> f a;;
```

The δ-reduction is thus:

```
let delta =
  List.fold_right union delta_rules (fun _ -> raise Reduce);;
```

To implement (β_v), we first need an auxiliary function that substitutes a variable for a value in a term. Since the expression to be substituted will always be a value, hence closed, we do not have to perform α-conversion to avoid variable capture.

```
let rec subst x v a =
  assert (evaluated v);
  match a with
  | Var y ->
      if x = y then v else a
  | Fun (y, a’) ->
      if x = y then a else Fun (y, subst x v a’)
  | App (a’, a’’) ->
      App (subst x v a’, subst x v a’’)
  | Let (y, a’, a’’) ->
      if x = y then Let (y, subst x v a’, a’’)
      else Let (y, subst x v a’, subst x v a’’)
  | Const c -> Const c;;
```

Then `beta` is straightforward:

```
let beta = function
```

```
    | App (Fun (x,a), v) when evaluated v -> subst x v a
    | Let (x, v, a) when evaluated v -> subst x v a
    | _ -> raise Reduce;;
```

Finally, top reduction is

```
  let top_reduction = union beta delta;;
```

The evaluation contexts E describe the occurrences inside programs where the reduction may actually occur. In general, a (one-hole) *context* is an expression with a hole —which can be seen as a distinguished constant, written $[\cdot]$— occurring exactly once. For instance, $\lambda x.x\ [\cdot]$ is a context. Evaluation contexts are contexts where the hole can only occur at some admissible positions that often described by a grammar. For ML, the (call-by-value) evaluation contexts are:

$$E ::= [\cdot] \mid E\ a \mid v\ E \mid \textbf{let}\ x = E\ \textbf{in}\ a$$

We write $E[a]$ the term obtained by filling the expression a in the evaluation context E (or in other words by replacing the constant $[\cdot]$ by the expression a).

Finally, the small-step reduction is the closure of redexes by the congruence rule:

$$\text{if } a \longrightarrow a' \text{ then } E[a] \longrightarrow E[a'].$$

The evaluation relation is then the transitive closure $\xrightarrow{\star}$ of the small step reduction $\longrightarrow$. Note that values are irreducible, indeed.

Implementation notes, file `reduce.ml`

There are several ways to treat evaluation contexts in practice. The most standard solution is not to represent them, *i.e.* to represent them as evaluation contexts of the host language, using its run-time stack. Typically, an evaluator would be defined as follows:

```
  let rec eval =
    let eval_top_reduce a = try eval (top_reduction a) with Reduce -> a in
    function
    | App (a1, a2) ->
        let v1 = eval a1 in
```

```
        let v2 = eval a2 in
        eval_top_reduce (App (v1, v2))
    | Let (x, a1, a2) ->
        let v1 = eval a1 in
        eval_top_reduce (Let (x, v1, a2))
    | a ->
        eval_top_reduce a;;
  let _ = eval e;;
```

The function `eval` visits the tree top-down. On the descent it evaluates all subterms that are not values in the order prescribed by the evaluation contexts; before ascent, it replaces subtrees bu their evaluated forms. If this succeeds it recursively evaluates the reduct; otherwise, it simply returns the resulting expression.

This algorithm is efficient, since the input term is scanned only once, from the root to the leaves, and reduced from the leaves to the root. However, this optimized implementation is not a straightforward implementation of the reduction semantics.

If efficiency is not an issue, the step-by-step reduction can be recovered by a slight change to this algorithm, stopping reduction after each step.

```
  let rec eval_step = function
    | App (a1, a2) when not (evaluated a1) ->
        App (eval_step a1, a2)
    | App (a1, a2) when not (evaluated a2) ->
        App (a1, eval_step a2)
    | Let (x, a1, a2) when not (evaluated a1) ->
        Let (x, eval_step a1, a2)
    | a -> top_reduction a;;
```

Here, contexts are still implicit, and redexes are immediately reduced and put back into their evaluation context. However, the `eval_step` function can easily be decomposed into three operations: `eval_context` that returns an evaluation context and a term, the reduction per say, and the reconstruction of the result by filling the result of the reduction back into the evaluation context. The simplest representation of contexts is to view them as functions form terms to terms as follows:

```
type context = expr -> expr;;
let hole : context = fun t -> t;;
let appL a t = App (t, a)
let appR a t = App (a, t)
let letL x a t = Let (x, t, a)
let ( ** ) e1 (e0, a0) = (fun a -> e1 (e0 a)), a0;;
```

Then, the following function split a term into a pair of an evaluation context and a term.

```
let rec eval_context : expr -> context * expr = function
  | App (a1, a2) when not (evaluated a1) ->
     appL a2 ** eval_context a1
  | App (a1, a2) when not (evaluated a2) ->
     appR a1 ** eval_context a2
  | Let (x, a1, a2) when not (evaluated a1) ->
     letL x a2 ** eval_context a1
  | a -> hole, a;;
```

Finally, it the one-step reduction rewrites the term as a pair $E[a]$ of an evaluation context E and a term t, apply top reduces the term a to a', and returns $E[a]$, exactly as the formal specification.

```
let eval_step a = let c, t = eval_context a in c (top_reduction t);;
```

The reduction function is obtain from the one-step reduction by iterating the process until no more reduction applies.

```
let rec eval a = try eval (eval_step a) with Reduce -> a ;;
```

This implementation of reduction closely follows the formal definition. Of course, it is less efficient the direct implementation. Exercise 1 presents yet another solution that combines small step reduction with an efficient implementation.

Remark 1 *The following rule could be taken as an alternative for* (Let_v).

$$\textbf{let } x = v \textbf{ in } a \longrightarrow (\lambda x.a)\ v$$

Observe that the right hand side can then be reduced to $a[v/x]$ *by* (β_v). *We chose the direct form, because in ML, the intermediate form would not necessarily be well-typed.*

Example 1 The expression $(\lambda x.(x * x))\ ((\lambda x.(x+1))\ 2)$ is reduced to the value 9 as follows (we underline the sub-term to be reduced):

$$\begin{array}{rlr}
 & (\lambda x.(x * x))\ (\underline{(\lambda x.(x+1)))\ 2}) & \\
\longrightarrow & (\lambda x.(x * x))\ (\underline{2+1}) & (\beta_v) \\
\longrightarrow & \underline{(\lambda x.(x * x))\ 3} & (\delta_+) \\
\longrightarrow & \underline{(3 * 3)} & (\beta_v) \\
\longrightarrow & 9 & (\delta_*)
\end{array}$$

We can check this example by running it through the evaluator:

```
eval e;;
```

```
- : expr = Const {name=Int 9; constr=true; arity=0}
```

Exercise 1 ((**) Representing evaluation contexts) *Evaluation contexts are not explicitly represented above. Instead, they are left implicit from the runtime stack and functions from terms to terms. In this exercise, we represent evaluation contexts explicitly into a dedicated data-structure, which enables to examined them by pattern matching.*

In fact, it is more convenient to hold contexts by their hole—where reduction happens. To this aim, we represent them upside-down, following Huet's notion of zippers [32]. Zippers are a systematic and efficient way of representing every step while walking along a tree. Informally, the zipper is closed when at the top of the tree; walking down the tree will open up the top of the zipper, turning the top of the tree into backward-pointers so that the tree can be rebuilt when walking back up, after some of the subtrees might have been changed.

Actually, the zipper definition can be read from the formal BNF definition of evaluations contexts:

$$E ::= [\cdot] \mid E\ a \mid v\ E \mid \textbf{let}\ x = E\ \textbf{in}\ a$$

The OCaml definition is:

```
type context =
  | Top
  | AppL of context * expr
  | AppR of value * context
```

```
  | LetL of string * context * expr
and value = int * expr
```

The left argument of constructor `AppR` *is always a value. A value is a expression of a certain form. However, the type system cannot enfore this invariant. For sake of efficiency, values also carry their arity, which is the number of arguments a value must be applied to before any reduction may occur. For instance, a constant of arity k is a value of arity k. A function is a value of arity* 1*. Hence, a fully applied contructor such as* 1 *will be given an strictly positive arity,* e.g. *1.*

Note that the type context is linear, in the sense that constructors have at more one context subterm. This leads to two opposite representations of contexts. The naive representation of context **let** $x = [\cdot]\ a_2$ **in** $a3$ *is* `LetL (x, AppL (Top, a2)), a3)`*. However, we shall represent them upside-down by the term* `AppL (LetL (x, Top, a3), a2)`*, following the idea of zippers —this justifies our choice of* `Top` *rather than* `Hole` *for the empty context. This should read “*a context where the hole is below the left branch of an application node whose right branch is a_3 and which is itself (the left branch of) a binding of x whose body is a_2 and which is itself at the top*”.*

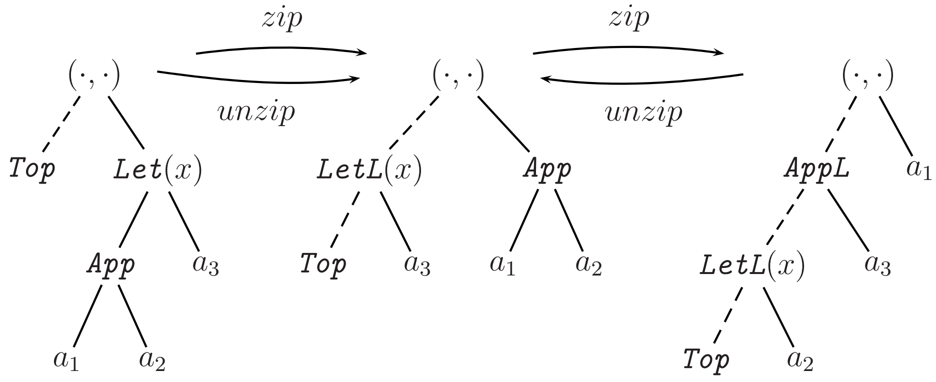
A term a_0 can usually be decomposed as a one hole context $E[a]$ in many ways if we do not impose that a is a reducible. For instance, taking $(a_1\ a_2)\ a_3$, allows the following decompositions

$$\begin{array}{cc} [\cdot][\textbf{let } x = a_1\ a_2 \textbf{ in } a_3] & (\textbf{let } x = [\cdot] \textbf{ in } a_3)[a_1\ a_2] \\ (\textbf{let } x = [\cdot]\ a_2 \textbf{ in } a_3)[a_1] & (\textbf{let } x = a_1\ [\cdot] \textbf{ in } a_3)[a_2] \end{array}$$

(The last decompistion is correct only when a_1 is a value.) These decompositions *can be described by a pair whose left-hand side is the context and whose right-hand side is the term to be placed in the hole of the context:*

```
Top    ,                              Let (x, App (a1, a2), a3)
LetL (x, Top ,          a3  ),        App (a1, a2)
AppL (LetL (Top, a2),   a3 ),         a1
AppR ((k, a1), LetL (Top, a3))    a2
```

They can also be represented graphically:



As shown in the graph, the different decompositions can be obtained by zipping (push some of the term structure inside the context) or unzipping (popping the structure from the context back to the term). This allows a simple change of focus, and efficient exploration and transformation of the region (both up the context and down the term) at the junction.

Give a program `context_fill` *of type* `context * expr -> expr` *that takes a decomposition* (E, a) *and returns the expression* $E[a]$*.* Answer
Define a function `decompose_down` *of type* `context * expr -> context * expr` *that given a decomposition* (E, a) *searches for a sub-context* E' *of* E *in evaluation position and the residual term* a' *at that position and returns the decomposition* $E[E'[\cdot]], a'$ *or it raises the exception* `Value` k *if* a *is a value of arity* k *in evaluation position or the exception* `Error` *if* a *is an error (irreducible but not a value) in evaluation position.* Answer
Starting with (Top, a)*, we may find the first position* (E_0, a_0) *where reduction may occur and then top-reduce* a_0 *into* a'_0*. After reduction, one wish to find the next evaluation position, say* (E_n, a_n) *given* (E_{n-1}, a'_{n-1}) *and knowing that* E_{n-1} *is evaluation context but* a'_{n-1} *may know be a value.*

Define an auxilliary function `decompose_up` *that takes an integer* k *and a decomposition* (c, v) *where* v *is a value of arity* k *and find a decomposition of* $c[v]$ *or raises the exception* `Not_found` *when non exists. The integer* k *represents the number of left applications that may be blindly unfolded before decomposing down.* Answer
Define a function `decompose` *that takes a context pair* (E, a) *and finds a*

decomposition of $E[a]$. It raises the exception `Not_found` *if no decomposition exists and the exception* `Error` *if an irreducible term is found in evaluation position.* Answer

Finally, define the `eval_step` *reduction, and check the evaluation steps of the program* `e` *given above and recover the function* `reduce` *of type* `expr -> expr` *that reduces an expression to a value.* Answer
Write a pretty printer for expressions and contexts, and use it to trace evaluation steps, automatically. Answer
Then, it suffices to use the OCaml toplevel tracing capability for functions `decompose` *and* `reduce_in` *to obtain a trace of evaluation steps (in fact, since the result of one function is immediately passed to the other, it suffices to trace one of them, or to skip output of traces).*

```
#trace decompose;;
#trace reduce_in;;
let _ = eval e;;
```

```
decompose ← [(fun x -> (x + 1) * (x + -1)) ((fun x -> x + 1) 2)]
reduce_in  ← (fun x -> (x + 1) * (x + -1)) [(fun x -> x + 1) 2]
decompose ← (fun x -> (x + 1) * (x + -1)) [2 + 1]
reduce_in  ← (fun x -> (x + 1) * (x + -1)) [2 + 1]
decompose ← (fun x -> (x + 1) * (x + -1)) [3]
reduce_in  ← [(fun x -> (x + 1) * (x + -1)) 3]
decompose ← [(3 + 1) * (3 + -1)]
reduce_in  ← [3 + 1] * (3 + -1)
decompose ← [4] * (3 + -1)
reduce_in  ← 4 * [3 + -1]
decompose ← 4 * [2]
reduce_in  ← [4 * 2]
decompose ← [8]
raises Not_found
- : expr = Const {name = Int 8; constr = true; arity = 0}
```

□

1.3.2 Properties of the reduction

The strategy we gave is *call-by-value*: the rule (β_v) only applies when the argument of the application has been reduced to value. Another simple reduction strategy is *call-by-name*. Here, applications are reduced before the arguments. To obtain a call-by-name strategy, rules (β_v) and (Let_v) need to be replaced by more general versions that allows the arguments to be arbitrary expressions (in this case, the substitution operation must carefully avoid variable capture).

$$\begin{array}{rcll} (\lambda x.a)\ a' & \longrightarrow & a[a'/x] & (\beta_n) \\ \mathbf{let}\ x = a'\ \mathbf{in}\ a & \longrightarrow & a[a'/x] & (Let_n) \end{array}$$

Simultaneously, we must restrict evaluation contexts to prevent reductions of the arguments before the reduction of the application itself; actually, it suffices to remove $v\ E$ and $\mathbf{let}\ x = E\ \mathbf{in}\ a$ from evaluations contexts.

$$E_n ::= [\cdot] \mid E_n\ a$$

There is, however, a slight difficulty: the above definition of evaluation contexts does not work for constants, since δ-rules expect their arguments to be reduced. If all primitives are strict in their arguments, their arguments could still be evaluated first, then we can add the following evaluations contexts:

$$E_n ::= \ldots \mid (f^n\ v_1\ \ldots v_{k-1}\ E_k\ a_{k+1}\ ...a_n)$$

However, in a call-by-name semantics, one may wish to have constants such as `fst` that only forces the evaluation of the top-structure of the terms. This is is slightly more difficult to model.

Example 2 The call-by-name reduction of the example 1 where all primitives are strict is as follows:

$$\begin{array}{rll} & \underline{(\lambda x.x * x)\ ((\lambda x.(x+1))\ 2)} & \\ \longrightarrow & \underline{((\lambda x.(x+1))\ 2)} * ((\lambda x.(x+1))\ 2) & (\beta_n) \\ \longrightarrow & \underline{(2+1)} * ((\lambda x.(x+1))\ 2) & (\beta_n) \\ \longrightarrow & 3 * \underline{((\lambda x.(x+1))\ 2)} & (\delta_+) \\ \longrightarrow & 3 * \underline{(2+1)} & (\beta_n) \\ \longrightarrow & \underline{3 * 3} & (\delta_+) \\ \longrightarrow & 9 & (\delta_*) \end{array}$$

As illustrated in this example, call-by-name may duplicate some computations. As a result, it is not often used in programming languages. Instead, Haskell and other lazy languages use a *call-by-need* or *lazy* evaluation strategy: as with call-by-name, arguments are not evaluated prior to applications, and, as with call-by-value, the evaluation is shared between all uses of the same argument. However, call-by-need semantics are slightly more complicated to formalize than call-by-value and call-by-name, because of the formalization of sharing. They are quite simple to implement though, using a reference to ensure sharing and closures to delay evaluations until they are really needed. Then, the closure contained in the reference is evaluated and the result is stored in the reference for further uses of the argument.

Classifying evaluations of programs Remark that the call-by-value evaluation that we have defined is deterministic by construction. According to the definition of the evaluation contexts, there is at most one evaluation context E such that a is of the form $E[a']$. So, if the evaluation of a program a reaches program $a\dagger$, then there is a unique sequence $a = a_0 \longrightarrow a_1 \longrightarrow \ldots a_n = a\dagger$. Reduction may become non-deterministic by a simple change in the definition of evaluation contexts. (For instance, taking all possible contexts as evaluations context would allow the reduction to occur anywhere.)

Moreover, reduction may be left non-deterministic on purpose; this is usually done to ease compiler optimizations, but at the expense of semantic ambiguities that the programmer must then carefully avoid. That is, when the order of evaluation does matter, the programmer has to use a construction that enforces the evaluation in the right order.

In OCaml, for instance, the relation is non-deterministic: the order of evaluation of an application is not specified, *i.e.* the evaluation contexts are:

$$\underbrace{E ::= [\cdot] \mid E\ a \mid a\ \underset{\Uparrow}{E} \mid \mathbf{let}\ x = E\ \mathbf{in}\ a}_{\text{Evaluation is possible even if } a \text{ is not reduced}}$$

When the reduction is not deterministic, the result of evaluation may still be deterministic if the reduction is *Church-Rosser*. A reduction relation has the Church-Rosser property, if for any expression a that reduces both to a' or a'' (following different branches) there exists an expression a''' such that both a' and a'' can in turn be reduced to a'''. (However, if the language has side effects, Church Rosser property will very unlikely be satisfied).

For the (deterministic) call-by-value semantics of ML, the evaluation of a program a can follow one of the following patterns:

$$a \longrightarrow a_1 \longrightarrow \ldots \begin{cases} a_n \equiv v & \text{normal evaluation} \\ a_n \not\longrightarrow \wedge\, a_n \not\equiv v & \text{run-time error} \\ a_n \longrightarrow \ldots & \text{loop} \end{cases}$$

Normal evaluation terminates, and the result is a value. Erroneous evaluation also terminates, but the result is an expression that is not a value. This models the situation when the evaluator would abort in the middle of the evaluation of a program. Last, evaluation may also proceed forever.

The type system will prevent run-time errors. That is, evaluation of well-typed programs will never get "stuck". However, the type system will not prevent programs from looping. Indeed, for a general purpose language to be interesting, it must be Turing complete, and as a result the termination problem for admissible programs cannot be decidable. Moreover, some non-terminating programs are in fact quite useful. For example, an operating system is a program that should run forever, and one is usually unhappy when it terminates —by accident.

Implementation notes

In the evaluator, errors can be observed as being irreducible programs that are not values. For instance, we can check that `e` evaluates to a value, while $(\lambda x.y)$ 1 does not reduce to a value.

```
evaluated (eval e);;
evaluated (eval (App (Fun ("x", Var "y"), int 1)));;
```

Conversely, termination cannot be observed. (One can only suspect non-termination.)

1.3.3 Big-step operational semantics

The advantage of the reduction semantics is its conciseness and modularity. However, one drawback of is its limitation to cases where values are a subset of programs. In some cases, it is simpler to let values differ from programs. In such cases, the reduction semantics does not make sense, and one must relates programs to answers in a simple "big" step.

A typical example of use of big-step semantics is when programs are evaluated in an environment e that binds variables (*e.g.* free variables occurring in the term to be evaluated) to values. Hence the evaluation relation is a triple $\rho \Vdash a \Rightarrow r$ that should be read "In the evaluation environment e the program a evaluates to the answer r."

Values are partially applied constants, totally applied constructors as before, or closures. A closure is a pair written $\langle \lambda x.a, e \rangle$ of a function and an environment (in which the function should be executed). Finally, answers are values or plus a distinguished answer `error`.

$$
\begin{aligned}
\rho &::= \emptyset \mid \rho, x \mapsto v \\
v &::= \langle \lambda x.a, \rho \rangle \mid \underbrace{C^n\ v_1 \ldots v_n}_{\substack{\text{Constructed} \\ \text{values}}} \mid \underbrace{c^n\ v_1 \ldots v_k}_{\substack{\text{Partially applied} \\ \text{constants}}} \qquad k < n \\
r &::= v \mid \texttt{error}
\end{aligned}
$$

The big-step evaluation relation (natural semantics) is often described via inference rules.

An inference rule written $\dfrac{P_1 \quad \ldots \quad P_n}{C}$ is composed of premises P_1, $\ldots P_n$ and a conclusion C and should be read as the implication: $P_1 \wedge \ldots P_n \Longrightarrow C$; the set of premises may be empty, in which case the inference rule is an axiom C.

The inference rules for the big-step operational semantics of Core ML are described in figure 1.1. For simplicity, we give only the rules for constants of arity 1. As for the reduction, we assume given an evaluation relation for primitives.

Rules can be classified into 3 categories:

- Proper evaluation rules: *e.g.* Eval-Fun, Eval-App, describe the evaluation process itself.

Figure 1.1: Big step reduction rules for Core ML

Eval-Const
$$\frac{\rho \Vdash a \Rightarrow v}{\rho \Vdash C^1\ a \Rightarrow C^1\ v}$$

Eval-Const-Error
$$\frac{\rho \Vdash a \Rightarrow \texttt{error}}{\rho \Vdash c\ a \Rightarrow c\ \texttt{error}}$$

Eval-Prim
$$\frac{\rho \Vdash a \Rightarrow v \qquad f^1\ v \longrightarrow v'}{\rho \Vdash f^1\ a \Rightarrow v'}$$

Eval-Prim-Error
$$\frac{\rho \Vdash a \Rightarrow v \qquad f^1\ v \not\longrightarrow v'}{\rho \Vdash f^1\ a \Rightarrow \texttt{error}}$$

Eval-Var
$$\frac{z \in \mathit{dom}\,(\rho)}{\rho \Vdash z \Rightarrow \rho(v)}$$

Eval-Fun
$$\frac{}{e \vdash \lambda x.a \Rightarrow \langle \lambda x.a, \rho \rangle}$$

Eval-App
$$\frac{\rho \vdash a \Rightarrow \langle \lambda x.a_0, \rho_0 \rangle \qquad \rho \vdash a' \Rightarrow v \qquad \rho_0, x \mapsto v \vdash a_0 : v'}{\rho \vdash a\ a' \Rightarrow v'}$$

Eval-App-Error
$$\frac{\rho \vdash a \Rightarrow C_1\ v_1}{\rho \vdash a\ a' \Rightarrow \texttt{error}}$$

Eval-App-Error-Left
$$\frac{\rho \vdash a \Rightarrow \texttt{error}}{\rho \vdash a\ a' \Rightarrow \texttt{error}}$$

Eval-App-Error-Right
$$\frac{\rho \vdash a \Rightarrow \langle \lambda x.a_0, \rho_0 \rangle \qquad \rho \vdash a' \Rightarrow \texttt{error}}{\rho \vdash a\ a' \Rightarrow \texttt{error}}$$

Eval-Let
$$\frac{\rho \vdash a \Rightarrow v \qquad \rho, x \mapsto v \vdash a' \Rightarrow v'}{\rho \vdash \mathbf{let}\ x = a\ \mathbf{in}\ a' \Rightarrow v'}$$

Eval-Let-Error
$$\frac{\rho \vdash a \Rightarrow \texttt{error}}{\rho \vdash \mathbf{let}\ x = a\ \mathbf{in}\ a' \Rightarrow \texttt{error}}$$

- Error rules: *e.g.* Eval-App-Error describe ill-formed computations.
- Error propagation rules: Eval-App-Left, Eval-App-Right describe the propagation of errors.

Note that error propagation rules play an important role, since they define the evaluation strategy. For instance, the combination of rules Eval-App-Error-Left and Eval-App-Error-Right states that the function must be evaluated before the argument in an application. Thus, the burden of writing error rules cannot be avoided. As a result, the big-step operation semantics is much more verbose than the small-step one. In fact, big-step style fails to share common patterns: for instance, the reduction of the evaluation of the arguments of constants and of the arguments of functions are similar, but they must be duplicated because the intermediate state $v_1\ v_2$ is not well-formed —it is not yet value, but no more an expression!

Another problem with the big-step operational semantics is that it cannot describe properties of diverging programs, for which there is not v such that $\rho \Vdash a \Rightarrow v$. Furthermore, this situation is not a characteristic of diverging programs, since it could result from missing error rules.

The usual solution is to complement the evaluation relation by a diverging predicate $\rho \Vdash a \Uparrow$.

Implementation notes

The big-step evaluation semantics suggests another more direct implementation of an interpreter.

```
type env = (string * value) list
and value =
  | Closure of var * expr * env
  | Constant of constant * value list
```

To keep closer to the evaluation rules, we represent errors explicitly using the following `answer` datatype. In practice, one would take avantage of exceptions making `value` be the default answer and `Error` be an exception instead. The construction `Error` would also take an argument to report the cause of error.

```
type answer = Error | Value of value;;
```

Next comes delta rules, which abstract over the set of primitives.

```
let val_int u =
  Value (Constant ({name = Int u; arity = 0; constr = true}, []));;
let delta c l =
  match c.name, l with
  | Name ”+”, [ Constant ({name=Int u}, []); Constant ({name=Int v}, [])] ->
      val_int (u + v)
  | Name ”∗”, [ Constant ({name=Int u}, []); Constant ({name=Int v}, [])] ->
        val_int (u ∗ v)
  | _ ->
      Error;;
```

Finally, the core of the evaluation.

```
let get x env =
  try Value (List.assoc x env) with Not_found -> Error;;
let rec eval env = function
  | Var x -> get x env
  | Const c -> Value (Constant (c, []))
  | Fun (x, a) -> Value (Closure (x, a, env))
  | Let (x, a1, a2) ->
      begin match eval env a1 with
      | Value v1 -> eval ((x, v1)::env) a2
      | Error -> Error
      end
  | App (a1, a2) ->
      begin match eval env a1 with
      | Value v1 ->
          begin match v1, eval env a2 with
          | Constant (c, l), Value v2 ->
              let k = List.length l + 1 in
              if c.arity < k then Error
              else if c.arity > k then Value (Constant (c, v2::l))
              else if c.constr then Value (Constant (c, v2::l))
              else delta c (v2::l)
          | Closure (x, e, env0), Value v2 ->
              eval ((x, v2 ) :: env0) e
```

```
          | _, Error -> Error
           end
      | Error -> Error
       end
  | _ -> Error ;;
```

Note that treatment of errors in the big-step semantics explicitly specifies a left-to-right evaluation order, which we have carefully reflected in the implementation. (In particular, if a_1 diverges and a_2 evaluates to an error, then a_1 a_2 diverges.)

```
eval [] e ;;
```

```
- : answer =
 Value (Constant ({name = Int 9; constr = true; arity = 0}, []))
```

While the big-step semantics is less interesting (because less precise) than the small-steps semantics in theory, its implementation is intuitive, simple and lead to very efficient code.

This seems to be a counter-example of practice meeting theory, but actually it is not: the big-step implementation could also be seen as efficient implementation of the small-step semantics obtained by (very aggressive) program transformations.

Also, the non modularity of the big-step semantics remains a serious drawback in practice. In conclusion, although the most commonly preferred the big-step semantics is not always the best choice in practice.

1.4 The static semantics of Core ML

We start with the less expressive but simpler static semantics called *simple types*. We present the typing rules, explain type inference, unification, and only then we shall introduce polymorphism. We close this section with a discussion about recursion.

1.4.1 Types and programs

Expressions of Core ML are untyped —they do not mention types. However, as we have seen, some expressions do not make sense. These are expressions that after a finite number of reduction steps would be stuck, *i.e.* irreducible while not being a value. This happens, for instance when a constant of arity 0, say integer 2, is applied, say to 1. To prevent this situation from happening one must rule out not only stuck programs, but also all programs reducing to stuck programs, that is a large class of programs. Since deciding whether a program could get stuck during evaluation is equivalent to evaluation itself, which is undecidable, to be safe, one must accept to also rule out other programs that would behave correctly.

Exercise 2 ((*) Progress in lambda-calculus) *Show that, in the absence of constants, programs of Core ML without free variables (*i.e. *lambda-calculus) are never stuck.* ☐

Types are a powerful tool to classify programs such that well-typed programs cannot get stuck during evaluations. Intuitively, types abstract over from the internal behavior of expressions, remembering only the shape (types) of other expression (integers, booleans, functions from integers to integers, etc.), that can be passed to them as arguments or returned as results.

We assume given a denumerable set of type symbols $g \in \mathcal{G}$. Each symbol should be given with a fixed arity. We write g^n to mean that g is of arity n, but we often leave the arity implicit. The set of types is defined by the following grammar.

$$\tau ::= \alpha \mid g^n(\tau_1, \ldots \tau_n)$$

Indeed, functional types, *i.e.* the type of functions play a crucial role. Thus, we assume that there is a distinguished type symbol of arity 2, the right arrow "$\rightarrow$" in $\mathcal{G}$; we also write $\tau \rightarrow \tau'$ for $\rightarrow (\tau, \tau')$. We write $ftv(\tau)$ the set of type variables occurring in τ.

Types of programs are given under typing assumptions, also called *typing environments*, which are partial mappings from program variables

Figure 1.2: Summary of types, typing environments and judgments

Types	$\tau ::= \alpha \mid \tau \to \tau \mid g^n(\tau_1, \ldots \tau_n)$
Typing environments	$A ::= \emptyset \mid A, z : \tau$ $z ::= x \mid c$
Typing judgments	$A \vdash a : \tau$

Figure 1.3: Typing rules for simple types

$$\textsc{Var-Const}\quad \frac{z \in dom\,(A)}{A \vdash x : A(z)} \qquad \textsc{Fun}\quad \frac{A, x : \tau \vdash a : \tau'}{A \vdash \lambda x.a : \tau \to \tau'} \qquad \textsc{App}\quad \frac{A \vdash a : \tau' \to \tau \qquad A \vdash a' : \tau'}{A \vdash a\ a' : \tau}$$

and constants to types. We use letter z for either a variable x or a constant c. We write $\emptyset$ for the empty typing environment and $A, x : \tau$ for the function that behaves as A except for x that is mapped to τ (whether or not x is in the domain of A). We also assume given an environment A_0 that assigns types to constants. The typing of programs is represented by a ternary relation, written $A \vdash a : \tau$ and called *typing judgments*, between type environments A, programs a, and types τ. We summarize all these definitions (expanding the arrow types) in figure 1.2.

Typing judgments are defined as the smallest relation satisfying the inference rules of figure 1.3. (See 1.3.3 for an introduction to inference rules)

Closed programs are typed the initial environment A_0. Of course, we must assume that the type assumptions for constants are consistent with their arities. This is the following asumption.

Assumption 0 (Initial environment) *The initial type environment A_0 has the set of constants for domain, and respects arities. That is, for any $C^n \in dom\,(A_0)$ then $A_0(C^n)$ is of the form $\tau_1 \to \ldots \tau_n \to \tau_0$.*

Type soundness asserts that well-typed programs cannot go wrong. This actually results from two stronger properties, that (1) reduction preserves typings, and (2) well-typed programs that are not values can be further reduced. Of course, those results can be only proved if the types of constants and their semantics (*i.e.* their associated delta-rules) are chosen accordingly.

To formalize soundness properties it is convenient to define a relation $\sqsubseteq$ on programs to mean the preservation of typings:

$$(a \sqsubseteq a') \iff \forall(A, \tau)(A \vdash a : \tau \Longrightarrow A \vdash a' : \tau)$$

The relation $\sqsubseteq$ relates the set of typings of two programs programs, regardless of their dynamic properties.

The preservation of typings can then be stated as $\sqsubseteq$ being a smaller relation than reduction. Of course, we must make the following assumptions enforcing consistency between the types of constants and their semantics:

Assumption 1 (Subject reduction for constants) *The δ-reduction preserves typings,* i.e., $(\delta) \subseteq (\sqsubseteq)$.

Theorem 1 (Subject reduction) *Reduction preserves typings*

Assumption 2 (Progress for constants) *The δ-reduction is well-defined. If $A_0 \vdash f^n\ v_1\ \ldots v_n : \tau$, then $f^n\ v_1\ \ldots v_n \in dom\,(\delta)_f$*

Theorem 2 (Progress) *Programs that are well-typed in the initial environment are either values or can be further reduced.*

Remark 2 *We have omitted the Let-nodes from expressions. With simple types, we can use the syntactic sugar* $\textbf{let}\ x = a_1\ \textbf{in}\ a_2 \stackrel{\triangle}{=} (\lambda x.a_2)\ a_1$. *Hence, we could derived the following typing rule, so as to type those nodes directly:*

$$\textsc{Let-Mono}\quad \frac{A \vdash a_1 : \tau_1 \qquad A, x : \tau_1 \vdash a_2 : \tau_2}{A \vdash \textbf{let}\ x = a_1\ \textbf{in}\ a_2 : \tau}$$

1.4.2 Type inference

We have seen that well-typed terms cannot get stuck, but can we check whether a given term is well-typed? This is the role of type inference. Moreover, type inference will characterize all types that can be assigned to a well-typed term.

The problem of type inference is: *given a type environment A, a term a, and a type τ, find all substitutions θ such that $\theta(A) \vdash a : \theta(\tau)$.* A solution θ is a principal solution of a problem $\mathcal{P}$ if all other solutions are instances of θ, *i.e.* are of the form $\theta' \circ \theta$ for some substitution θ'.

Theorem 3 (principal types) *The ML type inference problem admits principal solutions. That is, any solvable type-inference problem admits a principal solution.*

Moreover, there exists an algorithm that, given any type-inference problem, either succeeds and returns a principal solution or fails if there is no solution.

Usually, the initial type environment A_0 is closed, *i.e.* it has no free type variables. Hence, finding a principal type for a closed program a in the initial type environment is the same problem as finding a principal solution to the type inference problem (A, a, α).

Remark 3 *There is a variation to the type inference problem called* typing inference*:* given a term a, find the smallest type environment A and the smallest type τ such that $A \vdash a : \tau$. *ML does not have principal typings. See [35, 34, 76] for details.*

In the rest of this section, we show how to compute principal solutions to type inference problems. We first introduce a notation $A \triangleright a : \tau$ for type inference problems. Note that $A \triangleright a : \tau$ does not mean $A \vdash a : \tau$. The former is a (notation for a) triple while the latter is the assertion that some property holds for this triple. A substitution θ is a solution to the type inference problem $A \triangleright a : \tau$ if $\theta(A) \vdash a : \theta(\tau)$. A key property of type inference problems is that their set of solutions are closed by instantiation (*i.e.* left-composition with an arbitrary substitution). This results from a similar property for typing judgments: if $A \vdash a : \tau$, then $\theta(A) \vdash a : \theta(\tau)$ for any substitution θ.

This property allows to treat type inference problems as *constraint problems*, which are a generalization of *unification problems*. The constraint problems of interest here, written with letter U, are of one the following form.

$$U ::= \underbrace{A \rhd a : \tau}_{\text{typing problem}} \mid \underbrace{\tau_1 \doteq \ldots \tau_n}_{\text{multi-equation}} \mid U \wedge U \mid \exists \overline{\alpha}.\, U \mid \bot \mid \top$$

The two first cases are type inference problems and multi-equations (unification problems); the other forms are conjunctions of constraint problems, and the existential quantification problem. For convenience, we also introduce a trivial problem $\top$ and an unsolvable problem $\bot$, although these are definable.

It is convenient to identify constraint problems modulo the following equivalences, which obviously preserve the sets of solutions: the symbol $\wedge$ is commutative and associative. The constraint problem $\bot$ is absorbing and $\top$ is neutral for $\wedge$, that is $U \wedge \bot = \bot$ and $U \wedge \top = \top$. We also treat $\exists \alpha.\, U$ modulo renaming of bound variables, and extrusion of quantifiers; that is, if α is not free in U then $\exists \alpha'.\, U = \exists \alpha.\, U[\alpha/\alpha']$ and $U \wedge \exists \alpha.\, U' = \exists \alpha.\, (U \wedge U')$.

Type inference can be implemented by a system of rewriting rules that reduces any type inference problem to a unification problem (a constraint problem that does not constraint any type inference problem). In turns, type inference problems can then be resolved using standard algorithms (and also given by rewriting rules on unificands). Rewriting rules on unificands are written either $U \longrightarrow U'$ (or $\dfrac{U}{U'}\rightsquigarrow$) and should be read "$U$ rewrites to U'". Each rule should preserve the set of solutions, so as to be sound and complete.

The rules for type inference are given in figure 1.4. Applied in any order, they reduce any typing problem to a unification problem. (Indeed, every rule decomposes a type inference problem to smaller ones, where the size is measured by the height of the program expression.)

Figure 1.4: Simplification of type inference problems

I-Var-Fail

$$\text{if } x \notin \mathit{dom}\,(A) \qquad \frac{A \triangleright x : \tau}{\bot} \leadsto$$

I-Var

$$\text{if } x \in \mathit{dom}\,(A) \qquad \frac{A \triangleright x : \tau}{A(x) \doteq \tau} \leadsto$$

I-Fun

$$\alpha_1, \alpha_2 \notin \mathit{ftv}(\tau) \cup \mathit{ftv}(A) \qquad \frac{A \triangleright \lambda x.a : \tau}{\exists \alpha_1, \alpha_2.\, (A, x : \alpha_1 \triangleright a : \alpha_2 \wedge \tau \doteq \alpha_1 \to \alpha_2)} \leadsto$$

I-App

$$\alpha \notin \mathit{ftv}(\tau) \cup \mathit{ftv}(A) \qquad \frac{A \triangleright a_1\ a_2 : \tau}{\exists \alpha.\, (A \triangleright a_1 : \alpha \to \tau \wedge A \triangleright a_2 : \alpha)} \leadsto$$

For Let-bindings, we can either treat them as syntactic sugar and the rule Let-Sugar or use the simplification rule derived from the rule Let-Mono:

Let-Sugar

$$\frac{A \triangleright \textbf{let } x = a_1 \textbf{ in } a_2 : \tau}{A \triangleright (\lambda x.a_2)\ a_1 : \tau} \leadsto$$

Let-Mono

$$\frac{A \triangleright \textbf{let } x = a_1 \textbf{ in } a_2 : \tau}{\exists \alpha.\, (A \triangleright a_1 : \alpha \wedge A, x : \alpha \triangleright a_2 : \tau)} \leadsto$$

Implementation notes, file `infer.ml`

Since they are infinitely many constants (they contain integers), we represent the initial environment as a function that maps constants to types. It raises the exception `Free` when the requested constant does not exist.

We slightly differ from the formal presentation, by splitting bindings for constants (here represented by the global function `type_of_const`) and binding for variables (the only one remaining in type environments).

```
exception Undefined_constant of string
let type_of_const c =
```

```
  let int3 = tarrow tint (tarrow tint tint) in
  match c.name with
  | Int _ -> tint
  | Name ("+" | "*") -> int3
  | Name n -> raise (Undefined_constant n);;

exception Free_variable of var
let type_of_var tenv x =
  try List.assoc x tenv
  with Not_found -> raise (Free_variable x)
let extend tenv (x, t) = (x, t)::tenv;;
```

Type inference uses the function `unify` defined below to solved unification problems.

```
let rec infer tenv a t =
  match a with
  | Const c -> funify (type_of_const c) t
  | Var x -> funify (type_of_var tenv x) t

  | Fun (x, a) ->
      let tv1 = tvar() and tv2 = tvar() in
      infer (extend tenv (x, tv1)) a tv2;
      funify t (tarrow tv1 tv2)

  | App (a1, a2) ->
      let tv = tvar() in
      infer tenv a1 (tarrow tv t);
      infer tenv a2 tv

  | Let (x, a1, a2) ->
      let tv = tvar() in
      infer tenv a1 tv;
      infer (extend tenv (x, tv)) a2 t;;

let type_of a = let tv = tvar() in  infer [] a tv;  tv;;
```

As an example:

```
type_of e;;
```

1.4.3 Unification for simple types

Normal forms for unification problems are $\bot$, $\top$, or $\exists\overline{\alpha}.\,U$ where each U is a conjunction of multi-equations and each multi-equation contains at most one non-variable term. (Such multi-equations are of the form $\alpha_1 \doteq \ldots \alpha_n \doteq \tau$ or $\overline{\alpha} \doteq \tau$ for short.) Most-general solutions can be obtained straightforwardly from normal forms (that are not $\bot$).

The first step is to rearrange multi-equations of U into the conjunction $\overline{\alpha}_1 \doteq \tau_1 \wedge \ldots \overline{\alpha}_n \doteq \tau_n$ such that a variable of $\overline{\alpha}_j$ never occurs in τ_i for $i \leq j$. (Remark, that since U is in normal form, hence completely merged, variables $\overline{\alpha}_1, \ldots \overline{\alpha}_n$ are all distinct.) If no such ordering can be found, then there is a cycle and the problem has no solution. Otherwise, the composition $(\overline{\alpha}_1 \mapsto \tau_1) \circ \ldots (\overline{\alpha}_n \mapsto \tau_n)$ is a principal solution.

For instance, the unification problem $(g_1 \rightarrow \alpha_1) \rightarrow \alpha_1 \doteq \alpha_2 \rightarrow g_2$ can be reduced to the equivalent problem $\alpha_1 \doteq g_2 \wedge \alpha_2 \doteq (g_1 \rightarrow \alpha_1)$, which is in a solved form. Then, $\{\alpha_1 \mapsto g_2, \alpha_2 \mapsto (g_1 \rightarrow g_2)\}$ is a most general solution.

The rules for unification are standard and described in figure 1.5. Each rule preserves the set of solutions. This set of rules implements the maximum sharing so as to avoid duplication of computations. Auxiliary variables are used for sharing: the rule GENERALIZE allows to replace any occurrence of a subterm τ by a variable α and an additional equation $\alpha \doteq \tau$. If it were applied alone, rule GENERALIZE would reduce any unification problem into one that only contains small terms, *i.e.* terms of size one.

In order to obtain maximum sharing, non-variable terms should never be copied. Hence, rule DECOMPOSE requires that one of the two terms to be decomposed is a small term—which is the one used to preserve sharing. In case neither one is a small term, rule GENERALIZE can always be applied, so that eventually one of them will become a small term. Relaxing this constraint in the DECOMPOSE rule would still preserve the set of solutions, but it could result in unnecessarily duplication of terms.

Figure 1.5: Unification rules for simple types

Merge

$$\frac{\alpha \doteq e_1 \wedge \alpha \doteq e_2}{\alpha \doteq e_1 \doteq e_2} \rightsquigarrow$$

Decompose

$$\frac{g(\alpha_i^{i\in I}) \doteq g(\tau_i^{i\in I}) \doteq e}{g(\alpha_i^{i\in I}) \doteq e \wedge \bigwedge^{i\in I} (\alpha_i \doteq \tau_i)} \rightsquigarrow$$

Fail if $g_1 \neq g_2$

$$\frac{g_1(\overline{\tau}_1) \doteq g_2(\overline{\tau}_2) \doteq e}{\bot} \rightsquigarrow$$

Generalize if $\tau_0 \notin \mathcal{V}$ and $\alpha \notin ftv(g(\overline{\alpha}, \tau_0, \overline{\tau}')) \cup ftv(e)$

$$\frac{g(\overline{\tau}, \tau_0, \overline{\tau}') \doteq e}{\exists \alpha. \left(g(\overline{\tau}, \alpha, \overline{\tau}') \doteq e \wedge \alpha \doteq \tau_0 \right)} \rightsquigarrow$$

Trivial

$$\frac{\alpha \doteq \alpha \doteq e}{\alpha \doteq e} \rightsquigarrow$$

Cycle
if $\alpha_{i+1} \in ftv(e_i), \alpha_1 \in \tau, \tau \in e_n \setminus \mathcal{V}$

$$\bigwedge_{i=1}^{n} (\alpha_i \doteq e_i) \longrightarrow \bot$$

Each of these rules except (the side condition of) the Cycle rule have a constant cost. Thus, to be efficient, checking that the Cycle rule does not apply should preferably be postponed to the end. Indeed, this can then be done efficiently, once for all, in linear time on the size of the whole system of equations.

Note that rules for solving unificands can be applied in any order. They will always produce the same result, and more or less as efficiently. However, in case of failure, the algorithm should also help the user and report intelligible type-error messages. Typically, the last typing problem that was simplified will be reported together with an unsolvable subset of the remaining unification problem. Therefore, error messages completely depend on the order in which type inference and unification problems are reduced. This is actually an important matter in practice and one should pick a strategy that will make error report more pertinent. However, there does not seem to be an agreement on a best strategy, so far.

Implementation notes, file `unify.ml`

Before we describe unification itself, we must consider the representation of types and unificands carefully. As we shall see below, the two definitions are interleaved: unificands are pointers between types, and types can be represented by short types (of height at most 1) whose leaves are variables constrained to be equal to some other types.

More precisely, a multi-equation in canonical form $\alpha_1 \doteq \alpha_2 \doteq \ldots \tau$ can be represented as a chain of indirections $\alpha_1 \mapsto \alpha_2 \mapsto \ldots \tau$, where $\mapsto$ means "has the same canonical element as" and is implemented by a link (a pointer); the last term of the chain —a variable or a non-variable type— is the canonical element of all the elements of the chain. Of course, it is usually chosen arbitrarily. Conversely, a type τ can be represented by a variable α and an equation $\alpha \doteq \tau$, *i.e.* an indirection $\alpha \mapsto \tau$.

A possible implementation for types in OCaml is:

```
type type_symbol = Tarrow | Tint
type texp = { mutable texp : node; mutable mark : int }
and node = Desc of desc | Link of texp
and desc = Tvar of int | Tcon of type_symbol * texp list;;
```

The field `mark` of type `texp` is used to mark nodes during recursive visits.

Variables are automatically created with different identities. This avoid dealing with extrusion of existential variables. We also number variables with integers, but just to simplify debugging (and reading) of type expressions.

```
let count = ref 0
let tvar() = incr count; ref (Desc (Tvar !count));;
```

A conjunction of constraint problems can be inlined in the graph representation of types. For instance, $\alpha_1 \doteq \alpha_2 \to \alpha_2 \wedge \alpha_2 \doteq \tau$ can be represented as the graph $\alpha_1 \mapsto (\alpha_2 \to \alpha_2)$ where $\alpha_2 \mapsto \tau$.

Non-canonical constraint problems do not need to be represented explicitly, because they are reduced immediately to canonical unificands (*i.e.* they are implicitly represented in the control flow), or if non-solvable, an exception will be raised.

We define auxiliary functions that build types, allocate markers, cut off chains of indirections (function `repr`), and access the representation of a type (function `desc`).

```
let texp d = { texp = Desc d; mark = 0 };;
let count = ref 0
let tvar() = incr count; texp (Tvar !count);;
let tint = texp (Tcon (Tint, []))
let tarrow t1 t2 = texp (Tcon (Tarrow, [t1; t2]));;
let last_mark = ref 0
let marker() = incr last_mark; !last_mark;;

let rec repr t =
  match t.texp with
    Link u -> let v = repr u in t.texp <- Link v; v
  | Desc _ -> t

let desc t =
  match (repr t).texp with
    Link u -> assert false
  | Desc d -> d;;
```

We can now consider the implementation of unification itself. Remember that a type τ is represented by an equation $\alpha \doteq \tau$, and conversely, only decomposed multi-equations are represented, concretely; other multi-equations are represented abstractly in the control stack.

Let us consider the unification of two terms $(\alpha_1 \doteq \tau_1)$ and $(\alpha_2 \doteq \tau_2)$. If α_1 and α_2 are identical, then so must be τ_1 and τ_2 and and the to equations, so the problem is already in solved form. Otherwise, let us consider the multi-equation e equal to $\alpha_1 \doteq \alpha_2 \doteq \tau_1 \doteq \tau_2$. If τ_1 is a variable then e is effectively built by linking τ_1 to τ_2, and conversely if τ_2 is a variable. In this case e is fully decomposed, and the unification completed. Otherwise, e is equivalent by rule Decompose to the conjunction of $(\alpha_1 \doteq \alpha_2 \doteq \tau_2)$ and the equations e_i's resulting from the decomposition of $\tau_1 \doteq \tau_2$. The former is implemented by a link from α_1 to α_2. The later is implemented by recursive calls to the function `unify`. In case τ_1 and τ_2 are incompatible, then unification fails (rule FAIL).

```
exception Unify of texp * texp
exception Arity of texp * texp
```

```
let link t1 t2 = (repr t1).texp <- Link t2
let rec unify t1 t2 =
  let t1 = repr t1 and t2 = repr t2 in
  if t1 == t2 then () else
  match desc t1, desc t2 with
  | Tvar _, _ ->
      link t1 t2
  | _, Tvar _ ->
      link t2 t1
  | Tcon (g1, l1), Tcon (g2, l2) when g1 = g2 ->
      link t1 t2;
      List.iter2 unify l1 l2
  | _, _ -> raise (Unify (t1,t2)) ;;
```

This does not check for cycles, which we do separately at the end.

```
exception Cycle of texp list;;
let acyclic t =
  let visiting = marker() and visited = marker() in
  let cycles = ref [] in
  let rec visit t =
    let t = repr t in
    if t.mark > visiting then ()
    else if t.mark = visiting then cycles := t :: !cycles
    else
      begin
        t.mark <- visiting;
        begin match desc t with
       | Tvar _ -> ()
       | Tcon (g, l) -> List.iter visit l
        end;
        t.mark <- visited;
      end in
  visit t;
  if !cycles <> [] then raise (Cycle !cycles);;
let funify t1 t2 = unify t1 t2; acyclic t1;;
```

For instance, the following unification problems has only recursive solutions, which is detected by `cycle`;

```
let x = tvar() in funify x (tarrow x x);;

Uncaught exception:
Cycle [{texp= Desc ...; mark=...}; ...].
```

Exercise 3 ((*) Printer for acyclic types) *Write a simple pretty printer for acyclic types (using variable numbers to generate variable names).*
Answer □

1.4.4 Polymorphism

So far, we have only considered simple types, which do not allow any form of polymorphism. This is often bothersome, since a function such as the identity $\lambda x.x$ of type $\alpha \to \alpha$ should intuitively be applicable to any value. Indeed, binding the function to a name f, one could expect to be able to reuse it several times, and with different types, as in **let** $f = \lambda x.x$ **in** $f(\lambda x.(x + f\ 1))$. However, this expression does not typecheck, since while any type τ can be chosen for α only one choice can be made for the whole program.

One of the most interesting features of ML is its simple yet expressive form of polymorphism. ML allows type scheme to be assigned to let-bound variables that are the carrier of ML polymorphism.

A type scheme is a pair written $\forall\overline{\alpha}.\,\tau$ of a set of variables $\overline{\alpha}$ and a type τ. We identify τ with the empty type scheme $\forall.\,\tau$. We use the letter σ to represent type schemes. An instance of a type scheme $\forall\overline{\alpha}.\,\tau$ is a type of the form $\tau[\overline{\tau}'/\overline{\alpha}]$ obtained by a simultaneous substitution in τ of all quantified variables $\overline{\alpha}$ by simple types $\overline{\tau}'$ in τ. (Note that the notation $\tau[\overline{\alpha}/\overline{\tau}']$ is an abbreviation for $(\overline{\alpha} \mapsto \overline{\tau}')(\tau)$.)

Intuitively, a type scheme represents the set of all its instances. We write $ftv(\forall\overline{\alpha}.\,\tau)$ for the set of free types variables of $\forall\overline{\alpha}.\,\tau$, that is, $ftv(\tau) \setminus \overline{\alpha}$. We also lift the definition of free type variables to typings environments, by taking the free type variables of its co-domain:

$$ftv(A) = \bigcup_{z \in dom\,(A)} ftv(A(z))$$

Implementation notes, file `type-scheme.ml`

The representation of type schemes is straightforward (although other representations are possible).

```
type scheme = texp list * texp;;
```

Exercise 4 ((*) Free type variables for recursive types) *Implement the function* `ftv_type` *that computes* $ftv(\tau)$ *for types (as a slight modification to the function* `acyclic`*).* Answer

Write a (simple version of a) function `type_instance` *taking type scheme* σ *as argument and returning a type instance of* σ *obtained by renaming and stripping off the bound variables of* σ *(you may assume that* σ *is acyclic here).* Answer

Even if the input is acyclic, it is actually a graph, and may contain some sharing. It would thus be more efficient to preserve existing sharing during the copying. Write such a version. Answer □

So as to enable polymorphism, we introduce polymorphic bindings $z : \sigma$ in typing contexts. Since we can see a type τ as trivial type scheme $\forall\emptyset.\,\tau$, we can assumes that all bindings are of the form $z : \sigma$. Thus, we change rule Var to:

$$\textsc{Var-Const}\quad \frac{A(z) = \forall\overline{\alpha}.\,\tau}{A \vdash z : \tau[\overline{\tau}'/\overline{\alpha}]}$$

Accordingly, the initial environment A_0 may now contain type schemes rather that simple types. Polymorphism is also introduced in the environment by the rule for bindings, which should be revised as follows:

$$\textsc{Let}\quad \frac{A \vdash a : \tau \qquad A, x : \forall(ftv(\tau) \setminus ftv(A)).\,\tau \vdash a' : \tau'}{A \vdash \mathbf{let}\ x = a\ \mathbf{in}\ a' : \tau'}$$

That is, the type τ found for the expression a bound to x must be generalized “as much as possible”, that is, with respect to all variables appearing in τ but not in the context A, before being used for the type of variable x in the expression a'.

Conversely, the rule for abstraction remains unchanged: λ-bound variables remain monomorphic.

In summary, the set of typing rules of ML is composed of rules Fun, App from figure 1.3 plus rules Var-Const and Let from above.

Theorem 4 *Subject reduction and progress hold for ML.*

Type inference can also be extended to handle ML polymorphism. Replacing types by type schemes in typing contexts of inference problems does not present any difficulty. Then, the two rewriting I-Fun, I-App do not need to be changed. The rewriting rule I-Var can be easily be adjusted as follows, so as to constrain the type of a variable to be an instance of its declared type-scheme:

$$\begin{array}{l}\textsc{I-Var}\\ \dfrac{\begin{array}{c}\text{if } \forall\alpha.\,\tau' = A(x)\\ \text{and } \overline{\alpha}\cap ftv(\tau) = \emptyset\\ A \rhd x : \tau\end{array}}{\exists\alpha.\,\tau \doteq \tau'}\rightsquigarrow\end{array}$$

The Let rule requires a little more attention because there is a dependency between the left and right premises. One solution is to force the resolution of the typing problem related to the bound expression to a canonical form before simplifying the unificand.

$$\begin{array}{l}\textsc{I-Let}\\ \dfrac{\begin{array}{c}\text{if } \alpha \notin ftv(A),\ A \rhd a_1 : \alpha \rightsquigarrow \exists\overline{\beta}.\,U\\ \text{and } U solved,\ U \neq \bot\\ A \rhd \mathbf{let}\ x = a_1\ \mathbf{in}\ a_2 : \tau_2\end{array}}{A, x : \forall\alpha,\overline{\beta}.\,\hat{U}(\alpha) \vdash a_2 : \tau_2}\rightsquigarrow\end{array}$$

where $\hat{U}(\alpha)$ is a principal solution of U.

Implementation notes, file `poly.ml`

The implementation of type inference with ML polymorphism is a straightforward modification of type inference with simple types, once we have the auxiliary functions. We have already defined `type_instance` to be used for the implementation of the rule VAR-CONST. We also need a function `generalizable` to compute generalizable variables $ftv(t) \setminus ftv(A)$ from a type environment A and a type τ. The obvious implementation would compute $ftv(\tau)$ and $ftv(A)$ separately, then compute their set-difference. Although this could be easily implemented in linear type, we get a more efficient (and simpler) implementation by performing the whole operation simultaneously.

Exercise 5 ((**) Generalizable variables) *Generalize the implementationof* `ftv_type` *so as to obtain a direct implementation of* `generalizable` *variables.* Answer □

A naive computation of generalizable variables will visit both the type and the environment. However, the environment may be large while all free variables of the type may be bound in the most recent part of the type environment (which also include the case when the type is ground). The computation of generalizable variables can be improved by first computing free variables of the type first and maintaining an upper bound of the number of free variables while visiting the environment, so that this visit can be interrupted as soon as all variables of t are already found to be bound in A.

A more significant improvement would be to maintained in the structure of `tenv` the list of free variables that are not already free on the left. Yet, it is possible to implement the computation of generalizable variables without ever visiting `A` by maintaining a current level of freshness. The level is incremented when entering a let-binding and decremented on exiting; it is automatically assigned to every allocated variable; then generalizable variables are those that are still of fresh level after the weakening of levels due to unifications.

Finally, here is the type inference algorithm reviewed to take polymorphism into account:

```
let type_of_const c =
  let int3 = tarrow tint (tarrow tint tint) in
  match c.name with
  | Int _ -> [], tint
  | Name ("+" | "*") -> [], int3
  | Name n -> raise (Undefined_constant n);;
```

```
let rec infer tenv a t =
  match a with
  | Const c -> unify (type_instance (type_of_const c)) t
  | Var x -> unify (type_instance (type_of_var tenv x)) t

  | Fun (x, a) ->
      let tv1 = tvar() and tv2 = tvar() in
      infer (extend tenv (x, ([], tv1))) a tv2;
      unify t (tarrow tv1 tv2)

  | App (a1, a2) ->
      let tv = tvar() in
      infer tenv a1 (tarrow tv t);
      infer tenv a2 tv

  | Let (x, a1, a2) ->
      let tv = tvar() in
      infer tenv a1 tv;
      let s = generalizable tenv tv, tv in
      infer (extend tenv (x, s)) a2 t;;
let type_of a = let tv = tvar() in  infer [] a tv;  tv;;
```

1.5 Recursion

So as to be Turing-complete, ML should allow a form of recursion. This is provided by the **let rec** $f = \lambda x.a_1$ **in** a_2 form, which allows f to appear in $\lambda x.a_1$, recursively. The recursive expression $\lambda x.a_1$ is restricted to functions because, in a call-by-value strategy, it is not well-defined for arbitrary expressions.

1.5.1 Fix-point combinator

Rather than adding a new construct into the language, we can take advantage of the parameterization of the definition of the language by a set of primitives to introduce recursion by a new primitive `fix` of arity 2 and the following type:

$$\texttt{fix} : \forall \alpha_1, \alpha_2. \, ((\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2$$

The semantics of `fix` is given then by its δ-rule:

$$\texttt{fix} \; f \; v \longrightarrow f \; (\texttt{fix} \; f) \; v \tag{δ_{fix}}$$

Since `fix` is of arity 2, the expression $(\texttt{fix} \; f)$ appearing on the right hand side of the rule (δ_{fix}) is a value and its evaluation is frozen until it appears in an application evaluation context. Thus, the evaluation must continue with the reduction of the external application of f. It is important that `fix` be of arity 2, so that `fix` computes the fix-point *lazily*. Otherwise, if `fix` were of arity 1 and came with the following δ-rule,

$$\texttt{fix} \; f \longrightarrow f \; (\texttt{fix} \; f)$$

the evaluation of $\texttt{fix} \; f \; v$ would fall into an infinite loop (the active part is underlined):

$$\underline{\texttt{fix} \; f} \; v \longrightarrow f \; (\underline{\texttt{fix} \; f}) \; v \longrightarrow f \; (f \; (\underline{\texttt{fix} \; f})) \; v \longrightarrow ...$$

For convenience, we may use **let rec** $f = \lambda x.a_1$ **in** a_2 as syntactic sugar for **let** $f = \texttt{fix} \, (\lambda f.\lambda x.a_1)$ **in** a_2.

Remark 4 *The constant* `fix` *behaves exactly as the (untyped) expression*

$$\lambda f'.(\lambda f.\lambda x.f' \; (f \; f) \; x) \; (\lambda f.\lambda x.f' \; (f \; f) \; x)$$

However, this expression is not typable in ML (without recursive types).

Exercise 6 ((*) Non typability of fix-point) *Check that the definition of* `fix` *given above is not typable in ML.*

Answer ☐

Exercise 7 ((*) Using the fix point combinator) *Define the factorial function using* `fix` *and let-binding (instead of let-rec-bindings).*

Answer ☐

Exercise 8 ((**) Type soundness of the fix-point combinator) *Check that the hypotheses 1 and 2 are satisfied for the fix-point combinator* `fix` *.*

☐

Mutually recursive definitions The language OCaml allows mutually recursive definitions. For example,

$$\mathbf{let\ rec}\ f_1 = \lambda x.a_1\ \mathbf{and}\ f_2 = \lambda x.a_2\ \mathbf{in}\ a$$

where f and f' can appear in both a and a'. This can be seen as an abbreviation for

$$\begin{array}{lrl}
\mathbf{let\ rec}\ f'_1 = & \lambda f_2.\lambda x. & \mathbf{let}\ f_1 = f'_1\ f_2\ \mathbf{in} \\
 & & a_1 \\
\mathbf{in} & & \\
\mathbf{let\ rec}\ f'_2 = & \lambda x. & \mathbf{let}\ f_2 = f'_2\ \mathbf{in} \\
 & & \mathbf{let}\ f_1 = f'_1\ f_2\ \mathbf{in} \\
 & & a_2 \\
\mathbf{in} & & \\
a & &
\end{array}$$

This can be easily generalize to

$$\mathbf{let\ rec}\ f_1 = \lambda x.a_1\ \mathbf{and}\ \ldots f_n \lambda x.a_n\ \mathbf{in}\ a$$

Exercise 9 ((*) Multiple recursive definitions) *Can you translate the case for three recursive functions?*

$$\mathbf{let\ rec}\ f_1 = \lambda x.a_1\ \mathbf{and}\ f_2 = \lambda x.a_2\ \mathbf{and}\ f_3 = \lambda x.a_3\ \mathbf{in}\ a$$

Answer □

Recursion and polymorphism Since, the expression **let rec** $f = \lambda x.a$ **in** a' is understood as **let** $f = \texttt{fix}\,(\lambda f.\lambda x.a)$ **in** a', the function f is not polymorphic while typechecking the body $\lambda x.a$, since this occurs in the context $\lambda f.[\cdot]$ where f is λ-bound. Conversely, f may be be in a' (if the type of f allows) since those occurrences are *Let*-bound.

Polymorphic recursion refers to system that would allow f to be polymorphic in a' as well. Without restriction, type inference in these systems is not decidable. [28, 75].

1.5.2 Recursive types

By default, the ML type system does not allows recursive types (but it allows recursive datatype definitions —see Section 2.1.3). However, allowing recursive types is a simple extension. Indeed, OCaml uses this extension to assign recursive types to objects. The important properties of the type systems, including subject reduction and the existence of principal types, are preserved by this extension.

Indeed, type inference relies on unification, which naturally works on graphs, *i.e.* possibly introducing cycles, which are later rejected. To make the type inference algorithm work with recursive types, it suffices to remove the occur check rule in the unification algorithm. Indeed, one must then be careful when manipulating and printing types, as they can be recursive.

As shown in exercise 8, page 48, the fix point combinator is not typable in ML without recursive types. Unsurprisingly, if recursive types are allows, the call-by-value fix-point combinator `fix` is definable in the language.

Exercise 10 ((*) Fix-point with recursive types) *Check that* `fix` *is typable with recursive types.* Answer
Use let-binding to write a shorter equivalent version of `fix`*.* Answer □

Exercise 11 ((**) Printing recursive types) *Write a more sophisticated version of the function* `print_type` *that can handle recursive types (for instance, they can be printed as in OCaml, using* `as` *to alias types).* Answer □

See also section 3.2.1 for uses of recursive types in object types.

Recursive types are thus rather easy to incorporate into the language. They are quite powerful —they can type the fix-point— and also useful and sometimes required, as is the case for object types. However, recursive types are sometimes too powerful since they will often hide programmers' errors. In particular, it will detect some common forms of errors, such as missing or extra arguments very late (see exercise below for a hint). For this reason, the default in the OCaml system is to reject recursive types that do not involve an object type constructor in the recursion. However, for purpose of expressiveness or experimentation, the user can explicitly require unrestricted recursive types using the option `-rectypes` at his own risk of late detection of some from of errors —but the system remains safe, of course!

Exercise 12 ((**) Lambda-calculus with recursive types) *Check that in the absence of constants all closed programs are typable with recursive types.* Answer □

1.5.3 Type inference *v.s.* type checking

ML programs are untyped. Type inference finds most general types for programs. It would in fact be easy to instrument type inference, so that it simultaneously annotate every subterm with its type (and let-bounds with type schemes), thus transforming an untyped term into type terms.

Indeed, type terms are more informative than untyped terms, but they can still be ill-typed. Fortunately, it is usually easier to check typed terms than untyped terms for well-typedness. In particular, type checking does not need to "guess" types, hence it does not need first-order unification.

Both type inference and type checking are verifying well-typedness of programs with respect to a given type system. However, type inference assumes that terms are untyped, while type checking assumes that terms are typed. This does not mean that type checking is simpler than type inference. For instance, some type checking problems are undecidable [59]. Type checking and type inference could also be of the same level of difficulty, if type annotations are not sufficient. However, in general, type annotations may be enriched with more information so that type checking becomes easier. On the opposite, there is no other flexibility but the expressiveness of the type system to adjust the difficulty of type inference.

The approach of ML, which consists in starting with untyped terms, and later infer types is usually called *a la Curry*, and the other approach where types are present in terms from the beginning and only checked is called *a la Church*.

In general, type inference is preferred by programmers who are relieved from the burden of writing down all type annotations. However, explicit types are not just annotations to make type verification simpler, but also a useful tool in structuring programs: they play a role for documentation, enables modular programming and increase security. For instance, in ML, the module system on top of Core ML is explicitly typed.

Moreover, the difference between type inference and type checking is not always so obvious. Indeed, all nodes of the language carry implicit type information. For instance, there is no real difference between 1 and $1 : int$. Furthermore, some explicit type annotations can also be hidden behind new constants… as we shall do below.

Further reading

Reference books on the lambda calculus, which is at the core of ML are [6, 29]. Both include a discussion of the simply-typed lambda calculus. The reference article describing the ML polymorphism and its type inference algorithm, called W, is [16]. However, Mini-ML [14] is more often used as a starting point to further extensions. This also includes a description of type-inference. An efficient implementation of this algorithm is described in [63]. Of course, many other presentations can be found in the literature, sometimes with extensions.

Basic references on unification are [49, 31]. A good survey that also introduces the notion of existential unificands that we used in our presentation is [40].

Chapter 2

The core of OCaml

Many features of OCaml (and of other dialects of ML) can actually be formalized on top of core ML, either by selecting a particular choice of primitives, by encoding, or by a small extension.

2.1 Data types and pattern matching

The OCaml language contains primitive datatypes such as integers, floats, strings, arrays, etc. and operations over them. New datatypes can also be defined using a combinations of named records or variants and later be explored using pattern matching — a powerful mechanism that combines several projections and case analysis in a single construction.

2.1.1 Examples in OCaml

For instance, the type of play cards can be defined as follows:

```
type card = Card of regular | Joker
and regular = { suit : card_suit; name : card_name; }
and card_suit = Heart | Club | Spade | Diamond
and card_name =  Ace | King | Queen | Jack | Simple of int;;
```

This declaration actually defines four different data types. The type `card` of cards is a variant type with two cases. `Joker` is a special card. Other cards are of the form `Card v` where `v` is an element of the type `regular`.

In turn `regular` is the type of records with two fields `suit` and `name` of respective types `card_suit` and `card_name`, which are themselves variant types.

Cards can be created directly, using the variant tags and labels as constructors:

```
let club_jack = Card { name = Jack; suit = Club; };;
```

```
val club_jack : card = Card {suit=Club; name=Jack}
```

Of course, cards can also be created via functions:

```
let card n s = Card {name = n; suit = s}
let king s = card King s;;
```

```
val card : name -> suit -> card = <fun>
val king : suit -> card = <fun>
```

Functions can be used to shorten notations, but also as a means of enforcing invariants.

The language OCaml, like all dialects of ML, also offers a convenient mechanism to explore and de-structure values of data-types by pattern matching, also known as case analysis. For instance, we could define the value of a card as follows:

```
let value c =
  match c with
  | Joker -> 0
  | Card {name = Ace} -> 14
  | Card {name = King} -> 13
  | Card {name = Queen} -> 12
  | Card {name = Jack} -> 11
  | Card {name = Simple k} -> k;;
```

The function `value` explores the shape of the card given as argument, by doing case analysis on the outermost constructors, and whenever necessary, pursuing the analysis on the inner values of the data-structure. Cases are explored in a top-down fashion: when a branch fails, the analysis resumes with the next possible branch. However, the analysis stops as soon as the branch is successful; then, its right hand side is evaluated and returned as result.

Exercise 13 ((**) Matching Cards) *We say that a set of cards is* *compatible if it does not contain two regular cards of different values. The goal is to find hands with four compatible cards. Write a function* `find_compatible` *that given a hand (given as an unordered list of cards) returns a list of solutions. Each solution should be a compatible set of cards (represented as an unordered list of cards) of size greater or equal to four, and two different solutions should be incompatible.* Answer □

Data types may also be parametric, that is, some of their constructors may take arguments of arbitrary types. In this case, the type of these arguments must be shown as an argument to the type (symbol) of the data-structure. For instance, OCaml pre-defines the option type as follows:

```
type 'a option = Some of 'a | None
```

The option type can be used to get inject values v of type `'a` into `Some`(v) of type `'a option` with an extra value `None`. (For historical reason, the type argument `'a` is postfix in `'a option`.)

2.1.2 Formalization of superficial pattern matching

Superficial pattern matching (*i.e.* testing only the top constructor) can easily be formalized in core ML by the declaration of new type constructors, new constructors, and new constants. For the sake of simplicity, we assume that all datatype definitions are given beforehand. That is, we parameterize the language by a set of type definitions. We also consider the case of a single datatype definition, but the generalization to several definitions is easy.

Let us consider the following definition, prior to any expression:

$$\texttt{type}\ g(\overline{\alpha}) = C_1^g\ \texttt{of}\ \tau_i \mid \ldots C_n^g\ \texttt{of}\ \tau_n$$

where free variables of τ_i are all taken among $\overline{\alpha}$. (We use the standard prefix notation in the formalization, as opposed to OCaml postfix notation.)

This amounts to introducing a new type symbol g_f of arity given by the length of $\overline{\alpha}$, n unary constructors C_1^g, $\ldots C_n^g$, and a primitive f^g of arity $n+1$ with the following δ-rule:

$$f^g\ (C^g_k\ \ v)\ v_1\ \ldots v_k\ \ldots v_n \longrightarrow v_k\ v \qquad (\delta_g)$$

The typing environment must also be extended with the following type assumptions:

$$\begin{array}{ll} C^g_i & : \forall \overline{\alpha}.\, \tau_i \to g(\overline{\alpha}) \\ f^g & : \forall \overline{\alpha}, \beta.\, g(\overline{\alpha}) \to (\tau_1 \to \beta) \to \ldots (\tau_n \to \beta) \to \beta \end{array}$$

Finally, it is convenient to add the *syntactic sugar*

$$\texttt{match}\ a\ \texttt{with}\ C^g_1\ (x) \Rightarrow a_1 \ldots \mid C^g_n\ (x) \Rightarrow a_n$$

for

$$f^g\ a\ (\lambda x.a_1) \ldots (\lambda x.a_n)$$

Exercise 14 ((***) Type soundness for data-types) *Check that the hypotheses 1 and 2 are valid.* □

Exercise 15 ((**) Data-type definitions) *What happens if a free variable of τ_i is not one of the $\overline{\alpha}$'s? And conversely, if one of the $\overline{\alpha}$'s does not appear in any of the τ_i's?* Answer □

Exercise 16 ((*) Booleans as datatype definitions) *Check that the booleans are a particular case of datatypes.* Answer □

Exercise 17 ((***) Pairs as datatype definitions) *Check that pairs are a particular case of a generalization of datatypes.* Answer □

2.1.3 Recursive datatype definitions

Note that, since we can assume that the type symbol g is given first, then the types τ_i may refer to g. This allows, recursive types definitions such as the natural numbers in unary basis (analogous to the definition of list in OCaml!):

$$\textbf{type}\ I\!\!N = \mathit{Zero} \mid \mathit{Succ}\ \texttt{of}\ I\!\!N$$

OCaml imposes a restriction, however, that if a datatype definition of $g(\overline{\alpha})$ is recursive, then all occurrences of g should appear with exactly the same parameters $\overline{\alpha}$. This restriction preserves the decidability of the equivalence of two type definitions. That is, the problem "*Are two given datatype definitions defining isomorphic structures?*" would not be decidable anymore, if the restriction was relaxed. However, this question is not so meaningful, since datatype definitions are generative, and types (of datatypes definitions) are always compared by name. Other dialects of ML do not impose this restriction. However, the gain is not significant as long as the language does not allow polymorphic recursion, since it will not be possible to write interesting function manipulating datatypes that would not follow this restriction.

As illustrated by the following exercise, the fix-point combinator, and more generally the whole lambda-calculus, can be encoded using variant datatypes. Note that this is not surprising, since the fix point can be implemented by a δ-rule, and variant datatypes have been encoded with special forms of δ-rules.

Note that the encoding uses negative recursion, that is, a recursive occurrence on the left of an arrow type. It could be shown that restricting datatypes to positive recursion would preserve termination (of course, in ML without any other form of recursion).

Exercise 18 ((**) Recursion with datatypes) *The first goal is to encode lambda-calculus. Noting that the only forms of values in the lambda calculus are functions, and that a function take a value to eventually a value, use a datatype* `value` *to define two inverse functions* `fold` *and* `unfold` *of respective types:*

```
val fold : (value -> value) -> value = <fun>
val unfold : value -> value -> value = <fun>
```

Answer

Propose a formal encoding $[\![\cdot]\!]$ *of lambda-calculus into ML plus the two functions* `fold` *and* `unfold` *so that for an expression of the encode of any expression of the lambda calculus are well-typed terms.* Answer

Finally, check that $[\![\texttt{fix}]\!]$ *is well-typed.* Answer □

2.1.4 Type abbreviations

OCaml also allows type abbreviations declared as **type** $g(\overline{\alpha}) = \tau$. These are conceptually quite different from datatypes: note that τ is not preceded by a constructor here, and that multiple cases are not allowed. Moreover, a data type definition **type** $g(\overline{\alpha}) = C^g\ \tau$ would define a new type symbol g incompatible with all others. On the opposite, the type abbreviation **type** $g(\overline{\alpha}) = \tau$ defines a new type symbol g that is compatible with the top type symbol of τ since $g(\overline{\tau}')$ should be interchangeable with τ anywhere.

In fact, the simplest, formalization of abbreviations is to expand them in a preliminary phase. As long as recursive abbreviations are not allowed, this allows to replace all abbreviations by types without any abbreviations. However, this view of abbreviation raises several problem. As we have just mentioned, it does not work if abbreviations can be defined recursively. Furthermore, compact types may become very large after expansions. Take for example an abbreviation `window` that stands for a product type describing several components of windows: `title`, `body`, etc. that are themselves abbreviations for larger types.

Thus, we need another more direct presentation of abbreviations. Fortunately, our treatment of unifications with unificands is well-adapted to abbreviations: Formally, defining an abbreviation amounts to introducing a new symbol h together with an axiom $h(\overline{\alpha}) = \tau$. (Note that this is an axiom and not a multi-equation here.) Unification can be parameterized by a set of abbreviation definitions $\{h(\overline{\alpha}_h) = \tau_h \mid h \in \mathcal{A}\}$ Abbreviations are then expanded during unification, but only if they would otherwise produce a clash with another symbol. This is obtained by adding the following rewriting rule for any abbreviation h:

$$\begin{array}{l} \textsc{Abbrev} \qquad\qquad \text{if } g \neq h \\ \dfrac{\alpha \doteq h(\overline{\alpha}) \doteq g(\overline{\tau}) \doteq e}{\alpha \doteq \tau_h[\overline{\alpha}/\overline{\alpha}_h] \doteq g(\overline{\tau}) \doteq e} \rightsquigarrow \end{array}$$

Note that sharing is kept all the way, which is represented by variable α in both the premise and the conclusion: before expansions, several parts of the type may use the same abbreviation represented by α, and all of these nodes will see the expansions simultaneously.

The rule Abbrev can be improved, so as to keep the abbreviation even after expansion:

$$\textsc{Abbrev'} \qquad\qquad \text{if } g \neq h$$

$$\frac{\alpha \doteq h(\overline{\alpha}) \doteq g(\overline{\tau}) \doteq e}{\exists \alpha'. \, (\alpha \doteq h(\overline{\alpha}) \doteq e \wedge \alpha' \doteq \tau_h[\overline{\alpha}/\overline{\alpha}_h] \doteq g(\overline{\tau}))} \leadsto$$

The abbreviation can be recursive, in the sense that h may appear in τ_h but, as for data-types, with the tuple of arguments $\overline{\alpha}$ as the one of its definition. The the occurrence of τ_h in the conclusion of rule Abbrev' must be replaced by $\tau_h[\alpha/g(\overline{\alpha})]$.

Exercise 19 ((*) Mutually recursive definitions of abbreviations) *Explain how to model recursive definitions of type abbreviations* **type** $h_1(\overline{\alpha}) = \tau_1$ **and** $h_2(\overline{\alpha}_2) = \tau_2$ *in terms of several single but recursive definitions of abbreviations.* Answer □

See also Section 3.2.1 for use of abbreviations with object types.

2.1.5 Record types

Record type definitions can be formalized in a very similar way to variant type definitions. The definition

$$\texttt{type}\ g(\overline{\alpha}) = \{f_1^g \colon \tau_1; \ldots f_2^g \colon \tau_n\}$$

amounts to the introduction of a new type symbol g of arity given by the length of $\overline{\alpha}$, one n-ary constructor C^g and n unary primitives f_i^g with the following δ-rules:

$$f_i^g\ (C^g\ \ v_1\ \ldots v_i\ \ldots v_n) \longrightarrow v_i \qquad (\delta_g)$$

As for variant types, we require that all free variables of τ_i be taken among $\overline{\alpha}$. The typing assumptions for these constructors and constant are:

$$\begin{array}{ll} C^g & : \forall \overline{\alpha}. \, \tau_1 \to \ldots \tau_n \to \overline{\alpha}\ \texttt{g} \\ f_i^g & : \forall \overline{\alpha}. \, g(\overline{\alpha}) \to \tau_i \end{array}$$

The syntactic sugar is to write $a.f_i^g$ and $\{f_1^g = a_1; \ldots f_n^g = a_n\}$ instead of $f_i^g\ \ a$ and $C^g\ a_1\ \ldots a_n$.

2.2 Mutable storage and side effects

The language we have described so far is *purely functional*. That is, several evaluations of the same expression will always produce the same answer. This prevents, for instance, the implementation of a counter whose interface is a single function `next : unit -> int` that increments the counter and returns its new value. Repeated invocation of this function should return a sequence of consecutive integers —a different answer each time.

Indeed, the counter needs to memorize its state in some particular location, with read/write accesses, but before all, some information must be shared between two calls to `next`. The solution is to use mutable storage and interact with the store by so-called *side effects*.

In OCaml, the counter could be defined as follows:

```
let new_count =
 let r = ref 0 in
 let next () = r := !r+1; !r in
 next;;
```

Another, maybe more concrete, example of mutable storage is a bank account. In OCaml, record fields can be declared mutable, so that new values can be assigned to them later. Hence, a bank account could be a two-field record, its number, and its balance, where the balance is mutable.

```
type account = { number : int; mutable balance : float }
let retrieve account requested =
  let s = min account.balance requested in
  account.balance <- account.balance -. s; s;;
```

In fact, in OCaml, references are not primitive: they are special cases of mutable records. For instance, one could define:

```
type 'a ref = { mutable content : 'a }
let ref x = { content = x }
let deref r = r.content
let assign r x = r.content <- x; x
```

2.2.1 Formalization of the store

We choose to model single-field store cells, *i.e.* references. Multiple-field records with mutable fields can be modeled in a similar way, but the notations become heavier.

Certainly, the store cannot be modeled by just using δ-rules. There should necessarily be another mechanism to produce some side effects so that repeated computations of the same expression may return different values.

The solution is to model the store, rather intuitively. For that purpose, we introduce a denumerable collection of store locations $\ell \in \mathcal{L}$. We also extend the syntax of programs with store locations and with constructions for manipulating the store:

$$a ::= \ldots \mid \ell \mid \texttt{ref}\ a \mid \texttt{deref}\ a \mid \texttt{assign}\ a\ \ a'$$

Following the intuition, the store is modeled as a global partial mapping s from store locations to values. Small-step reduction should have access to the store and be able to change its content. We model this by transforming pairs a/s composed of an expression and a store rather than by transforming expressions alone.

Store locations are values.

$$v ::= \ldots \mid \ell$$

The semantics of programs that do not manipulate the store is simply lifted to leave the store unchanged:

$$a/s \longrightarrow a'/s \text{ if } a \longrightarrow a'$$

Primitives operating on the store behaves as follows:

$$\begin{array}{rll} \texttt{ref}\ v/s & \longrightarrow \ell/s, \ell \mapsto v & \ell \notin \mathit{dom}\,(s) \\ \texttt{deref}\ \ell/s & \longrightarrow s(\ell)/s & \ell \in \mathit{dom}\,(s) \\ \texttt{assign}\ \ell\ \ v/s & \longrightarrow v/s, \ell \mapsto v & \ell \in \mathit{dom}\,(s) \end{array}$$

Hence, we must count store location among values: Additionally, we lift the context rule to value-store pairs:

$$E[a]/s \longrightarrow E[a']/s \text{ if } a/s \longrightarrow a'/s$$

Example 3 Here is a simple example of reduction:

$$\begin{array}{ll}
 & \mathbf{let}\ x = \underline{\texttt{ref 1}}\ \mathbf{in}\ \texttt{assign}\ x\ \ (1 + \texttt{deref}\ x)/\emptyset \\
\longrightarrow & \underline{\mathbf{let}\ x = \ell\ \mathbf{in}\ \texttt{assign}\ x\ \ (1 + \texttt{deref}\ x)}/\ell \mapsto 1 \\
\longrightarrow & \texttt{assign}\ \ell\ (1 + \underline{\texttt{deref}\ \ell})/\ell \mapsto 1 \\
\longrightarrow & \texttt{assign}\ \ell\ (\underline{1 + 1})/\ell \mapsto 1 \\
\longrightarrow & \underline{\texttt{assign}\ \ell\ (2)}/\ell \mapsto 1 \\
\longrightarrow & 2/\ell \mapsto 2
\end{array}$$

Remark 5 *Note that, we have not modeled garbage collection: new locations created during reduction by the* Ref *rule will remain in the store forever.*

An attempt to model garbage collection of unreachable locations is to use an additional rule.

$$a/s \longrightarrow a/(s \setminus \ell) \qquad\qquad \ell \notin a$$

However, this does not work for several reasons.

Firstly, the location ℓ may still be accessible, indirectly: starting from the expression a one may reach a location ℓ' whose value $s(\ell')$ may still refer to ℓ. Changing the condition to $\ell \notin a, (s \setminus \ell)$ would solve this problem but raise another one: cycles in s will never be collected, even if not reachable from a. So, the condition should be that of the form "ℓ is not accessible from a using store s". Writing, this condition formally, is the beginning of a specification of garbage collection...

Secondly, it would not be correct to apply this rule locally, to a subterm, and then lift the reduction to the whole expression by an application of the context rule. There are two solutions to this last problem: one is to define a notion of toplevel reduction to prevent local applications of garbage collection; The other one is to complicate the treatment of store so that locations can be treated locally (see [77] for more details).

In order to type programs with locations, we must extend typing environment with assumptions for the type of locations:

$$A ::= \ldots \mid A, \ell : \tau$$

Remark that store locations are not allowed to be polymorphic (see the discussion below). Hence the typing rule for using locations is simply

$$\textsc{Loc}\quad \frac{\ell : \tau \in A}{A \vdash \ell : \tau}$$

Operations on the store can be typed as the application of constants with the following type schemes in the initial environment A_0:

$$\begin{aligned} \texttt{ref}\ \_ &: \forall \alpha.\, \alpha \to \texttt{ref}\ \alpha \\ \texttt{deref}\ \_ &: \forall \alpha.\ \texttt{ref}\ \alpha \to \alpha \\ \texttt{assign}\ \_\ \_ &: \forall \alpha.\ \texttt{ref}\ \alpha \to \alpha \to \alpha \end{aligned}$$

(Giving specific typing rules Ref, Deref, and Assign would unnecessarily duplicate rule App into each of them)

2.2.2 Type soundness

We first define store typing judgments: we write $A \vdash a/s : \tau$ if there exists a store extension A' of A (*i.e.* outside of domain of A) such that $A' \vdash a : \tau$ and $A' \vdash s(\ell) : A'(\ell)$ for all $\ell \in \mathit{dom}\,(A')$. We then redefine $\sqsubseteq$ to be the inclusion of store typings.

$$(a/s \sqsubseteq a'/s') \iff \forall (A, \tau)(A \vdash a/s : \tau \Longrightarrow A \vdash a'/s' : \tau)$$

Theorem 5 (Subject reduction) *Store-reduction preserves store-typings.*

Theorem 6 (Progress) *If $A_0 \vdash a/s : \tau$, then either a is a value, or a/s can be further reduced.*

2.2.3 Store and polymorphism

Note that store locations cannot be polymorphic. Furthermore, so as to preserve subject reduction, expressions such as `ref` v should not be polymorphic either, since `ref` v reduces to ℓ where ℓ is a new location of the same type as the type of v. The simplest solution to enforce this restriction is to restrict **let** $x = a$ **in** a' to the case where a is a value v (other cases can still be seen as syntactic sugar for $(\lambda x.a')\ a$.) Since `ref` a is not a value —it is an application— it then cannot be polymorphic. Replacing a by a value v does not make any difference, since `ref` is not a constructor but a primitive. Of course, this solution is not optimal, *i.e.* there are safe cases that are rejected. All other solutions that have been explored end up to be too complicated, and also restrictive. This solution, known as "value-only polymorphism" is unambiguously the best compromise between simplicity and expressiveness.

To show how subject reduction could fail with polymorphic references, consider the following counter-example.

let `id` = `ref` (**fun** `x -> x`) **in** (`id` := `succ`; !`id` `true`);;

If "id" had a polymorphic type $\forall \alpha.\, \tau$, it would be possible to assign to `id` a function of the less general type, *e.g.* the type `int -> int` of `succ`, and then to read the reference with another incompatible less general type `bool -> bool`; however, the new content of `id`, which is the function `succ`, does not have type `bool -> bool`.

Another solution would be to ensure that values assigned to `id` have a type scheme at least as general as the type of the location. However, ML cannot force expressions to have polymorphic types.

Exercise 20 ((**) Recursion with references) *Show that the fix point combinator* `fix` *can be defined using references alone (*i.e. *using without recursive bindings, recursive types* etc.*).* Answer □

2.2.4 Multiple-field mutable records

In OCaml references cells are just a particular case of records with mutable fields. To model those, one should introduce locations with several fields as well. The does not raise problem in principle but makes the notations significantly heavier.

2.3 Exceptions

Exceptions are another imperative construct. As for references, the semantics of exceptions cannot be given only by introducing new primitives and δ-rules.

We extend the syntax of core ML with:

$$a ::= \ldots \mid \texttt{try}\ a\ \texttt{with}\ x \Rightarrow a \mid \texttt{raise}\ a$$

We extend the evaluation contexts, so as to allow evaluation of exceptions and exception handlers.

$$E :: \ldots \mid \texttt{try}\ E\ \texttt{with}\ x \Rightarrow a \mid \texttt{raise}\ E$$

Finally, we add the following redex rules:

$$\begin{array}{rl} \texttt{try}\ v\ \texttt{with}\ x \Rightarrow a \longrightarrow v & (\mathit{Try}) \\ \texttt{try}\ E'[\texttt{raise}\ v]\ \texttt{with}\ x \Rightarrow a \longrightarrow \mathbf{let}\ x = v\ \mathbf{in}\ a & (\mathit{Raise}) \end{array}$$

with the side condition for the Raise rule that the evaluation context E' does not contain any node of the form $(\texttt{try}\ \_\ \texttt{with}\ \_ \Rightarrow \_)$. More precisely, such evaluation contexts can be defined by the grammar:

$$E' ::= [\cdot] \mid E'\ a \mid v\ E' \mid \texttt{raise}\ E'$$

Informally, the Raise rule says that if the evaluation of a raises an exception with a value v, then the evaluation should continue at the first enclosing handler by applying the right hand-side of the handler value v. Conversely, is the evaluation of a returns a value, then the Try rule simply removes the handler.

The typechecking of exceptions raises similar problems to the typechecking of references: if an exception could be assigned a polymorphic type σ, then it could be raised with an instance τ_1 of σ and handled with the asumption that it has type τ_2 —another instance of σ. This could lead to a type error if τ_1 and τ_2 are incompatible. To avoid this situation, we assume given a particular closed type τ_0 to be taken for the type of exceptions. The typing rules are:

$$\textsc{Raise}\quad \frac{A \vdash a : \tau_0}{A \vdash \texttt{raise}\ a : \alpha} \qquad\qquad \textsc{Try}\quad \frac{A \vdash a_1 : \tau \qquad A, x : \tau_0 \vdash a_2 : \tau}{A \vdash \texttt{try}\ a_1\ \texttt{with}\ x \Rightarrow a_2 : \tau}$$

Exercise 21 ((**) Type soundness of exceptions) *Show the correctness of this extension.* ☐

Exercise 22 ((**) Recursion with exceptions) *Can the fix-point combinator be defined with exceptions?*

Answer ☐

Further reading

We have only formalized a few of the ingredients of a real language. Moreover, we abstracted over many details. For instance, we assumed given the full program, so that type declaration could be moved ahead of all expressions.

Despite many superficial differences, Standard ML is actually very close to OCaml. Standard ML has also been formalized, but in much more details [51, 50]. This is a rather different task: the lower level and finer grain exposition, which is mandatory for a specification document, may unfortunately obscure the principles and the underlying simplicity behind ML.

Among many extensions that have been proposed to ML, a few of them would have deserved more attention, because there are expressive, yet simple to formalize, and in essence very close to ML.

Records as datatype definitions have the inconvenience that they must always be declared prior to their use. Worse, they do not allow to define a function that could access some particular field uniformly in any record containing at least this field. This problem, known as polymorphic record access, has been proposed several solutions [65, 57, 33, 37], all of which relying more or less directly on the powerful idea of row variables [73]. Some of these solutions simultaneously allow to extend records on a given field uniformly, *i.e.* regardless of the other fields. This operation, known as polymorphic record extension, is quite expressive. However, extensible records cannot be typed as easily or compiled as efficiently as simple records.

Dually, variant allows building values of an open sum type by tagging with labels without any prior definition of all possible cases. Actually, OCaml was recently extended with such variants [23]

Datatypes can also be used to embed existential or universal types into ML [41, 64, 53, 24].

Chapter 3

The object layer

We first introduce objects and classes, informally. Then, we present the core of the object layer, leaving out some of the details. Last, we show a few advanced uses of objects.

3.1 Discovering objects and classes

In this section, we start with a series of very basic examples, and present the key features common to all object oriented languages; then we introduce polymorphism, which play an important role in OCaml.

Object, classes, and types. There is a clear distinction to be emphasized between objects, classes, and types. *Objects* are values which are returned by evaluation and that can be sent as arguments to functions. Objects only differ from other values by the way to interact with them, that is to send them messages, or in other words, to invoke their methods.

Classes are not objects, but definitions for building objects. Classes can themselves be built from scratch or from other classes by inheritance. Objects are usually created from classes by instantiation (with the `new` construct) but can also be created from other objects by *cloning* or *overriding*.

Neither classes nor objects are types. Object have object types, which are regular types, similar to but different from arrow or product types. Classes also have types. However, class types are not regular types, as much as classes are not regular expressions, but expressions of a small class language.

Classes may be in an inheritance (sub-classing) relation, which is the case when a class inherits from another one. Object types may be in a subtyping relation. However, there is no correspondence to be made between sub-classing and subtyping, anyway.

3.1.1 Basic examples

A class is a model for objects. For instance, a class `counter` can be defined as follows.

```
class counter = object
  val mutable n = 0
  method incr = n <- n+1
  method get = n
end;;
```

That is, objects of the class `counter` have a mutable field `n` and two methods `incr` and `get`. The field `n` is used to record the current value of the counter and is initialized to `0`. The two methods are used to increment the counter and read its current, respectively.

As for any other declaration, the OCaml system infer a principal type for this declaration:

```
class counter :
  object
    val mutable n : int
    method get : int
    method incr : unit
  end
```

The class type inferred mimics the declaration of the class: it describes the types of each field and each method of the class.

An object is created from a class by taking an instance using the `new` construct:

```
let c = new counter;;
```

```
val c : counter = <obj>
```

Then, methods of the object can be invoked —this is actually the only form of interaction with objects.

```
c#incr; c#incr; c#get;;
```

```
- : int = 2
```

Note the use of `#` for method invocation. The expression `c.incr` would assume that `c` is a record value with an `incr` field, and thus fail here.

Fields are encapsulated, and are accessible only via methods. Two instances of the same class produce different objects with different encapsulated state. The field `n` is not at all a class-variable that would be shared between all instances. On the contrary, it is created when taking an instance of the class, independently of other objects of the same class.

```
(new counter)#get;;
```

```
- : int = 0
```

Note that the generic equality (the infix operator `=`) will always distinguish two different objects even when they are of the same class and when their fields have identical values:

```
(new counter) = (new counter);;
```

```
- : bool = false
```

Objects have their own identity and are never compared by structure.

Classes Classes are often used to encapsulate a piece of state with methods. However, they can also be used, without any field, just as a way of grouping related methods:

```
class out =
  object
    method char x = print_char x
    method string x = print_string x
  end;;
```

A similar class with a richer interface and a different behavior:

```
class fileout filename =
  object
    val chan = open_out filename
```

```
    method char x = output_char chan x
    method string x = output_string chan x
    method seek x = seek_out chan x
  end;;
```

This favors the so-called "programming by messages" paradigm:

```
let stdout = new out and log = new fileout "log";;
let echo_char c = stdout#char c; log#char c;;
```

Two objects may answer the same message differently, by running their own methods, *i.e.* depending on their classes.

Inheritance Classes are used not only to create objects, but also to create richer classes by inheritance. For instance, the `fileout` class can be enriched with a method to close the output channel:

```
class fileout' filename =
  object (self)
    inherit fileout filename
    method close = close_out chan
  end
```

It is also possible to define a class for the sole purpose of building other classes by inheritance. For instance, we may define a class of `writer` as follows:

```
class virtual writer =
  object (this)
    method virtual char : char -> unit
    method string s =
      for i = 0 to String.length s −1 do this#char s.[i] done
    method int i = this#string (string_of_int i)
  end;;
```

The class `writer` refers to other methods of the same class by sending messages to the variable `this` that will be bound dynamically to the object running the method. The class is flagged `virtual` because it refers to the method char that is not currently defined. As a result, it cannot be instantiated into an object, but only inherited. The method `char` is virtual, and it will remain virtual in subclasses until it is defined. For instance, the class `fileout` could have been defined as an extension of the class `writer`.

```
class fileout filename = object
  inherit writer
  method char x = output_char chan x
  method seek pos = seek_out pos
end
```

Late binding During inheritance, some methods of the parent class may be redefined. Late binding refers to the property that the most recent definition of a method will be taken from the class at the time of object creation. For instance, another more efficient definition of the class `fileout` would ignore the default definition of the method `string` for writers and use the direct, faster implementation:

```
class fileout filename = object
  inherit writer
  method char x = output_char chan x
  method string x = output_string chan x
  method seek pos = seek_out pos
end
```

Here the method `int` will call the new efficient definition of method `string` rather than the default one (as an early binding strategy would do). Late binding is an essential aspect of object orientation. However, it is also a source of difficulties.

Type abbreviations The definition of a class simultaneously defines a type abbreviation for the type of the objects of that class. For instance, when defining the objects `out` and `log` above, the system answers were:

```
val stdout : out = <obj>
val log : fileout = <obj>
```

Remember that the types `out` and `fileout` are only abbreviations. Object types are structural, *i.e.* one can always see their exact structure by expanding abbreviations at will:

```
(stdout : < char : char -> unit; string : string -> unit >);;
```

```
- : out = <obj>
```

(Abbreviations have stronger priority than other type expressions and are kept even after they have been expanded if possible.) On the other hand, the following type constraint fails because the type `fileout` of `log` contains an additional method `seek`.

```
(log : < char : char -> unit; string : string -> unit >);;
```

3.1.2 Polymorphism, subtyping, and parametric classes

Polymorphism play an important role in the object layer. Types of objects such as `out`, `fileout` or

```
<char : char -> unit; string: string -> unit >
```

are said to be *closed*. A closed type exhaustively enumerates the set of accessible methods of an object of this types. On the opposite, an *open* object type only specify a subset of accessible methods. For instance, consider the following function:

```
let send_char x = x#char;;
```

```
val send_char : < char : 'a; .. > -> 'a = <fun>
```

The domain of `send_char` is an object having at least a method `char` of type `'a`. The ellipsis `..` means that the object received as argument may also have additional methods. In fact, the ellipsis stands for an anonymous row variable, that is, the corresponding type is polymorphic (not only in `'a` but also in `..`). It is actually a key point that the function `send_char` is polymorphic, so that it can be applied to *any* object having a `char` method. For instance, it can be applied to both `stdout` or `log`, which are of different types:

```
let echo c = send_char stdout c; send_char log c;;
```

Of course, this would fail without polymorphism, as illustrated below:

```
(fun m -> m stdout c; m log c) send_char;;
```

Subtyping In most object-oriented languages, an object with a larger interface may be used in place of an object with a smaller one. This property, called *subtyping*, would then allow `log` to be used with type `out`, *e.g.* in place of `stdout`. This is also possible in OCaml, but the use of subtyping must be indicated explicitly. For instance, to put together `stdout` and `log` in a same homogeneous data-structure such as a list, `log` can be coerced to the typed `stdout`:

```
let channels = [stdout; (log : fileout :> out)];;
```

```
val channels : out list = [<obj>; <obj>]
```

```
let braodcast m = List.iter (fun x -> x#string m) channels;;
```

The domain of subtyping coercions may often (but not always) be omitted; this is the case here and we can simply write:

```
let channels = [stdout; (log :> out)];;
```

In fact, the need for subtyping is not too frequent in OCaml, because polymorphism can often advantageously be used instead, in particular, for polymorphic method invocation. Note that reverse coercions from a supertype to a supertype are never possible in OCaml.

Parametric classes Polymorphism also plays an important role in parametric classes. Parametric classes are the counterpart of polymorphic data structures such as lists, sets, *etc.* in object-oriented style. For instance, a class of stacks can be parametric in the type of its elements:

```
class ['a] stack = object
  val mutable p : 'a list = []
  method push v = p <- v :: p
  method pop =
    match p with h :: t -> p <- t; h | [] -> failwith "Empty"
end;;
```

The system infers the following polymorphic type.

```
class ['a] stack :
  object
    val mutable p : 'a list
    method pop : 'a method push : 'a -> unit
  end
```

The parameter must always be introduced explicitly (and used inside the class) when defining parametric classes. Indeed, a parametric class does not only define the code of the class, but also defines a type abbreviation for objects of this class. So this constraint is analogous to the fact that type variables free in the definition of a type abbreviation should be bound in the parameters of this abbreviation.

Parametric classes are quite useful when defining general purpose classes. For instance, the following class can be used to maintain a list of subscribers and relay messages to be sent all subscribers via the message `send`.

```
class ['a] relay = object
    val mutable l : 'a list = []
    method add x = if not (List.mem x l) then l <- x::l
    method remove x = l <- List.filter (fun y -> x <> y) l
    method send m = List.iter m l
  end;;
```

While parametric classes are polymorphic, objects of parametric classes are not. The creation of an instance of a class `new` *c* must be compared with the creation of a reference `ref` *a*. Indeed, the creation of an object may also create mutable fields, and therefore it cannot safely be polymorphic. (Even if the class types does not show any mutable fields, they might have just been hidden from a parent class.)

The type of self Another important feature of OCaml is its ability to precisely relate the types of two objects without knowing their exact shape. For instance, consider the library function that returns a shallow copy (a clone) of any object given as argument:

```
Oo.copy : (< .. > as 'a) -> 'a
```

Firstly, this type indicates that the argument must be an object; we can recognize the anonymous row variable "`..`", which stands for "any other methods"; secondly, it indicates that whatever the particular shape of the argument is, the result type remains exactly the same as the argument type. This type of `Oo.copy` is polymorphic, the types `fileout -> fileout`, `<gnu : int> -> <gnu : int>`, or `< > -> < >` being some example of instances. On the opposite, `int -> int` is not a correct type for `Oo.copy`.

Copying may also be internalized as a particular method of a class:

```
class copy = object (self) method copy = Oo.copy self end;;
```

```
class copy : object ('a) method copy : 'a end
```

The type `'a` of `self`, which is put in parentheses, is called *self-type*. The class type of the class `copy` indicates that the class `copy` has a method `copy` and that this method returns an object with of self-type. Moreover, this property will remain true in any subclass of `copy`, where self-type will usually become a specialized version of the the self-type of the parent class.

This is made possible by keeping self-type an open type and the class polymorphic in self-type. On the contrary, the type of the objects of a class is always closed. It is actually an instance of the self-type of its class. More generally, for any class C, the type of objects of a subclass of C is an instance of the self-type of class C.

Exercise 23 ((*) Self types) *Explain the differences between the following two classes:*

```
class c1 = object (self) method c = Oo.copy self end
class c2 = object (self) method c = new c2 end;;
```

Answer ☐

Exercise 24 ((**) Backups) *Define a class* `backup` *with two methods* `save` *and* `restore`*, so that when inherited in an (almost) arbitrary class the method* `save` *will backup the internal state, and the method* `restore` *will return the object in its state at the last backup.* Answer ☐

There is a functional counterpart to the primitive `Oo.copy` that avoids the use of mutable fields. The construct `{< >}` returns a copy of self; thus, it can only be used internally, in the definition of classes. However, it has the advantage of permitting to change the values of fields while doing the copy. Below is a functional version of backups introduced in the exercise 24 (with a different interface).

```
class original =
  object (self)
    val original = None
    method copy = {< original = Some self >}
    method restore =
      match original with None -> self | Some x -> x
  end;;
```

Here, the method `copy`, which replaces the method `save` of the imperative version, returns a copy of self in which the original version has been stored *unchanged.* Remark that the field `original` does not have to be mutable.

Exercise 25 ((***) Logarithmic Backups) *Write a variant of either the class* `backup` *or* `original` *that keeps all intermediate backups.*

Add a method `clean` *that selectively remove some of the intermediate backups. For instance, keeping only versions of age* 2^0, 2^1, *...* 2^n. ☐

Binary methods We end this series of examples with the well-known problem of binary methods. These are methods that take as argument an object (or a value containing an object) of the same type as the type of self. Inheriting from such classes in often a problem. However, this difficulty is unnoticeable in OCaml as a result of the expressive treatment of self types and the use of polymorphism.

As an example, let us consider two players: an amateur and a professional. Let us first introduce the amateur player.

```
class amateur = object (self)
  method play x risk = if Random.int risk > 0 then x else self
end;;
```

The professional player inherits from the amateur, as one could expect. In addition, the professional player is assigned a certain level depending on his past scores. When a professional player plays against another player, he imposes himself a penalty so as to compensate for the difference of levels with his partner.

```
class professional k = object (self)
  inherit amateur as super
  method level = k
  method play x risk = super#play x (risk + self#level − x#level)
end;;
```

The class `professional` is well-typed and behaves as expected.

However, a professional player cannot be considered as an amateur player, even though he has more methods. Otherwise, a professional could play with an amateur and ask for the amateur's level, which an amateur would not like, since he does not have any level information.

This is a common pattern with binary methods: as the binary method of a class C expects an argument of self-type, *i.e.* of type the type of objects of class C, an object of a super-class C' of C does not usually have is an interface rich enough to be accepted by the method of C.

An object of a class with a binary method has a recursive type where recursion appears at least once in a contravariant position (this actually is a more accurate, albeit technical definition of binary methods).

```
class amateur : object ('a)
  method play : 'a -> int -> 'a
end
class professional : object ('a)
  method level : int
  method play : 'a -> int -> 'a
end
```

As a result of this contravariant occurrence of recursion, the type of an object that exposes a binary method does not possess any subtype but itself. For example, `professional` is not a subtype of `amateur`, even though it has more methods. Conversely, `< level : int >` is a correct subtype for an object of the class `professional` that does not expose its binary method; thus, this type has non-trivial subtypes, such as `amateur` or `professional`.

Exercise 26 ((**) Object-oriented strings) *Define a class* `string` *that embeds most important operations on strings in a class.*

Extend the previous definition with a method `concat`*.* Answer □

3.2 Understanding objects and classes

In this section, we formalize the core of the object layer of OCaml. In this presentation, we make a few simplifications that slightly decrease the expressiveness of objects and classes, but retain all of their interesting facets.

One of the main restrictions is to consider fields immutable. Indeed, mutable fields are important in practice, but imperative features are rather orthogonal to object-oriented mechanisms. Imperative objects are well explained as the combination of functional objects with references:

mutable fields can usually be replaced by immutable fields whose content is a reference. There is a lost though, which we will discuss below. Thus we shall still describe mutable fields, including their typechecking, but we will only describe their semantics informally.

As was already shown in the informal presentation of objects and classes, polymorphism is really a key to the expressiveness of OCaml's objects and classes. In particular, it is essential that:

- Object types are *structural* (*i.e.* they structure is transparent) and use *row variables* to allow polymorphism;
- Class types are polymorphic in the type of self to allow further refinements.

Besides increasing expressiveness, polymorphism also plays an important role for type inference: it allows to send messages to objects of unknown classes, and in particular, without having to determine the class they belong to. This is permitted by the use of structural object types and row variables. Structural types mean that the structure of types is always transparent and cannot be hidden by opaque names. (We also say that object-types have structural equality, as opposed to by-name equality of data-types.) Of course, objects are "first class" entities: they can be parameters of functions, passed as arguments or return as results.

On the contrary, classes are "second class" entities: they cannot be parameters of other expressions. Hence, only existing classes, *i.e.* of known class types can be inherited or instantiated into objects. As a result, type reconstruction for classes does not require much more machinery than type inference for components of classes, that is, roughly, than type inference for expressions of the core language.

Syntax for objects and classes We assume three sets $u \in \mathcal{F}$ of field names, $m \in \mathcal{M}$ of method names, and $z \in \mathcal{Z}$ of class names.

To take objects and classes into account, the syntax of the core language is extended as described in fig 3.1. Class bodies are either new classes, written `object B end`, or expressions of a small calculus of classes that including class variables, abstraction (over regular values —not classes), and application of classes to values.

Figure 3.1: Syntax for objects and classes

Expressions

$$a ::= \ldots \mid \texttt{new}\ a \mid a\#m \mid \textbf{class}\ z = d\ \textbf{in}\ a$$

Class expressions

$$d ::= \underbrace{\textbf{object}\ B\ \textbf{end}}_{\text{creation}} \mid \underbrace{z \mid \lambda x.d \mid d\ a}_{\text{abstraction and instantiation}}$$

Class bodies

$$B ::= \emptyset \mid B\ \textbf{inherit}\ d \mid B\ \underbrace{u = a}_{\text{field}} \mid B\ \underbrace{m = \varsigma(x)\, a}_{\text{method}}$$

A minor difference with OCaml is we chose to bind self in each method rather than once for all at the beginning of each class body. Methods are thus of the form $\varsigma(x)\, m$ where x is a binder for self. Of course, we can always see the OCaml expression **object** $(x)\, u{=}a; m_1{=}a_1; m_2{=}a_1$ **end** as syntactic sugar for **object** $u = a; m_1 = \varsigma(x)\, a_1; m_2 = \varsigma(x)\, a_1$ **end**. Indeed, the latter is easier to manipulate formally, while the former is shorter and more readable in programs.

As suggested above, type-checking objects and classes are orthogonal, and rely on different aspects of the design. We consider them separately in two different sections.

3.2.1 Type-checking objects

The successful type-checking of objects results from a careful combination of the following features: structural object types, row variables, recursive types, and type abbreviations. Structural types and row polymorphism allow polymorphic invocation of messages. The need for recursive types arise from the structural treatment of types, since object types are recursive in essence (objects often refer to themselves). Another consequence of structural types is that the types of objects tend to be very large. Indeed, they describe the types of all accessible methods, which themselves are often functions between objects with large types. Hence, a smart type abbreviation mechanism is used to keep types relatively small and their representation compact. Type abbreviations are not required in theory, but they are crucial in practice, both for smooth interaction with the user and for reasonable efficiency of type inference. Furthermore, observing that some forms of object types are never inferred allows to keep all row variables anonymous, which significantly simplifies the presentation of object types to the user.

Object types Intuitively, an object type is composed of the row of all visible methods with their types (for closed object types), and optionally ends with a row variable (for open object types). However, this presentation is not very modular. In particular, replacing row variables by a row would not yield a well-formed type. Instead, we define types in two steps. Assuming a countable collection of row variables $\varrho \in \mathcal{R}$, raw types and rows are described by the following grammars:

$$\tau ::= \ldots \mid \langle \rho \rangle \qquad\qquad \rho ::= 0 \mid \varrho \mid m : \tau ; \rho$$

This prohibits row variables to be used anywhere else but at the end of an object type. Still, some raw types do not make sense, and should be rejected. For instance, a row with a repeated label such as $(m : \tau ; m : \tau' ; \rho)$ should be rejected as ill-formed. Other types such as $\langle m : \tau ; \rho \rangle \to \langle \rho \rangle$ should also be ruled out since replacing ρ by $m : \tau' ; \rho$ would produce an ill-formed type. Indeed, well-formedness should be preserved by well-formed substitutions. A modular criteria is to sort raw types by assigning to each row a set of labels that it should not define, and by assigning to a toplevel row ρ (one appearing immediately under the type constructor $\langle \cdot \rangle$) the sort $\emptyset$.

Then, types are the set of well-sorted raw types. Furthermore, rows are considered modulo left commutation of fields. That is, $m : \tau ; (m' : \tau' ; \rho)$ is equal to $m' : \tau' ; (m : \tau ; \rho)$. For notational convenience, we assume that $(m : \_ ; \_)$ binds tighter to the right, so we simply write $(m : \tau ; m' : \tau' ; \rho)$.

Remark 6 *Object types are actually similar to types for (non-extensible) polymorphic records: polymorphic record access corresponds to message invocation; polymorphic record extension is not needed here, since OCaml class-based objects are not extensible. Hence, some simpler kinded approach to record types can also be used [57]. (See end of Chapter 2, page 66 for more references on polymorphic records.)*

Message invocation Typing message invocation can be described by the following typing rule:

$$\textsc{Message}\quad \frac{A \vdash a : \langle m : \tau; \rho \rangle}{A \vdash a\#m : \tau}$$

That is, if an expression a is an object with a method m of type τ and maybe other methods (captured by the row ρ), then the expression $a\#m$ is well-typed and has type τ.

However, instead of rule Message, we prefer to treat message invocation ($a\#m$) as the application of a primitive ($\_\#m$) to the expression a. Thus, we assume that the initial environment contains the collection of assumptions $((\_.m) : \forall \alpha, \varrho.\, \langle m : \alpha; \varrho \rangle \rightarrow \alpha)^{m \in \mathcal{M}}$. We so take advantage of the parameterization of the language by primitives and avoid the introduction of new typing rules.

Type inference for object types Since we have not changed the set of expressions, the problem of type inference (for message invocation) reduces to solving unification problems, as before. However, types are now richer and include object types and rows. So type inference reduces to unification with those richer types.

The constraint that object types must be well-sorted significantly limits the application of left-commutativity equations and, as a result, solvable unification problems possess principal solutions. Furthermore, the unification algorithm for types with object-types can be obtained by a simple modification of the algorithm for simple types.

Exercise 27 ((**) Object types) *Check that the rewriting rules preserves the sorts.* □

Figure 3.2: Unification for object types

Use rules of table 1.5 where Fail excludes pairs composed of two symbols of the form $(m : \_; \_)$. and add the following rule:

$$\textsc{Mute} \qquad\qquad \text{if } m_1 \neq m_2 \text{ and } \alpha \notin \{\alpha_1, \alpha_2\} \cup ftv(e)$$

$$\frac{(m_1 : \alpha_1; \varrho_1) \doteq (m_2 : \alpha_2; \varrho_2) \doteq e}{\exists \varrho.\, (m_1 : \alpha_1; m_2 : \alpha_2; \varrho) \doteq e \wedge \varrho_1 \doteq (m_2 : \alpha_2; \varrho) \wedge \varrho_2 \doteq (m_1 : \alpha_1; \varrho)} \rightsquigarrow$$

Anonymous row variables In fact, OCaml uses yet another restriction on types, which is not mandatory but quite convenient in practice, since it avoids showing row variables to the user. This restriction is global: in any unificand we forces any two rows ending with the same row variable to be equal. Such unificands can always may be written as

$$U \wedge \bigwedge^{i \in I} \left(\exists \varrho_i.\, \langle \overline{m_i : \tau_i}; \varrho_i \rangle \doteq e_i\right)$$

where U, all τ_i's and e_i's do not contain any row variable. In such a case, we may abbreviate $\exists \varrho_i.\, \langle \overline{m_i : \tau_i}; \varrho_i \rangle \doteq e_i$ as $\langle m_i : \tau_i; 1 \rangle \doteq e$, using an anonymous row variable 1 instead of ϱ.

It is important to note that the property can always be preserved during simplification of unification problems.

Exercise 28 ((**) Unification for object types) *Give simplification rules for restricted unification problems that maintain the problem in a restricted form (using anonymous row variables).* Answer □

An alternative presentation of anonymous row variables is to use kinded types instead: The equation $\alpha \doteq \langle \overline{m : \tau}; 1 \rangle$ can be replaced by a kind constraint $\alpha :: \langle \overline{m : \tau}; 1 \rangle$ (then $\alpha \doteq \langle \overline{m : \tau}; 0 \rangle$ is also replaced by $\alpha :: \langle \overline{m : \tau}; 0 \rangle$).

Recursive types Object types may be recursive. Recursive types appear with classes that return self or that possess binary methods; they also often arise with the combination of objects that can call one another. Unquestionably, recursive types are important.

In fact, recursive types do not raise any problem at all. Without object types, *i.e.* in ML, types are terms of a free algebra, and unification with infinite terms for free algebras is well-known: deleting rule Cycle from the rewriting rules of figure 1.5 provides a unification algorithm for recursive simple types.

However, in the presence of object types, types are no longer the terms of a free algebra, since row are considered modulo left-commutativity axioms. Usually, axioms do not mix well with recursive types (in general, pathological solutions appear, and principal unifiers may be lost.) Unrestricted left-commutativity axiom is itself problematic. Fortunately, the restriction of object types by sort constraints, which limits the use of left-commutativity, makes objects-types behave well with recursive types.

More precisely, object-types can be extended with infinite terms exactly as simple types. Furthermore, a unification algorithm for recursive object types can be obtained by removing the Cycle rule from the rewriting rules of both figures 1.5 and 3.2.

Remark 7 *Allowing recursive types preserves type-soundness. However, it often turns programmers' simple mistakes, such as the omission of an argument, into well-typed programs with unexpected recursive types. (All terms of the λ-calculus —without constants— are well-typed with recursive types.) Such errors may be left undetected, or only detected at a very late stage, unless the programmer carefully check the inferred types.*

Recursive types may be restricted, e.g. *such that any recursive path crosses at least an object type constructor. Such a restriction may seem arbitrary, but it is usually preferable in practice to no restriction at all.*

Type sharing Finite types are commonly represented as trees. Recursive types, *i.e.* infinite regular trees, can be represented in several ways. The standard notation $\mu\theta.\tau$ can be used to represent the infinitely unfolded tree $\tau[\tau[\ldots/\theta]/\theta]$. Alternatively, a regular tree can also be represented as a pair $\tau \mid U$ of a term and a set of equations in canonical form. The advantage of this solution is to also represent *shared* sub-terms. For instance, the type $\alpha \to \alpha \mid \alpha \doteq \tau$ is different from the type $\tau \to \tau$ when sharing is taken into account. Sharing may be present in the source program (for instance if the user specifies type constraints;) it may also be introduced during unification. Keeping sharing increases efficiency; when displaying types, it also keeps them concise, and usually, more readable.

Moreover, type sharing is also used to keep row variables anonymous. For instance, $\langle m : \texttt{int}; \varrho\rangle \to \langle m : \texttt{int}; \varrho\rangle$, is represented as $\alpha \to \alpha$ where $\alpha \doteq \langle m : \texttt{int}; 1\rangle$. The elimination of sharing would either be incorrect or require the re-introduction of row variables.

In OCaml, type sharing is displayed using the `as` construct. A shared sub-term is printed as a variable, and its equation is displayed in the tree at the left-most outer-most occurrence of the sub-term. The last example is thus displayed as $(\langle m : \texttt{int}; 1\rangle\ \texttt{as}\ \alpha) \to \alpha$. The previous example is simply $(\tau\ \texttt{as}\ \alpha) \to \alpha$. The recursive type $\mu\alpha.\langle m : \texttt{int} \to \alpha\rangle$, which is represented by α where $\alpha \doteq \langle m : \texttt{int} \to \alpha\rangle$, is displayed as $(\langle m : \texttt{int} \to \alpha\rangle\ \texttt{as}\ \alpha)$.

Remark that the `as` construct binds globally (while in $(\mu\alpha.\tau) \to \alpha$, the variable α used to name the recursive sub-term of the left branch is a free variable of the right branch).

Type abbreviations Although object types are structural, there are also named, so as to be short and readable. This is done using type abbreviations, generated by classes, and introduced when taking object instances.

Type abbreviations are transparent, *i.e.* they can be replaced at any time by their definitions. For instance, consider the following example

class `c` = **object method** `m` = 1 **end**;;

class *`c` :* ***object method*** *`m` : `int`* ***end***

let `f x` = `x#m` **in let** `p` = **new** `c` **in** `f p`;;

The function `f` expects an argument of type $\langle m : \alpha; 1\rangle$ while `p` has type `c`. If `c` were a regular type symbol, the unification of those two types would fail. However, since it is an abbreviation, the unification can proceed, replacing `c` by its definition $\langle m : \texttt{int}\rangle$, and finally returning the substitution $\alpha \mapsto \texttt{int}$.

3.2.2 Typing classes

Typechecking classes is eased by a few design choices. First, we never need to guess types of classes, because the only form of abstraction over classes is via functors, where types must be declared. Second, the distinction between fields and methods is used to make fields never visible in objects types; they can be accessed only indirectly via method calls. Last, a key point for both simplicity and expressiveness is to type classes as if they were taking self as a parameter; thus, the type of self is an open object type that collects the minimum set of constraints required to type the body of the class. In a subclass a refined version of self-type with more constraints can then be used.

In summary, the type of a basic class is a triple $\zeta(\tau)(F; M)$ where τ is the type of self, F the list of fields with their types, and M the list of methods with their types. However, classes can also be parameterized by values, *i.e.* functions from values to classes. Hence, class types also contain functional class types. More precisely, they are defined by the following grammar (we use letter φ for class types):

$$\varphi ::= \zeta(\tau)(F; M) \mid \tau \to \varphi$$

Class bodies Typing class bodies can first be explored on a small example. Let us consider the typing of class **object** $u{=}a_u; m{=}\varsigma(x)\, a_m$ **end** in a typing context A. This class defines a field u and a method m, so it should have a class type of the form $\zeta(\tau)(u : \tau_u; m : \tau_m)$. The computation of fields of an object takes place before to the creation of the object itself. So as to prevent from accessing yet undefined fields, neither methods nor self are visible in field expressions. Hence, the expression a_u is typed in context A. That is, we must have $A \vdash a_u : \tau_u$. On the contrary, the body of the method m can see self and the field u, of types τ and τ_u, respectively. Hence, we must have $A, x : \tau, u : \tau_u \vdash a_m : \tau_m$. Finally, we check that the type assumed for the m method in the type of

Figure 3.3: Typing rules for class bodies

$$\frac{\textsc{Empty}}{A \vdash \emptyset : \zeta(\tau)(\emptyset; \emptyset)} \qquad \overset{\textsc{Field}}{\frac{A \vdash B : \zeta(\tau)(F; M) \qquad A \vdash a : \tau'}{A \vdash (B, u = a) : \zeta(\tau)(F \oplus u : \tau'; M)}}$$

$$\overset{\textsc{Method}}{\frac{A \vdash B : \zeta(\tau)(F; M) \qquad A, x : \tau, F \vdash a : \tau'}{A \vdash (B, m = \varsigma(x)\, a) : \zeta(\tau)(F; M \oplus m : \tau')}}$$

$$\overset{\textsc{Inherit}}{\frac{A \vdash B : \zeta(\tau)(F; M) \qquad A \vdash d : \zeta(\tau)(F'; M')}{A \vdash B \ \mathbf{inherit}\ d : \zeta(\tau)(F \oplus F'; M \oplus M')}}$$

self is the type inferred for method m in the class body. That is, *i.e.* we must have $\tau = \langle m : \tau_m; \rho \rangle$.

The treatment of the general case uses an auxiliary judgment $A \vdash B : \zeta(\tau)(F; M)$ to type the class bodies, incrementally, considering declarations from left to right. The typing rules for class bodies are given in figure 3.3. To start with, an empty body defines no field and no method and leaves the type of self unconstrained (rule Empty). A field declaration is typed in the current environment and the type of body is enriched with the new field type assumption (rule Field). A method declaration is typed in the current environment extended with the type assumption for self, and all type assumptions for fields; then the type of the body is enriched with the new method type assumption (rule Method). Last, an inheritance clause simply combines the type of fields and methods from the parent class with those of current class; it also ensures that the type of self in the parent class and in the current class are compatible. Fields or methods defined on both sides should have compatible types, which is indicated by the $\oplus$ operator, standing for compatible union.

Figure 3.4: Typing rules for class expressions

$$\textsc{Object}\quad \frac{A \vdash B : \zeta(\tau)(F; M) \qquad \tau = \langle M; \rho \rangle}{A \vdash \textbf{object } B \textbf{ end} : \zeta(\tau)(F; M)} \qquad\qquad \textsc{Class-Var}\quad \frac{d : \forall \overline{\alpha}.\, \varphi}{A \vdash d : \varphi[\overline{\tau}/\overline{\alpha}]}$$

$$\textsc{Class-Fun}\quad \frac{A, x : \tau \vdash d : \varphi}{A \vdash \lambda x.d : \tau \to \varphi} \qquad\qquad \textsc{Class-App}\quad \frac{A \vdash d : \tau \to \varphi \qquad A \vdash a : \tau}{A \vdash d\, a : \varphi}$$

Figure 3.5: Extra typing rules for expressions

$$\textsc{Class}\quad \frac{A \vdash d : \varphi \qquad A, z : \forall (ftv(\varphi) \setminus ftv(A)).\, \varphi \vdash a : \tau}{A \vdash \textbf{class } z = d \textbf{ in } a : \tau}$$

$$\textsc{New}\quad \frac{A \vdash d : \zeta(\tau)(F; M) \qquad \tau = \langle M; 0 \rangle}{A \vdash \texttt{new}\ d : \tau}$$

Class expressions The rules for typing class expressions, described in Figure 3.4, are quite simple. The most important of them is the Object rule for the creation of a new class: once the body is typed, it suffices to check that the type of self is compatible with the types of the methods of the class. The other rules are obvious and similar to those for variables, abstraction and application in Core ML.

Finally, we must also add the rules of Figure 3.5, for the two new forms of expressions. A class binding **class** $z = d$ **in** a is similar to a let-binding (rule Class): the type of the class d is generalized and assigned to the class name z before typing the expression a. Thus, when the class z is inherited in a, its class type is an instance of the class type of d. Last, the creation of objects is typed by constraining the type of self to be exactly the type of the methods of the class (rule New). Note the difference with Object rule where the type of self may contain methods that are not yet defined in the class body. These methods would be flagged `virtual` in OCaml. Then the class itself would be virtual, which would prohibit taking any instance. Indeed, the right premise of the New rule would fail in this case. Of course, the New rule enforces that all methods that are used recursively, *i.e.* bound in present type of self, are also defined.

Mutable fields The extension with mutable fields is mostly orthogonal to the object-oriented aspect. This could use an operation semantics with store as in Section 2.2.

Then, methods types should also see for each a field assignment primitive ($u \leftarrow \_$) for every field u of the class. Thus the Method typing rule could be changed to

$$\frac{\begin{array}{l}\textsc{Method}\\ A \vdash B : \zeta(\tau)(F; M) \qquad A, x : \tau, F, F^{\leftarrow} \vdash a : \tau'\end{array}}{A \vdash (B, m = \varsigma(x)\, a) : \zeta(\tau)(F; M \oplus m : \tau')}$$

where $F^{\leftarrow}$ stands for $\{(u \leftarrow \_ : F(u) \rightarrow \texttt{unit}) \mid u \in \mathit{dom}\,(F)\}$.

Since now the creation of objects can extend the store, the New rule should be treated as an application, *i.e.* preventing type generalization, while with applicative objects it could be treated as a non-expansive expression and allow generalization.

Overriding As opposed to assignment, overriding creates a new fresh copy of the object where the value of some fields have been changed. This is an atomic operation, hence the overriding operation should take a list of pairs, each of which being a field to be updated and the new value for this field.

Hence, to formalize overriding, we assume given a collection of primitives $\{\langle u_1 = \_; \ldots; u_n = \_\rangle\}$ for all $n \in \mathbb{N}$ and all sets of fields $\{u_1, \ldots u_n\}$ of size n. As for assignment, rule methods should make some of these primitives visible in the body of the method, by extending the typing

Figure 3.6: Closure and consistency rules for subtyping

Closure rules

$$
\begin{array}{rcl}
\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' & \Longrightarrow & \tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2' \\
\langle \tau \rangle \leq \langle \tau' \rangle & \Longrightarrow & \tau \leq \tau' \\
(m : \tau_1; \tau_2) \leq (m : \tau_1'; \tau_2') & \Longrightarrow & \tau_1 \leq \tau_1' \wedge \tau_2 \leq \tau_2'
\end{array}
$$

Consistency rules

$$
\begin{array}{rcl}
\tau \leq \tau_1 \to \tau_2 & \Longrightarrow & \tau \text{ is of the shape } \tau_1' \to \tau_2' \\
\tau \leq \langle \tau_0 \rangle & \Longrightarrow & \tau \text{ is of the shape } \langle \tau_0' \rangle \\
\tau \leq (m : \tau_1; \tau_2) & \Longrightarrow & \tau \text{ is of the shape } (m : \tau_1'; \tau_2') \\
\tau \leq \mathtt{Abs} & \Longrightarrow & \tau = \mathtt{Abs} \\
\tau \leq \alpha & \Longrightarrow & \tau = \alpha
\end{array}
$$

environment of the Method rule of Figure 3.3. We use the auxiliary notation $\{\langle u_1 : \tau_1; \dots u_n : \tau_n \rangle\}^\tau$ for the typing assumption

$$(\{\langle u_1 = \_; \dots u_n = \_\rangle\} : \tau_1 \to \dots \tau_n \to \tau)$$

and $F \star \tau$ the typing environment $\bigcup_{F' \subset F} \{\langle F' \rangle\}^\tau$. Then, the new version of the Method rule is:

$$
\textsc{Method}\quad
\frac{A \vdash B : \zeta(\tau)(F; M) \qquad A, x : \tau, F, F \star \tau \vdash a : \tau'}{A \vdash (B, m = \varsigma(x)\, a) : \zeta(\tau)(F; M \oplus m : \tau')}
$$

Subtyping Since uses of subtyping are explicit, they do not raise any problem for type inference. In fact, subtyping coercions can be typed as applications of primitives. We assume a set of primitives $(\_ : \tau_1 :> \tau_2)$ of respective type scheme $\forall \bar{\alpha}.\, \tau_1 \to \tau_2$ for all pairs of types such that $\tau_1 \leq \tau_2$. Note that the types τ_1 and τ_2 used here are given and not inferred.

The subtyping relation $\leq$ is standard. It is structural, covariant for object types and on the right hand side of the arrow, contravariant on the left hand side of the arrow, and non-variant on other type constructors. Formally, the relation $\leq$ can be defined as the largest transitive relation on regular trees that satisfies the closure and consistency rules of figure 3.6:

Subtyping should not be confused with inheritance. First, the two relations are defined between elements of different sets: inheritance relates classes, while subtyping relates object types (not even class types). Second, there is no obvious correspondence between the two relations. On the one hand, as shown by examples with binary methods, if two classes are in an inheritance relation, then the types of objects of the respective classes are not necessarily in a subtyping relation. On the other hand, two classes that are implemented independently are not in an inheritance relation; however, if they implement the same interface (*e.g.* in particular if they are identical), the types of objects of these classes will be equal, hence in a subtyping relation. (The two classes will define two different abbreviations for the same type.) This can be checked on the following program:

```
class c1 = object end
class c2 = object end;;
fun x -> (x : c1 :> c2);;
```

We have `c1` $\leq$ `c2` but `c1` does not inherit from `c2`.

Exercise 29 (Project —type inference for objects) *Extend the small type checker given for the core calculus to include objects and classes.* □

3.3 Advanced uses of objects

We present here a large, realistic example that illustrates many facets of objects and classes and shows the expressiveness of Objective Caml.

The topic of this example is the modular implementation of window managers. Selecting the actions to be performed (such as moving or redisplaying windows) is the managers' task. Executing those actions is the windows' task. However, it is interesting to generalize this example into a design pattern known as the *subject-observer*. This design pattern has been a challenge [10]. The observers receive information from the subjects and, in return, request actions from them. Symmetrically, the subjects execute the requested actions and communicate any useful information to their observers. Here, we chose a protocol relying on trust, in which the subject asks to be observed: thus, it can manage the list of its observers himself. However, this choice could really be inverted and a more authoritative protocol in which the master (the observer) would manage the list of its subjects could be treated in a similar way.

Unsurprisingly, we reproduce this pattern by implementing two classes modeling subjects and observers. The class `subject` that manages the list of observers must be parameterized by the type `'observer` of the objects of the class `observer`. The class `subject` implements a method `notify` to relay messages to all observers, transparently. A piece of information is represented by a procedure that takes an observer as parameter; the usage is that this procedure calls an appropriate message of the observer; the name of the message and its arguments are hidden in the procedure closure. A message is also parameterized by the sender (a subject); the method `notify` applies messages to their sender before broadcasting them, so that the receiver may call back the sender to request a new action, in return.

```
class ['observer] subject =
  object (self : 'mytype)
    val mutable observers : 'observer list = []
    method add obs = observers <- obs :: observers
    method notify (message : 'observer -> 'mytype -> unit) =
      List.iter (fun obs -> message obs self) observers
  end;;
```

The template of the observer does not provide any particular service and is reduced to an empty class:

```
class ['subject] observer = object end;;
```

To adapt the general pattern to a concrete case, one must extend, in parallel, both the `subject` class with methods implementing the actions that the observer may invoke and the `observer` class with informations that the subjects may send. For instance, the class `window` is an instance of the class `subject` that implements a method `move` and notifies all observers of its movements by calling the `moved` method of observers. Consistently, the manager inherits from the class `observer` and implements a method `moved` so as to receive and treat the corresponding notification messages sent by windows. For example, the method `moved` could simply call back the `draw` method of the window itself.

```
class ['observer] window =
  object (self : 'mytype)
    inherit ['observer] subject
    val mutable position = 0
    method move d =
      position <- position + d; self#notify (fun x -> x#moved)
    method draw = Printf.printf "[Position = %d]" position;
  end;;

class ['subject] manager =
  object
    inherit ['subject] observer
    method moved (s : 'subject) : unit =  s#draw
  end;;
```

An instance of this pattern is well-typed since the manager correctly treats all messages that are send to objects of the `window` class.

```
let w = new window in w#add (new manager); w#move 1;;
```

This would not be the case if, for instance, we had forgotten to implement the `moved` method in the `manager` class.

The subject-observer pattern remains modular, even when specialized to the window-manager pattern. For example, the window-manager pattern can further be refined to notify observers when windows are resized. It suffices to add a notification method `resize` to windows and, accordingly, an decision method `resized` to managers:

```
class ['observer] large_window =
  object (self)
    inherit ['observer] window as super
    val mutable size = 1
    method resize x =
      size <- size + x; self#notify (fun x -> x#resized)
    method draw = super#draw; Printf.printf "[Size = %d]" size;
  end;;
```

```
class ['subject] big_manager =
  object
    inherit ['subject] manager as super
    method resized (s:'subject) = s#draw
  end;;
```

Actually, the pattern is quite flexible. As an illustration, we now add another kind of observer used to spy the subjects:

```
class ['subject] spy =
  object
    inherit ['subject] observer
    method resized (s:'subject) = print_string "<R>"
    method moved (s:'subject) = print_string "<M>"
  end;;
```

To be complete, we test this example with a short sequence of events:

```
let w = new large_window in
   w#add (new big_manager); w#add (new spy);
   w#resize 2; w#move 1;;
```

```
 <R>[Position = 0][Size = 3]<M>[Position = 1][Size = 3]- : unit = ()
```

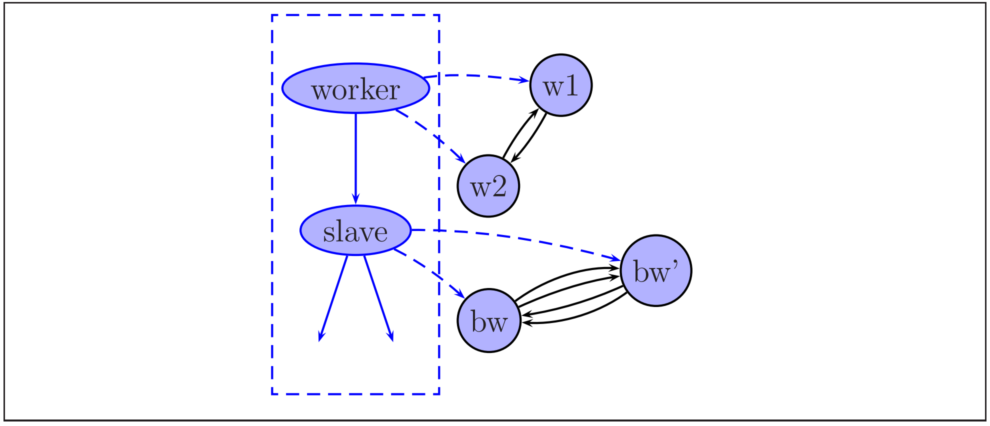
Exercise 30 (Project —A widget toolkit) *Implement a widget toolkit from scratch,* i.e. *using the Graphics library. For instance, starting with rectangular areas as basic widgets, containers, text area, buttons, menus, etc. can be derived objects. To continue, scroll bars, scrolling rectangles, etc. can be added.*

The library should be design with multi-directional modularity in mind. For instance, widgets should be derived from one another as much as possible so as to ensure code sharing. Of course, the user should be able to customize library widgets. Last, the library should also be extensible by an expert.

In additional to the implementation of the toolkit, the project could also illustrate the use of the toolkit itself on an example. ☐

The subject/observer pattern is an example of component inheritance. With simple object-oriented programming, inheritance is related to a single class. For example, figure 3.7 sketches a common, yet advanced situation where several objects of the same `worker` class interact

Figure 3.7: Traditional inheritance



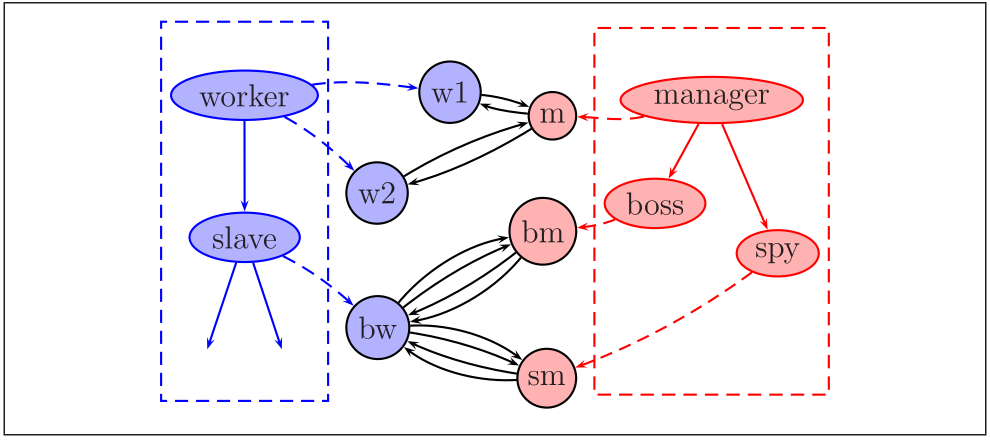
intimately, for instance through binary methods. In an inherited `slave` class, the communication pattern can then be enriched with more connections between objects of the same class. This pattern can easily be implemented in OCaml, since binary methods are correctly typed in inherited classes, as shown on examples in Section 3.1.2.

A generalization of this pattern is often used in object-oriented components. Here, the intimate connection implies several objects of related but different classes. This is sketched in figure 3.8 where objects of the `worker` class interact with objects of the `manager` class. What is then often difficult is to allow inheritance of the components, such that objects of the subclasses can have an enriched communication pattern and still interact safely. In the sketched example, objects of the `slave` class on the one hand and object of `boss` or `spy` classes on the other hand do interact with a richer interface.

The subject/observer is indeed an instance of this general pattern. As shown above, it can be typed successfully. Moreover, all the expected flexibility is retained, including in particular, the refinement of the communication protocol in the sub-components.

The key ingredient in this general pattern is, as for binary methods, the use of structural open object types and their parametric treatment in subclasses. Here, not only the selftype of the current class, but also

Figure 3.8: Component inheritance



the selftype the other classes recursively involved in the pattern are abstracted in each class.

Further reading

The addition of objects and classes to OCaml was first experimented in the language ML-ART [64] —an extension of ML with abstract types and record types— in which objects were not primitive but programmed. Despite some limitations imposed in OCaml, for sake of simplification, and on the opposite some other extensions, ML-ART can be still be seen as an introspection of OCaml object oriented features. Conversely, the reader is referred to [66, 72] for a more detailed (and more technical) presentation.

Moby [20] is another experiment with objects to be mentioned because it has some ML flavor despite the fact that types are no longer inferred. However, classes are more closely integrated with the module system, including a view mechanism [68].

A short survey on the problem of binary methods is [9]. The "Subject/Observer pattern" and other solutions to it are also described in [10]. Of course, they are also many works that do not consider type inference.

A good but technical reference book is [1].

Chapter 4

The module language

Independently of classes, Objective Caml features a powerful module system, inspired from the one of Standard ML.

The benefits of modules are numerous. They make large programs *compilable* by allowing to split them into pieces that can be separately compiled. They make large programs *understandable* by adding structure to them. More precisely, modules encourage, and sometimes force, the specification of the links (interfaces) between program components, hence they also make large programs *maintainable* and *reusable*. Additionally, by enforcing abstraction, modules usually make programs *safer*.

4.1 Using modules

Compared with other languages already equipped with modules such as Modular-2, Modula-3, or Ada, the originality of the ML module system is to be a small typed functional language "on top" of the base language. The ML module system can actually be *parameterized* by the base language, which need not necessarily be ML. Thus, it could provide a language for modules other base languages.

4.1.1 Basic modules

Basic modules are *structures*, *i.e.* collections of phrases, written **struct** $p_1 \ldots p_n$ **end**. Phrases are those of the core language, plus definitions of sub modules **module** $X = M$ and of module types **module type** $T = S$. Our first example is an implementation of stacks.

```
module Stack =
  struct
    type 'a t = {mutable elements : 'a list }
    let create () = { elements = [] }
    let push x s = s.elements <- x :: s.elements
    let pop s =
      match s.elements with
        h::t -> s.elements <- t; h
      | []   -> failwith ”Empty stack”
  end;;
```

Components of a module are referred to using the “dot notation”:

```
let s = Stack.create () in Stack.push 1 s; Stack.push 2 s; Stack.pop s;;
```

```
− : int = 2
```

Alternatively, the directive `open` S allows to further skip the prefix and the dot, simultaneously: **struct** `open` $S \ldots (f\ x : t) \ldots$ **end**. A module may also be a subcomponent of another module:

```
module T =
  struct
    module R = struct let x = 0 end
    let y = R.x + 1
  end
```

The “dot notation” and `open` extends to and can be used in sub-modules. Note that the directive `open T.R` in a module `Q` makes all components of `T.R` visible to the rest of the module `Q` but it does not add these components to the module `Q`.

The system infers signatures of modules, as it infers types of values. Types of basic modules, called *signatures*, are sequences of (type) specifications, written **sig** $s_1 \ldots s_n$ **end**. The different forms of specifications are described in figure 4.1. For instance, the system's answer to the

Figure 4.1: Specifications

Specification of	**form**
values	`val` $x : \sigma$
abstract types	`type` t
manifest types	`type` $t = \tau$
exceptions	`exception` E
classes	`class` z `:object` … `end`
sub-modules	`module` $X : S$
module types	`module type` T $[\ = M]$

`Stack` example was:

```
module Stack :
  sig
    type 'a t = { mutable elements : 'a list; }
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
  end
```

An explicit signature constraint can be used to restrict the signature inferred by the system, much as type constraints restrict the types inferred for expressions. Signature constraints are written $(M : S)$ where M is a module and S is a signature. There is also the syntactic sugar **module** $X : S = M$ standing for **module** $X = (M : S)$.

Precisely, a signature constraint is two-fold: first, it checks that the structure complies with the signature; that is, all components specified in S must be defined in M, with types that are at least as general; second, it makes components of M that are not components of S inaccessible. For instance, consider the following declaration:

```
module S : sig type t val y : t end =
  struct type t = int let x = 1 let y = x + 1 end
```

Then, both expressions `S.x` and `S.y + 1` would produce errors. The former, because x is not externally visible in `S`. The latter because the component `S.y` has the abstract type `S.t` which is not compatible with type `int`.

Signature constraints are often used to enforce type abstraction. For instance, the module `Stack` defined above exposes its representation. This allows stacks to be created directly without calling `Stack.create`.

```
Stack.pop { Stack.elements = [2; 3]};;
```

However, in another situation, the implementation of stacks might have assumed invariants that would not be verified for arbitrary elements of the representation type. To prevent such confusion, the implementation of stacks can be made abstract, forcing the creation of stacks to use the function `Stack.create` supplied especially for that purpose.

```
module Astack :
    sig
      type 'a t
      val create : unit -> 'a t
      val push : 'a -> 'a t -> unit
      val pop : 'a t -> 'a
    end = Stack;;
```

Abstraction may also be used to produce two isomorphic but incompatible views of a same structure. For instance, all currencies are represented by floats; however, all currencies are certainly not equivalent and should not be mixed. Currencies are isomorphic but disjoint structures, with respective incompatible units `Euro` and `Dollar`. This is modeled in OCaml by a signature constraint.

```
module Float =                    module type CURRENCY =
  struct                            sig
    type t = float                    type t
    let unit = 1.0                    val unit  : t
    let plus = (+.)                   val plus : t -> t -> t
    let prod = ( *. )                 val prod : float -> t -> t
  end;;                             end;;
```

Remark that multiplication became an external operation on floats in the signature `CURRENCY`. Constraining the signature of `Float` to be

`CURRENCY` returns another, incompatible view of `Float`. Moreover, repeating this operation returns two isomorphic structures but with incompatible types `t`.

```
module Euro = (Float : CURRENCY);;
module Dollar = (Float : CURRENCY);;
```

In `Float` the type `t` is concrete, so it can be used for ”float”. Conversely, it is abstract in modules `Euro` and `Dollar`. Thus, `Euro.t` and `Dollar.t` are incompatible.

```
let euro x = Euro.prod x Euro.unit;;
Euro.plus (euro 10.0) (euro 20.0);;

Euro.plus (euro 50.0) Dollar.unit;;
```

Remark that there is no code duplication between `Euro` and `Dollar`.

A slight variation on this pattern can be used to provide multiple views of the same module. For instance, a module may be given a restricted interface in a given context so that certain operations (typically, the creation of values) would not be permitted.

```
module type PLUS =                module type PLUS_Euro =
  sig                               sig
    type t                            type t = Euro.t
    val plus : t -> t -> t            val plus : t -> t -> t
  end;;                             end;;
module Plus = (Euro : PLUS)       module Plus = (Euro : PLUS_Euro)
```

On the left hand side, the type `Plus.t` is incompatible with `Euro.t`. On the right, the type `t` is partially abstract and compatible with `Euro.t`; the view `Plus` allows the manipulation of values that are built with the view `Euro`. The `with` notation allows the addition of type equalities in a (previously defined) signature. The expression `PLUS with type t = Euro.t` is an abbreviation for the signature

```
  sig
    type t = Euro.t
    val plus: t -> t -> t
  end
```

The `with` notation is a convenience to create partially abstract signatures and is often inlined:

```
module Plus = (Euro : PLUS with type t = Euro.t);;
Plus.plus Euro.unit Euro.unit;;
```

Separate compilation Modules are also used to facilitate separate compilation. This is obtained by matching toplevel modules and their signatures to files as follows. A compilation unit `A` is composed of two files:

- The implementation file `a.ml` is a sequence of phrases, like phrases within **struct** . . . **end**.

- The interface file `a.mli` (optional) is a sequence of specifications, such as within `sig... end`.

Another compilation unit `B` may access `A` as if it were a structure, using either the dot notation `A.x` or the directive `open A`. Let us assume that the source files are: `a.ml`, `a.mli`, `b.ml`. That is, the interface of a `B` is left unconstrained. The compilations steps are summarized below:

Command	**Compiles**	**Creates**
`ocamlc -c a.mli`	interface of `A`	`a.cmi`
`ocamlc -c a.ml`	implementation of `A`	`a.cmo`
`ocamlc -c b.ml`	implementation of `B`	`b.cmo`
`ocamlc -o myprog a.cmo b.cmo`	linking	`myprog`

The program behaves as the following monolithic code:

```
module A : sig (* content of a.mli *) end =
  struct (* content of a.ml *) end
module B = struct (* content of b.ml *) end
```

The order of module definitions correspond to the order of `.cmo` object files on the linking command line.

4.1.2 Parameterized modules

A *functor*, written **functor** $(S : T) \to M$, is a function from modules to modules. The body of the functor M is explicitly parameterized by the module parameter S of signature T. The body may access the components of S by using the dot notation.

```
module M = functor(X : T) ->
  struct
    type u = X.t * X.t
    let y = X.g(X.x)
  end
```

As for functions, it is not possible to access directly the body of `M`. The module `M` must first be explicitly applied to an implementation of signature `T`.

```
    module T1 = T(S1)
    module T2 = T(S2)
```

The modules `T1`, `T2` can then be used as regular structures. Note that `T1` et `T2` share their code, entirely.

4.2 Understanding modules

We refer here to the literature. See the bibliography notes below for more information of the formalization of modules [27, 44, 45, 69].

For more information on the implementation, see [46].

4.3 Advanced uses of modules

In this section, we use the running example of a bank to illustrate most features of modules and combined them together.

Let us focus on bank accounts and, in particular, the way the bank and the client may or may not create and use accounts. For security purposes, the client and the bank should obviously have different access privileges to accounts. This can be modeled by providing different views of accounts to the client and to the bank:

```
module type CLIENT =          (* client's view *)
  sig
    type t
    type currency
    val deposit : t -> currency -> currency
    val retrieve : t -> currency -> currency
  end;;

module type BANK =          (* banker's view *)
  sig
    include CLIENT
    val create : unit -> t
  end;;
```

We start with a rudimentary model of the bank: the account book is given to the client. Of course, only the bank can create the account, and to prevent the client from forging new accounts, it is given to the client, abstractly.

```
module Old_Bank (M : CURRENCY) :
    BANK with type currency = M.t =
  struct
    type currency = M.t
    type t = { mutable balance : currency }
    let zero = M.prod 0.0 M.unit and neg = M.prod (-1.0)

    let create() = { balance = zero }
    let deposit c x =
      if x > zero then c.balance <- M.plus c.balance x; c.balance
    let retrieve c x =
      if c.balance > x then deposit c (neg x) else c.balance
  end;;

module Post = Old_Bank (Euro);;
module Client :
    CLIENT with type currency = Post.currency and type t = Post.t
        = Post;;
```

This model is fragile because all information lies in the account itself.

For instance, if the client loses his account, he loses his money as well, since the bank does not keep any record. Moreover, security relies on type abstraction to be unbreakable…

However, the example already illustrates some interesting benefits of modularity: the clients and the banker have different views of the bank account. As a result an account can be created by the bank and used for deposit by both the bank and the client, but the client cannot create new accounts.

```
let my_account = Post.create ();;
Post.deposit my_account (euro 100.0);
Client.deposit my_account (euro 100.0);;
```

Moreover, several accounts can be created in different currencies, with no possibility to mix one with another, such mistakes being detected by typechecking.

```
module Citybank = Old_Bank (Dollar);;
let my_dollar_account = Citybank.create();;

Citybank.deposit my_account;;
Citybank.deposit my_dollar_account (euro 100.0);;
```

Furthermore, the implementation of the bank can be changed while preserving its interface. We use this capability to build, a more robust —yet more realistic— implementation of the bank where the account book is maintained in the bank database while the client is only given an account number.

```
module Bank (M : CURRENCY) : BANK with type currency = M.t =
  struct
    let zero = M.prod 0.0 M.unit and neg = M.prod (−1.0)
    type t = int
    type currency = M.t

    type account = { number : int; mutable balance : currency }
    (* bank database *)
    let all_accounts = Hashtbl.create 10 and last = ref 0
    let account n = Hashtbl.find all_accounts n

    let create() = let n = incr last; !last in
```

```
      Hashtbl.add all_accounts n {number = n; balance = zero}; n

    let deposit n x = let c = account n in
      if x > zero then c.balance <- M.plus c.balance x; c.balance

    let retrieve n x = let c = account n in
      if c.balance > x then (c.balance <- M.plus c.balance x; x)
      else zero
  end;;
```

Using functor application we can create several banks. As a result of generativity of function application, they will have independent and private databases, as desired.

```
module Central_Bank = Bank (Euro);;
module Banque_de_France = Bank (Euro);;
```

Furthermore, since the two modules `Old_bank` and `Bank` have the same interface, one can be used instead of the other, so as to created banks running on different models.

```
module Old_post = Old_Bank(Euro)
module Post = Bank(Euro)
module Citybank = Bank(Dollar);;
```

All banks have the same interface, however they were built. In fact, it happens to be the case that the user cannot even observe the difference between either implementation; however, this would not be true in general. Indeed, such a property can not be enforced by the typechecker.

Exercise 31 (Polynomials with one variable)

1. *Implement a library with operations on polynomials with one variable.*

   The coefficients form a ring that is given as a parameter to the library.

2. *Use the library to check, for instance, the identity* $(1+X)(1-X) = 1-X^2$.

3. *Check the equality* $(X+Y)(X-Y) = (X^2 - Y^2)$ *by treating polynomials with two variables as polynomials with one variable* X *and where the coefficients are the ring of the polynomials with one variable* Y.

4. *Write a program that reads a polynomial on the command line and evaluates it at each of the points given in* `stdin` *(one integer per line); the result should be printed in* `stdout`.

□

Chapter 5

Mixing modules and objects

Modules and classes play different roles. On the one hand, modules can be embedded, and parameterized by types and values. Modules also allow value and type abstraction on a large scale. On the other hand, classes provide inheritance and its late binding mechanism; they can also be parameterized by values, but on a small scale. At first, there seems to be little redundancy. Indeed, to benefit from both features simultaneously, modules and classes are often combined together.

However, both modules and classes provide means of structuring the code. Despite their resemblance at first glance, modular and object-oriented programming styles are in fact diverging: once a choice has been made to represent a structure by a module or by an object, many other choices are forced, which is sometimes bothersome.

In this Chapter, we discusses the overlapping of features and the specificities, and show how to use them in harmony.

5.1 Overlapping

Many abstract datatypes can be defined using either classes or modules. A representative example is the type of stacks. Stacks have been defined as a parametric class in section 3.1.2 where operations on stacks are methods embedded into stack-objects, and as a module defining an abstract type of stacks and the associated operations in section 4.1.1.

The following table summarizes the close correspondence between those two implementations.

	class version	module version
The type of stacks	`'a stack`	`'a Astack.t`
Create a stack	`new stack`	`Astack.created ()`
Push x on s	s`#push` x	`Astack.push` x s
Pop from s	s`#pop`	`Astack.pop` s

More generally, all algebraic abstract types that are modifiable in place and such that all associated operations are unary (*i.e.* they take only one argument of the abstract type) can be defined as classes or as modules in almost the same way. Here, the choice between those two forms is mainly a question of programming style: some programmers prefer to see the operations of the abstract type as methods attached to values of this type, while others prefer to see them as functions outside values of the abstract type.

Moreover, the two alternatives are still comparable when extending the stack implementation with new operations. For instance, in both cases, a new implementation of stacks can be derived from the older one with an additional method `top` to explore the top of the stack without popping it. Both implementations, described in Figure 5.1, are straightforward: the class approach uses inheritance while the module approach uses the `include` construct. (The effect of `include Stack` is to rebind all components of the `Stack` structure in the substructure.)

However, the two approaches definitely differ when considering late binding, which is only possible with the class approach. For instance, redefining the implementation of `pop` would automatically benefit to the implementation of `top` (*i.e.* the method `top` would call the new definition of `pop`). This mechanism does not have any counterpart in OCaml modules.

In conclusion, if inheritance is needed, the class approach seems more appropriate, and it becomes the only possible (direct) solution if, moreover, late binding is required. Conversely, for abstract datatypes used with binary operations, such as sets with a merge operation, the module approach will be preferable, as long as inheritance is not used. Further-

Figure 5.1: Class *v.s.* Module versions of stacks

```
class ['a] stack_ext =
  object (self)
    inherit ['a] stack
    method top =
      let s = self#pop in
      self#push s; s
end;;
```

```
module StackExt =
  struct
    include Stack
    let top p =
      let s = pop p in
      push s p; s
  end;;
```

more, the module approach is the only one that allows to return private data to the user, but abstractly, so as to preserve its integrity. Of course, the module system is also the basis for separate compilation.

5.2 Combining modules and classes

To benefit from the advantages of objects and modules simultaneously, an application can easily combine both aspect. Typically, modules will be used at the outer level, to provide separate compilation, inner structures, and privacy outside of the module boundaries, while classes will be components of modules, and offer extendibility, open recursion and late binding mechanisms.

We first present typical examples of such patterns, with increasing complexity and expressiveness. We conclude with a more complex —but real— example combining many features in an unusual but interesting manner.

5.2.1 Classes as module components

The easiest example is probably the use of modules to simply group related classes together. For instance, two classes `nil` and `cons` that are related by their usage, can be paired together in a module.

```
module Cell = struct
  exception Nil
  class ['a] nil =
    object (self : 'alist)
      method hd : 'a = raise Nil
      method tl : 'alist = raise Nil
      method null = true
    end;;
```

```
class ['a] cons h t =
  object (_ : 'alist)
    val hd = h val tl = t
    method hd : 'a = h
    method tl : 'alist = t
    method null = false
  end;;
end;;
```

Besides clarity of code, one quickly take advantage of such grouping. For instance, the `nil` and `cons` classes can be extended simultaneously (but this is not mandatory) to form an implementation of lists:

```
module List = struct
  class ['a] nil =
    object
      inherit ['a] Cell.nil
      method length = 0
    end;;
```

```
class ['a] cons h t =
  object
    inherit ['a] Cell.cons h t
    method length = 1+tl#length
  end;;
end;;
```

In turn, the module `List` can also be extended —but in another direction— by adding a new “constructor” `append`. This amounts to adding a new class with the same interface as that of the first two.

Remark 8 *In OCaml, lists are more naturally represented by a sum data type, which moreover allows for pattern matching. However, datatypes are not extensible.*

In this example, grouping could be seen as a structuring convenience, because a flattened implementation of all classes would have worked as well. However, grouping becomes mandatory for friend classes.

Friend classes State encapsulation in objects allows to abstract their representation by hiding all instance variables. Thus, reading and writing capabilities can be controlled by providing only the necessary methods. However, whether to expose some given part of the state is an all-or-nothing choice: either it is confined to the object or revealed to the whole world.

It is often the case that some, but not all, objects can access each other's state. A typical example (but not the only one) are objects with binary methods. A binary method of an object is called with another object of the same class as argument, so as to interact with it. In most cases, this interaction should be intimate, *e.g.* depend on the details of their representations and not only on their external interfaces. For instance, only objects having the same implementation could be allowed to interact. With objects and classes, the only way to share the representation between two different objects is to expose it to the whole world.

Modules, which provide a finer-grain abstraction mechanism, can help secure this situation, making the type of the representation abstract. Then, all friends (classes or functions) defined within the same module and sharing the same abstract view know the concrete representation.

This can be illustrated on the bank example, by turning currency into a class:

```
module type CURRENCY = sig
  type t
  class c : float ->
    object ('a)
      method v : t
      method plus : 'a -> 'a
      method prod : float -> 'a
    end
end;;
module Currency = struct
  type t = float
  class c x =
    object (_ : 'a)
      val v = x method v = v
```

```
      method plus(z:'a) = {< v = v +. z#v >}
      method prod x = {< v = x *. v >}
    end
  end;;
  module Euro = (Currency : CURRENCY);;
```

Then, all object of the class `Euro.c` can be combined, still hiding the currency representation.

A similar situation arises when implementing sets with a union operation, tables with a merge operation, *etc.*

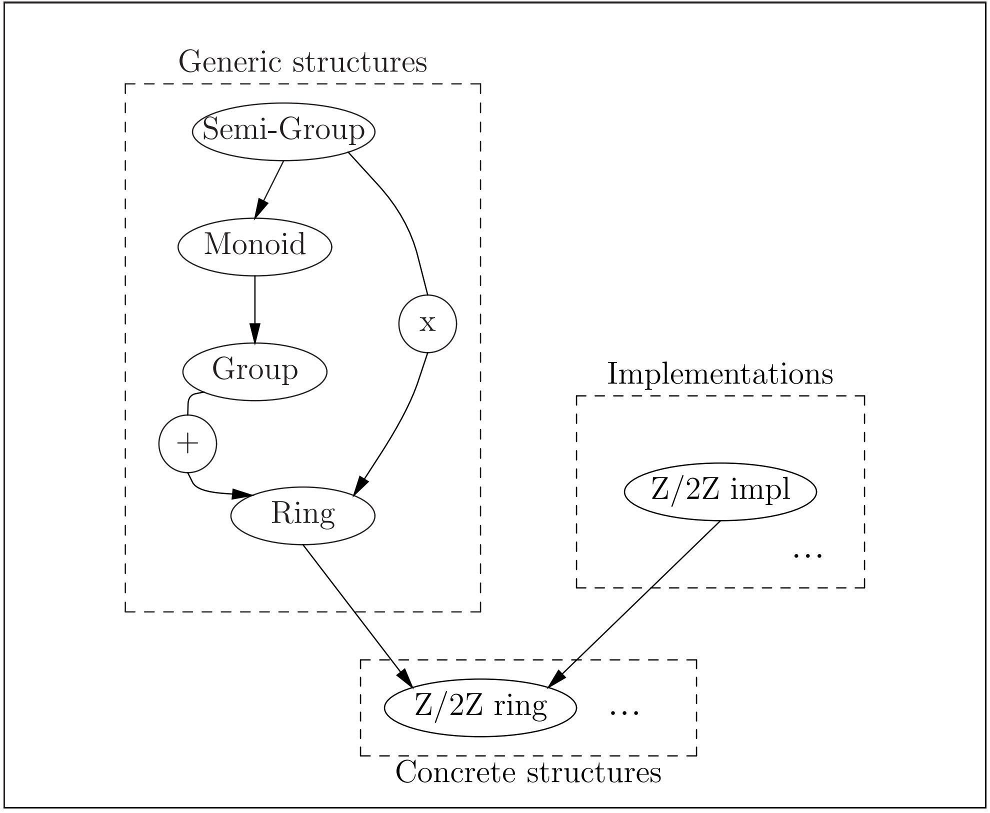
5.2.2 Classes as pre-modules

We end this Chapter with an example that interestingly combines some features of classes objects and modules. This example is taken from the algebraic-structure library of the formal computation system FOC [7]. The organization of such a library raises important problems: on the one hand, algebraic structures are usually described by successive refinements (a group is a monoid equipped with an additional inverse operation). The code structure should reflect this hierarchy, so that at least the code of the operations common to a structure and its derived structures can be shared. On the other hand, type abstraction is crucial in order to hide the real representations of the structure elements (for instance, to prevent from mixing integers modulo p and integers modulo q when p is not equal to q). Furthermore, the library should remain extensible.

In fact, we should distinguish generic structures, which are abstract algebraic structures, from concrete structures, which are instances of algebraic structures. Generic structures can either be used to derive richer structures or be instantiated into concrete structures, but they themselves do not contain elements. On the contrary, concrete structures can be used for computing. Concrete structures can be obtained from generic ones by supplying an implementation for the basic operations. This schema is sketched in figure 5.2. The arrows represent the expected code sharing.

In general, as well as in this particular example, there are two kinds of expected clients of a library: experts and final users. Indeed, a good library should not only be usable, but also re-usable. Here for instance, final users of the library only need to instantiate some generic structures to concrete ones and use these to perform computation. In addition, a few experts should be able to extend the library, providing new generic structures by enriching existing ones, making them available to the final users and to other experts.

Figure 5.2: Algebraic structures



The first architecture considered in the FOC project relies on modules, exclusively; modules facilitates type abstraction, but fails to provide code sharing between derived structures. On the contrary, the second architecture represents algebraic structures by classes and its elements by objects; inheritance facilitates code sharing, but this solution fails to provide type abstraction because object representation must be exposed, mainly to binary operations.

The final architecture considered for the project mixes classes and modules to combine inheritance mechanisms of the former with type abstraction of the latter. Each algebraic structure is represented by a module with an abstract type `t` that is the representation type of algebraic structure elements (*i.e.* its "carrier"). The object `meth`, which collects all the operations, is obtained by inheriting from the virtual class that is parameterized by the carrier type and that defines the derived operations. For instance, for groups, the virtual class `['a] group` declares the basic group operations (`equal`, `zero`, `plus`, `opposite`) and defines the derived operations (`not_equal`, `minus`) once and for all:

```
class virtual ['a] group =
  object(self)
    method virtual equal: 'a -> 'a -> bool
    method not_equal x y = not (self#equal x y)
    method virtual zero: 'a
    method virtual plus: 'a -> 'a -> 'a
    method virtual opposite: 'a -> 'a
    method minus x y = self#plus x (self#opposite y)
  end;;
```

A class can be reused either to build richer generic structures by adding other operations or to build specialized versions of the same structure by overriding some operations with more efficient implementations. The late binding mechanism is then used in an essential way.

(In a more modular version of the group structure, all methods would be private, so that they can be later ignored if necessary. For instance, a group should be used as the implementation of a monoid. All private methods are made public, and as such become definitely visible, right before a concrete instance is taken.)

A group is a module with the following signature:

```
module type GROUP =
  sig
    type t
    val meth: t group
```

```
    end;;
```

To obtain a concrete structure for the group of integers modulo p, for example, we supply an implementation of the basic methods (and possibly some specialized versions of derived operations) in a class `z_pz_impl`. The class `z_pz` inherits from the class `[int] group` that defines the derived operations and from the class `z_pz_impl` that defines the basic operations. Last, we include an instance of this subclass in a structure so as to hide the representation of integers modulo p as OCaml integers.

```
class z_pz_impl p =
  object
    method equal (x : int) y = (x = y)
    method zero = 0
    method plus x y = (x + y) mod p
    method opposite x = p − 1 − x
  end;;
class z_pz p =
  object
    inherit [int] group
    inherit z_pz_impl p
  end;;
module Z_pZ =
  functor (X: sig val p : int end) ->
   ( struct
      type t = int
      let meth = new z_pz X.p
      let inj x =
        if x >= 0 && x < X.p then x else failwith "Z_pZ.inj"
      let proj x = x
    end : sig
      type t
      val meth: t group
      val inj: int -> t
      val proj: t -> int
    end);;
```

This representation elegantly combines the strengths of modules (type abstraction) and classes (inheritance and late binding).

Exercise 32 (Project —A small subset of the FOC library) *As an exercise, we propose the implementation of a small prototype of the FOC library. This exercise is two-fold.*

On the one hand, it should include more generic structures, starting with sets, and up to at least rings and polynomials.

On the other hand, it should improve on the model given above, by inventing a more sophisticated design pattern that is closer to the model sketched in figure 5.2 and that can be used in a systematic way.

For instance, the library could provide both an open view and the abstraction functor for each generic structure. The open view is useful for writing extensions of the library. Then, the functor can be used to produce an abstract concrete structure directly from an implementation.

*The pattern could also be improved to allow a richer structure (*e.g. *a ring) to be acceptable in place only a substructure is required (*e.g. *an additive group).*

The polynomials with coefficients in $\mathbb{Z}/2\mathbb{Z}$ *offers a simple yet interesting source of examples.* □

Further reading

The example of the FOC system illustrates a common situation that calls for hybrid mechanisms for code structuring that would more elegantly combine the features of modules and classes. This is an active research area, where several solutions are currently explored. Let us mention in particular "mixin" modules and objects with "views". The former enrich the ML modules with inheritance and a late binding mechanism [18, 4, 5]. The latter provide a better object-encapsulation mechanism, in particular in the presence of binary operations and "friend" functions; views also allow to forget or rename methods more freely [68, 71].

Other object-oriented languages, such as CLOS, detach methods from objects, transforming them into overloaded functions. This approach is becoming closer to traditional functional programming. Moreover, it extends rather naturally to multi-methods [13, 22, 8] that allow to recover the symmetry between the arguments of a same algebraic type. This approach is also more expressive, since method dispatch may depend on several arguments simultaneously rather than on a single one in a privileged position. However, this complicates abstraction of object representation. Indeed, overloading makes abstraction more difficult, since the precise knowledge of the type of arguments is required to decide what version of the method should be used.

Further reading

The OCaml compiler, its programming environment and its documentation are available at the Web site `http://caml.inria.fr`. The documentation includes the reference manual of the language and some tutorials.

The recent book of Chailloux, Manoury and Pagano [12] is a complete presentation of the OCaml language and of its programming environment. The book is written in French, but an English version should be soon available electronically. Other less recent books [15, 74, 26] use the language Caml Light, which approximatively correspond to the core OCaml language, covering neither its module system, nor objects.

For other languages of the ML family, [58] is an excellent introductory document to Standard ML and [51, 50] are the reference documents for this language. For Haskell, the reference manual is [38] and [70, 30] give a very progressive approach to the language.

Typechecking and semantics of core ML are formalized in several articles and book Chapters. A concise and self-contained presentation can also be found in [43, 42, chapter 1]. A more modern formalization of the semantics, using small-step reductions, and type soundness can be found in [77]. Several introductory books to the formal semantics of programming languages [25, 52, 67] consider a subset of ML as an example. Last, [11] is an excellent introductory article to type systems in general.

The object and class layer of OCaml is formalized in [66]. A reference book on object calculi is [1]; this book, a little technical, formalizes the elementary mechanisms underlying object-oriented languages. Another integration of objects in a language of the ML family lead to the prototype language *Moby* described in [20]; a view mechanism for this language has been proposed in [21].

Several formalization of Standard ML and OCaml modules have been proposed. Some are based on calculi with unique names [51, 48], others use type theoretical concepts [27, 44]; both approaches are compared and related in [45, 69].

Beyond ML

ML is actually better characterized by its type system than by its set of features. Indeed, several generalizations of the ML type system have been proposed to increase its expressiveness while retaining its essential properties and, in particular, type inference.

Subtyping polymorphism, which is used both in popular languages such as Java and in some academic higher-order languages, has long been problematic in an ML context. (Indeed, both Java and higher-order languages share in common that types of abstractions are not inferred.) However, there has been proposals to add subtyping to ML while retaining type-inference. They are all based on a form of subtyping constraints [3, 19, 62] or, more generally, on typing constraints [54] and differ from one another mostly by their presentation. However, none of these works have yet been turned into a large scale real implementation. In particular, displaying types to the user is a problem that remains to be dealt with.

Other forms of polymorphism are called *ad hoc*, or *overloading* polymorphism by optimists. Overloading allows to bind several unrelated definitions to the same name in the same scope. Of course, then, for each occurrence of an overloaded name, one particular definition must be chosen. This selection process, which is called name resolution, can be done either at compile-time or at run-time, and overloading is called *static* or *dynamic*, accordingly. Name resolution for static overloading is done in combination with type-checking and is based on the type context in which the name is used. For example, static overloading is used in Standard ML for arithmetic operations and record accesses. Type information may still be used in the case of dynamic overloading, but to add, whenever necessary, run-time type information that will be used to guide the name resolution, dynamically. For example, type classes in Haskell [38] are a form of dynamic overloading where type information is carried by class dictionaries [55, 36], indirectly. Extensional polymorphism [17] is a more explicit form of dynamic overloading: it allows to pattern-match on the type of expressions at run-time. This resembles to, but also differ from dynamics values [47, 2].

The system $F^{\omega}_{<:}$, which features both higher-order types and subtyping, is quite expressive, hence very attractive as the core of a programming language. However, its type inference problem is not decidable [75]. A suggestion to retain its expressiveness and the convenience of implicit typing simultaneously is to provide only partial type reconstruction [60, 56]. Here, the programmer must write some, but not all, type information. The goal is of course that very little type information will actually be necessary to make type reconstruction decidable. However, the difficulty remains to find a simple specification of where and when annotations are mandatory, without requiring too many obvious or annoying annotations. An opposite direction to close the gap between ML and higher-order type systems is to embed higher-order types into ML types [24]. However, this raises difficulties that are similar to partial type reconstruction.

Actually, most extensions of ML explored so far seem to fit into two categories. Either, they reduce to insignificant technical changes to core ML, sometimes after clever reformulation though, or they seem to increase the complexity in disproportion with the gain in expressiveness. Thus, the ML type system might be a stable point of equilibrium —a best compromise between expressiveness and simplicity. This certainly contributed to its (relative) success. This also raised the standards for its its successor.

Appendix A

First steps in OCaml

Let us first check the “Hello world” program. Use the editor to create a file `hello.ml` containing the following single line:

```
print_string "Hello world!\n";;
```

Then, compile and execute the program as follows:

```
ocamlc −o hello hello.ml
./hello
```

```
Hello World
```

Alternatively, the same program could have been typed interactively, using the interpreter `ocaml` as a big desk calculator, as shown in the following session:

```
ocaml
```

```
        Objective Caml version 3.00

#
```

```
print_string "hello world!\n";;
```

```
hello world!
− : unit = ()
```

To end interactive sessions type `^D` (`Control D`) or call the `exit` function of type `int -> unit`:

```
exit 0;;
```

Note that the `exit` function would also terminate the execution in a compiled program. Its integer argument is the return code of the program (or of the interpreter).

Exercise 33 ((*) Unix commands true and false) *Write the Unix commands* `true` *et* `false` *that do nothing but return the codes* `0` *and* `1`*, respectively.* Answer ☐

The interpreter can also be used in batch mode, for running scripts. The name of the file containing the code to be interpreted is passed as argument on the command line of the interpreter:

```
ocaml hello.ml

Hello World
```

Note the difference between the previous command and the following one:

```
ocaml < hello.ml

        Objective Caml version 3.00

# Hello World
- : unit = ()
#
```

The latter is a "batch" interactive session where the input commands are taken from the file `hello.ml`, while the former is a script execution, where the commands of the file are evaluated in script mode, which turns off interactive messages.

Phrases of the core language are summarized in the table below:

– value definition	`let` $x = e$
– [mutually recursive] function definition[s]	`let` [`rec`] $f_1\ x_1\ ...\ = e_1\ ...$ [`and` $f_n\ x_n\ ...\ = e_n$]
– type definition[s]	`type` $q_1 = t_1 ...$ [`and` $q_n = t_n$]
– expression	e

Phrases (optionally) end with "`;;`".

```
(* That is a comment (* and this is a comment inside
   a comment *) continuing on several lines *)
```

Note that an opening comment paren "`(*`" will absorb everything as part of the comment until a well-balanced closing comment paren "`*)`" is found. Thus, if you inadvertently type the opening command, you may think that the interpreter is broken because it swallows all your input without ever sending any output but the prompt.

Use `^C` (Control-C) to interrupt the evaluation of the current phrase and return to the toplevel if you ever fall in this trap!

Typing `^C` can also be used to stop a never-ending computation. For instance, try the infinite loop

```
while true do () done;;
```

and observe that there is no answer. Then type `^C`. The input is taken into account immediately (with no trailing carriage return) and produces the following message:

```
^C
 Interrupted.
```

Expressions are

– local definition	`let` $x = e_1$ `in` e_2
(+ mutually recursive local function definitions)	
– anonymous function	`fun` x_1 ... x_n `->` e
– function call	$f\ x_1\ ...\ x_n$
– variable	x ($M.x$ if x is defined in M)
– constructed value	(e_1, e_2)
including constants	`1`, `'c'`, `"aa"`
– case analysis	`match` e `with` p_1 `->` $e_1 \ldots \mid p_n$ `->` e_n
– handling exceptions	`try` e `with` p_1 `->` $e_1 \ldots \mid p_n$ `->` e_n
– raising exceptions	`raise` e

– for loop	`for` $i = e_0$ [`down`]`to` e_f `do` e `done`
– while loop	`while` e_0 `do` e `done`
– conditional	`if` e_1 `then` e_2 `else` e_3
– sequence	$e; e'$
– parenthesis	(e) or `begin` e `end`

Remark that there is no notion of instruction or procedure, since all expressions must return a value. The unit value `()` of type `unit` conveys no information: it is the unique value of its type.

The expression e in `for` and `while` loops, and in sequences must be of type `unit` (otherwise, a warning message is printed).

Therefore, useless results must explicitly be thrown away. This can be achieved either by using the `ignore` primitive or an anonymous binding.

```
ignore;;
- : 'a -> unit = <fun>
ignore 1; 2;;
- : int = 2
let _ = 1 in 2;;
- : int = 2
```

(The anonymous variable `_` used in the last sentence could be replaced by any regular variable that does not appear in the body of the `let`)

Basic types, constants, and primitives are described in the following table.

Type	Constants	Operations
unit	`()`	no operation!
bool	`true   false`	`&&   ||   not`
char	`'a'   '\n'   '\097'`	`Char.code   Char.chr`
int	1 2 3	+ `-` `*` / `max_int`
float	1.0 2. 3.14 6e23	+. -. *. /. `cos`
string	`"a\tb\010c\n"`	^ `s.[i]   s.[i] <- c`

Polymorphic types and operations

arrays	`[| 0; 1; 2; 3 |]`	`t.(i)   t.(i) <- v`
pairs	(1, 2)	`fst   snd`
tuples	(1, 2, 3, 4)	Use pattern matching!

Infixes become prefixes when put between parentheses.

For instance, `( + )` x_1 x_2 is equivalent to x_1 `+` x_2. Here, it is good practice to leave a space between the operator and the parenthesis, so as not to fall in the usual trap: The expression “`(*)`” would not mean the product used as a prefix, but the unbalanced comment starting with the character “`)`” and waiting for its closing comment paren “`*)`” closing paren.

Array operations are polymorphic, but arrays are homogeneous:

```
[| 0; 1; 3 |];;
- : int array = [|0; 1; 3|]
[| true; false |];;
- : bool array = [|true; false|]
```

Array indices vary from 0 to $n - 1$ where n is the array size.

Array projections are *polymorphic*: they operate on any kind of array:

```
fun x -> x.(0);;
- : 'a array -> 'a = <fun>
fun t k x -> t.(k) <- x;;
- : 'a array -> int -> 'a -> unit = <fun>
```

Arrays must always be initialized:

```
Array.create;;
- : int -> 'a -> 'a array = <fun>
```

The type of the initial element becomes the type of the array.

Tuples are heterogeneous; however, their arity is fixed by their type: a pair `(1, 2)` of `int * int` and a triple `(1, 2, 3)` of type `int * int * int` are incompatible.

The projections are polymorphic but are defined only for a fixed arity. For instance, `fun (x, y, z) -> y` returns the second component of any triple. There is no particular syntax for projections, and pattern matching must be used. The only exceptions are `fst` and `snd` for pairs defined in the standard library.

Records In OCaml, records are analogous to variants and must be declared before being used. See for example the type `regular` used for cards (Exercise 2.1, page 53). Mutable fields of records must be declared as such at the definition of the record type they belong to.

```
type 'a annotation = { name : string; mutable info : 'a};;
 type 'a annotation = { name : string; mutable info : 'a; }
fun x -> x.info;;
 - : 'a annotation -> 'a = <fun>
let p = { name = "John"; info = 23 };;
 val p : int annotation = {name="John"; info=23}
p.info <- p.info + 1;;
 - : unit = ()
```

Command line Arguments passed on the command line are stored in the string array `Sys.argv`, the first argument being the name of the command.

Exercise 34 ((*) Unix command echo) *Implement the Unix echo function.* Answer □

The standard library `Arg` provides an interface to extract arguments from the command line.

Input-output A summary of primitives for manipulating channels and writing on them is given in the two tables below. See the core and standard libraries for an exhaustive list.

Predefined channels

```
stdin : in_channel
stdout : out_channel
stderr : out_channel
```

Creating channels

```
open_out : string -> out_channel
open_in : string -> in_channel
close_out : out_channel -> unit
```

Reading on stdin

`read_line` : `unit -> string`
`read_int` : `unit -> int`

Writing on stdout

`print_string` : `string -> unit`
`print_int` : `int -> unit`
`print_newline` : `unit -> unit`

Exercise 35 ((**) Unix cat and grep commands) *Implement the Unix* `cat` *command that takes a list of file names on the command line and print the contents of all files in order of appearance; if there is no file on the command line, it prints stdin.* Answer
The Unix `grep` *command is quite similar to* `cat` *but only list the lines matching some regular expression. Implement the command* `grep` *by a tiny small change to the program* `cat`*, thanks to the standard library* `Str`*.*
Answer □

Exercise 36 ((**) Unix wc command) *Implement the Unix* `wc` *command that takes a list of file names on the command line and for each file count characters, words, and lines; additionally, but only if there were more than one file, it presents a global summary for the union of all files.*
Answer □

Appendix B

Variant types and labeled arguments

In this appendix we briefly present two recent features of the OCaml language and illustrate their use in combination with classes. Actually, they jointly complement objects and classes in an interesting way: first, they provide a good alternative to multiple class constructors, which OCaml does not have; second, variant types are also a lighter-weight alternative to datatype definitions and are particularly appropriate to simulate simple typecases in OCaml. Note that the need for typecases is sufficiently rare, thanks to the expressiveness of OCaml object type-system, that an indirect solution to typecases is quite acceptable.

B.1 Variant types

Variants are tagged unions, like ML datatypes. Thus, they allow values of different types to be mixed together in a collection by tagging them with variant labels; the values may be retrieved from the collection by inspecting their tags using pattern matching.

However, unlike datatypes, variants can be used without a preceding type declaration. Furthermore, while a datatype constructor belong to a unique datatype, a variant constructor may belong to any (open) variant.

Quick overview Just like sum type constructors, variant tags must be capitalized, but they must also be prefixed by the back-quote character as follows:

```
let one = ‘Int 1 and half = ‘Float 0.5;;
```

```
val one : [> ‘Int of int] = ‘Int 1
val half : [> ‘Float of float] = ‘Float 0.5
```

Here, variable `one` is bound to a variant that is an integer value tagged with `‘Int`. The `>` sign in the type `[> ‘Int of int]` means that `one` can actually be a assigned a super type. That is, values of this type can actually have another tag. However, if they have have tag `‘Int` then they must carry integers. Thus, both `one` and `half` have compatible types and can be stored in the same collection:

```
let collection = [ one; half ];;
```

```
val collection : [> ‘Int of int | ‘Float of float] list =
  [‘ Int 1; ‘Float 0.5]
```

Now, the type of `collection` is a list of values, that can be integers tagged with `‘Int` or floating point values tagged with `‘Float`, or values with another tag.

Values of a heterogeneous collection can be retrieved by pattern matching and then reified to their true type:

```
let float  = function
  | ‘ Int x -> float_of_int x
  | ‘ Float x -> x;;
```

```
val float : [< ‘Int of int | ‘Float of float] -> float = <fun>
```

```
let total =
  List.fold_left (fun x y -> x +. float y) 0. collection ;;
```

Implementing typecase with variant types The language ML does not keep types at run time, hence there is no typecase construct to test the types of values at run time. The only solution available is to explicitly tag values with constructors. OCaml data types can be used for that purpose but variant types may be more convenient and more flexible here since their constructors do not have to be declared in advance, and their tagged values have all compatible types.

For instance, we consider one and two dimensional point classes and combine their objects together in a container.

```
class point1 x = object method getx = x + 0 end;;
let p1 = new point1 1;;
```

To make objects of the two classes compatible, we always tag them. However, we also keep the original object, so as to preserve direct access to the common interface.

```
let pp1 = p1, ‘Point1 p1;;
```

We provide testing and coercion functions for each class (these two functions could of also be merged):

```
exception Typecase;;
let is_point1 = function _, ‘Point1 q -> true | _ -> false;;
let to_point1 = function _, ‘Point1 q -> q | _ -> raise Typecase;;
```

as well as a safe (statically typed) coercion `point1`.

```
let as_point1 = function pq -> (pq :> point1 * _);;
```

Similarly, we define two-dimensional points and their auxiliary functions:

```
class point2 x y = object inherit point1 x method gety = y + 0 end;;
let p2 = new point2 2 2;;
let pp2  = (p2 :> point1), ‘Point2 p2;;
let is_point2 = function _, ‘Point2 q -> true | _ -> false;;
let to_point2 = function _, ‘Point2 q -> q | _ -> raise Typecase;;
let as_point2 = function pq -> (pq :> point2 * _);;
```

Finally, we check that objects of both classes can be collected together in a container.

```
let l =
  let ( @:: ) x y = (as_point1 x) :: y in
  pp1 @:: pp2 @ :: [];;
```

Components that are common to all members of the collection can be accessed directly (without membership testing) using the first projection.

```
let getx p = (fst p)#getx;;
List.map getx l;;
```

Conversely, other components must accessed selectively via the second projection and using membership and conversion functions:

```
let gety p =  if is_point2 p then (to_point2 p) # gety else 0;;
```

```
List.map gety l;;
```

B.2 Labeled arguments

In the core language, as in most languages, arguments are anonymous.

Labeled arguments are a convenient extension to the core language that allow to consistently label arguments in the declaration of functions and in their application. Labeled arguments increase safety, since argument labels are checked against their definitions. Moreover, labeled arguments also increase flexibility since they can be passed in a different order than the one of their definition. Finally, labeled arguments can be used solely for documentation purposes.

For instance, the erroneous exchange of two arguments of the same type —an error the typechecker would not catch— can be avoided by labeling the arguments with distinct labels. As an example, the module `StdLabels.String` provides a function `sub` with the following type:

```
StdLabels.String.sub;;
```

```
– : string -> pos:int -> len:int -> string = <fun>
```

This function expects three arguments: the first one is anonymous, the second and third ones are labeled `pos` and `len`, respectively. A call to this function can be written

```
String.sub "Hello" ~pos:0 ~len:4
```

or equivalently,

```
String.sub "Hello" ~len:4 ~pos:0
```

since labeled arguments can be passed to the function in a different order. Labels are (lexically) enclosed between `~` and `:`, so as to distinguish them from variables.

By default, standard library functions are not labeled. The module `StdLabels` redefines some modules of the standard library with labeled versions of some functions. Thus, one can include the command

```
open StdLabels;;
```

at the beginning of a file to benefit from labeled versions of the libraries. Then, `String.sub` could have been used as a short hand for

`StdLabels.String.sub` in the example above.

Labeled arguments of a function are declared by labeling the arguments accordingly in the function declaration. For example, the labeled version of `substring` could have been defined as

```
let substring s ~pos:x ~length:y = String.sub s x y;;
```

Additionally, there is a possible short-cut that allows us to use the name of the label for the name of the variable. Then, both the ending `:` mark at the end of the label and the variable are omitted. Hence, the following definition of `substring` is equivalent to the previous one.

```
let substring s ~pos ~length = String.sub s pos length;;
```

B.3 Optional arguments

Labels can also be used to declare default values for some arguments.

Quick overview Arguments with default values are called optional arguments, and can be omitted in function calls —the corresponding default values will be used. For instance, one could have declared a function `substring` as follows

```
let substring ?pos:(p=0) ~length:l s = String.sub s p l;;
```

This would allow to call `substring` with its `length` argument and an anonymous string, leaving the position to its default value 0. The anonymous string parameter has been moved as the last argument, inverting the convention taken in `String.sub`, so as to satisfy the requirement than an optional argument must always be followed by an anonymous argument which is used to mark the end optional arguments and replace missing arguments by their default values.

Application to class constructors In OCaml, objects are created from classes with the `new` construct. This amounts to having a unique constructor of the same name as the name of the class, with the same arity as that of the class.

In object-oriented languages, it is common and often quite useful to have several ways of building objects of the same class. One common example are is to have default values for some of the parameters. Another situation is to have two (or more) equivalent representations for an object, and to be able to initialized the object using the object either way. For instance, complex points can be defined by giving either cartesian or polar coordinates.

One could think of emulating several constructors by defining different variants of the class obtained by abstraction and application of the original class, each one providing a new class constructor. However, this schema breaks modularity, since classes cannot be simultaneously refined by inheritance.

Fortunately, labeled arguments and variant types can be used together to provide the required flexibility, as it there were several constructors, but with a unique class that can be inherited.

For example, two-dimensional points can be defined as follows:

```
class point ~x:x0 ?y:(y0=0) () =
  object method getx = x0 + 0 method gety = y0 + 0 end;;
```

(The extra unit argument is used to mark the end of optional arguments.) Then, the y coordinate may be left implicit, which defaults to 0.

```
let p1 = new point ~x:1 ();;
let p2 = new point ~x:1 ~y:2 ();;
```

Conversely, one could define the class so that

```
class point arg =
  let x0, y0 =
   match arg with
   | ` Cart (x,y) -> x, y
   | ` Polar(r,t) -> r *. cos t, r *. sin t in
  object method getx = x0 method gety = y0 end;;
```

Then, points can be build by either passing cartesian or polar coordinates

```
let p1 = new point (`Cart (1.414, 1.));;
let p2 = new point (`Polar (2., 0.52));;
```

In this case, one could also choose optional labels for convenience of notation, but at the price of some dynamic detection of ill-formed calls:

```
class point ?x ?y ?r ?t () =
  let x0, y0 =
    match x, y, r, t with
   | Some x, Some y, None, None -> x, y
   | None, None, Some r, Some t -> r *. cos t, r *. sin t
   | _, _, _, _ -> failwith ”Cart and Polar coordinates can't be mixed” in
  object method getx = x0 method gety = y0 end;;
let p1 = new point ~x:2. ~y:0.52 ();;
let p2 = new point ~r:1.414 ~t:0.52 ();;
```

Appendix C

Answers to exercises

Exercise 1, page 21

```
let rec fill_context = function
  | Top, e as ce -> e
  | AppL (c, l), e -> fill_context (c, App (e, l))
  | AppR ((_, e1), c), e2 -> fill_context (c, App (e1, e2))
  | LetL (x, c, e2), e1 -> fill_context (c, Let (x, e1, e2));;
```

Exercise 1 (continued)

```
exception Error of context * expr;;
exception Value of int;;
let rec decompose_down (c,e as ce) =
  match e with
  | Var _ -> raise (Error (c, e))
  | Const c when c.constr -> raise (Value (c.arity + 1))
  | Const c -> raise (Value (c.arity))
  | Fun (_, _) -> raise (Value 1)
  | Let (x, e1, e2) -> decompose_down (LetL (x, c, e2), e1)
  | App (e1, e2) as e ->
      try decompose_down (AppL (c, e2), e1) with Value k1 ->
        try decompose_down (AppR ((k1, e1), c), e2) with Value k2 ->
```

```
                if k1 > 1 then raise (Value (k1 −1)) else ce;;
```

Exercise 1 (continued)

```
let rec decompose_up k (c, v as cv) =
  if k > 0 then
    match c with
    | Top -> raise Not_found
    | LetL (x, c’, e) ->
       ( c’, (Let (x, v, e)))
    | AppR ((k’, v’), c’) ->
        decompose_up (k’−1) (c’, App (v’, v))
    | AppL (c’, e) ->
        try decompose_down (AppR ((k, v), c’), e)
        with Value _ -> decompose_up (k−1) (c’, App (v, e))
  else cv;;
```

Exercise 1 (continued)

We first attempt decomposing the term a further and if it is a value of arity k, then we decompose the context E up to k left-application levels.

```
let decompose ce =
  try decompose_down ce with Value k -> decompose_up k ce;;
```

Exercise 1 (continued)

```
let reduce_in ((c : context), e) = (c, top_reduction e);;
let eval_step ce = reduce_in (decompose ce);;
```

The evaluation if the iteration of the one-step reduction until the expression is a value.

```
let rec eval_all ce = try eval_all (eval_step ce) with Not_found -> ce;;
let eval e = fill_context (eval_all (Top, e));;
eval e;;
```

```
 − : expr = Const {name=Int 9; constr=true; arity=0}
```

Exercise 1 (continued)

We write a single function that enables to print a context alone, an expression alone, or a context and an expression to put in the hole. Actually, we print all of them as expressions but use a special constant as a hook to recognize when we reached the context.

```
let hole = Const {name = Name "[]"; arity = 0; constr = true};;
let rec expr_with expr_in_hole k out  =
  let expr = expr_with expr_in_hole in
  let string x = Format.fprintf out x in
  let paren p f =
    if k > p then string "("; f(); if k > p then string ")" in
  function
    | Var x -> string "%s" x
    | Const _ as c when c = hole ->
      string "[%a]" (expr_with hole 0) expr_in_hole
    | Const {name = Int n} -> string "%d" n
    | Const {name = Name c} -> string "%s" c
    | Fun (x,a) ->
      paren 0 (fun()-> string "fun %s -> %a" x (expr 0) a)
    | App (App (Const {name = Name ("+" | "*" as n)}, a1), a2) ->
      paren 1 (fun()->
        string "%a %s %a" (expr 2) a1 n (expr 2) a2)
    | App (a1, a2) ->
      paren 1 (fun()-> string "%a %a" (expr 1) a1 (expr 2) a2)
    | Let (x, a1, a2) ->
      paren 0 (fun()->
        string "let x = %a in %a" (expr 0) a1 (expr 0) a2);;

let print_context_expr (c, e) =
  expr_with e 0 Format.std_formatter (fill_context (c, hole))
let print_expr e = expr_with hole 0 Format.std_formatter e
let print_context c = print_expr (fill_context (c, hole));;
#install_printer print_context_expr;;
```

Exercise 3, page 43

```
let print_type t =
  let rec print k out t =
    let string x = Printf.fprintf out x in
    let paren p f =
      if k > p then string ”(”; f(); if k > p then string ”)” in
    let t = repr t in
    begin match desc t with
    | Tvar n -> string ”’a%d” n
    | Tcon (Tint, []) -> string ”int”
    | Tcon (Tarrow, [t1; t2]) ->
        paren 0 (fun() ->
          string ”%a -> %a” (print 1) t1 (print 0) t2)
    | Tcon (g, l) -> raise (Arity (t, t))
    end in
  acyclic t;
  print 0 stdout t;;
```

Exercise 4, page 44

We can either chose an function version:

```
let ftv_type t =
  let visited = marker() in
  let rec visit ftv t =
    let t = repr t in
    if t.mark = visited then ftv
    else
      begin
        t.mark <- visited;
        match desc t with
       | Tvar _ -> t::ftv
       | Tcon (g, l) -> List.fold_left visit ftv l
      end in
  visit [] t;;
```

or an imperative version:

```
let ftv_type t =
  let ftv = ref [] in
```

```
    let visited = marker() in
    let rec visit t =
      let t = repr t in
      if t.mark = visited then ()
      else
        begin
          t.mark <- visited;
          match desc t with
         | Tvar _ -> ftv := t::!ftv
         | Tcon (g, l) -> List.iter visit l
        end in
    visit t; !ftv;;
```

Exercise 4 (continued)

```
  let type_instance (q, t) =
    acyclic t;
    let copy t = let t = repr t in t, tvar() in
    let copied = List.map copy q in
    let rec visit t =
      let t = repr t in
      try List.assq t copied with Not_found ->
        begin match desc t with
       | Tvar _ | Tcon (_ , []) -> t
       | Tcon (g, l) -> texp (Tcon (g, List.map visit l))
        end in
    visit t;;
```

Exercise 4 (continued)

To keep sharing, every instance of a node will be kept in a table, and the mark of the old node will be equal (modulo a translation) to the index of the corresponding node in the table. Since we do not know at the beginning the size of the table. we write a library of extensible arrays.

```
  module Iarray =
    struct
      type 'a t = { mutable t : 'a array; v : 'a }
```

```
    let create k v = { t = Array.create (max k 3) v; v = v }
    let get a i =
      if Array.length a.t > i then a.t.(i) else a.v
    let set a i v =
      let n = Array.length a.t in
      if n > i then a.t.(i) <- v else
      begin
        let t = Array.create (2 * n) a.v in
        Array.blit a.t 0 t 0 n;
        a.t <- t;
        t.(i) <- v;
      end
  end;;
```

Now, we can define the instance function. The polymorphic variables are created first. Then, when we meet variables that have not been created, we know that they should be shared rather than duplicated.

```
let type_instance (q, t) =
  let table = Iarray.create 7 (tvar()) in
  let poly = marker() in
  let copy_var t =
      let t' = tvar() in let p = marker() in
      t.mark <- p; Iarray.set table (p - poly) t'; t' in
  let q' = List.map copy_var q in
  let rec visit t =
    let t = repr t in
    if t.mark > poly then Iarray.get table (t.mark -poly)
    else
      begin match desc t with
      | Tvar _ | Tcon (_ , []) -> t
      | Tcon (g, l) ->
          let t' = copy_var t in
          t'.texp <- Desc (Tcon (g, List.map visit l)); t'
      end in
  visit t;;
```

Exercise 5, page 46

The function `visit_type` visit all nodes of a type that are not marked to be `excluded` and has not yet been `visited` and applying a function `f` to each visited none.

```
let visit_type exclude visited f t = let rec visit t =
  let t = repr t in
  if t.mark = exclude || t.mark == visited then ()
  else
    begin
      t.mark <- visited; f t;
      match desc t with
      | Tvar _ -> ()
      | Tcon (g, l) -> List.iter visit l
    end in
visit t;;
```

The generalization mark variables that are free in the environment, then list variables in the type that are noted mark as free in the environment.

```
let generalizable tenv t0 =
  let inenv = marker() in
  let mark m t = (repr t).mark <- m in
  let visit_asumption (x, (q, t)) =
    let bound = marker() in
    List.iter (mark bound) q; visit_type bound inenv ignore t in
  List.iter visit_asumption tenv;
  let ftv = ref [] in
  let collect t = match desc t with Tvar _ -> ftv := t::!ftv | _ -> () in
  let free = marker() in
  visit_type inenv free collect t0;
  ! ftv;;
let x = tvar();;
generalizable [] (tarrow x x);;
```

Exercise 6, page 48

It suffices to remark there is a subterm $f\ f$ in a context where f is bound to a lambda. Hence both occurrences of f must have the same type, which is not possible since the type of the right occurrence should be the domain of the (arrow) type of the other one.

Exercise 7, page 48

```
let fact' fact x = if x = 0 then 1 else x * fact (x-1);;
let fact x = fix fact' x;;
```

Exercise 9, page 49

$$
\begin{array}{lrl}
\textbf{let rec } f_1' = & \lambda f_2.\lambda f_3.\lambda x. & \textbf{let } f_1 = f_1'\ f_2\ f_3 \textbf{ in} \\
 & & a_1 \\
\textbf{in} & & \\
\textbf{let rec } f_2' = & \lambda f_3.\lambda x. & \textbf{let } f_2 = f_2'\ f_3 \textbf{ in} \\
 & & \textbf{let } f_1 = f_1'\ f_2\ f_3 \textbf{ in} \\
 & & a_2 \\
\textbf{in} & & \\
\textbf{let rec } f_3' = & \lambda x. & \textbf{let } f_3 = f_3' \textbf{ in} \\
 & & \textbf{let } f_2 = f_2'\ f_3 \textbf{ in} \\
 & & \textbf{let } f_1 = f_1'\ f_2\ f_3 \textbf{ in} \\
 & & a_3 \\
\textbf{in} & & \\
a & &
\end{array}
$$

Exercise 10, page 50

Let us be lazy and use `ocaml -rectypes`:

```
let fix =
 (fun f' ->
     ( fun f -> (fun x -> f' (f f) x))
     ( fun f -> (fun x -> f' (f f) x))
  );;
 val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

```
let fact = fix fact' in fact 5;;
 - : int = 120
```

Exercise 10 (continued)

```
let fix f' = let g f x = f' (f f) x in g g;;
 val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
let fact = fix fact' in fact 5;;
 - : int = 120
```

Exercise 11, page 50

```
let print_type t =
  let cyclic = marker() in
  begin try acyclic t
  | with Cycle l -> List.iter (fun t -> (repr t).mark <- cyclic) l;
  end;
  let named = marker() in
  let rec print k out t =
    let string x = Printf.fprintf out x in
    let paren p f =
      if k > p then string "("; f(); if k > p then string ")" in
    let t = repr t in
    if t.mark > named then string "'a%d" (t.mark - named)
    else
      begin match desc t with
      | Tvar n ->
          t.mark <- marker(); string "'a%d" (t.mark - named)
      | Tcon (Tint, []) ->
          string "int"
      | Tcon (Tarrow, [t1; t2]) when t.mark = cyclic ->
          t.mark <- marker();
          string "(%a -> %a as 'a%d)"
           ( print 1) t1 (print 1) t2 (t.mark - named);
      | Tcon (Tarrow, [t1; t2]) ->
```

```
          paren 0 (fun() ->
            string "%a -> %a" (print 1) t1 (print 1) t2)
      | Tcon (g, l) -> raise (Arity (t, t))
      end in
  print 0 stdout t;;
```

Exercise 12, page 51

The only way the type-checker is by finding unbound variables, or by unification errors. The former cannot occur with closed terms. In turn, unification can failed either with occur check or with clashes when attempting to unify two terms with different top symbols. Recursive types removes occur-check. In the absence of primitive operations, types are either variables or arrow types. The only type constructor is →, so that there will never be any clash during unification.

Exercise 13, page 54

The only difficulty comes from the `Joker` card, which can be used in place of any other card. As a result, we cannot use a notion of binary equivalence of two cards, which would not be transitive. Instead, we define an asymmetric relation `agrees_with`: for instance, the card `Joker` agrees with `King`, but not the converse. For convenience, we also define the relation `disagree_with`, which is the negation of the symmetric relation `agrees_with`.

```
let agrees_with x y =
  match x, y with
    Card u, Card v -> u.name = v.name
  | _, Joker -> true
  | Joker, Card _ -> false
let disagrees_with x y = not (agrees_with y x);;
```

We actually provide a more general solution `find_similar` that searches sets of `k` similar cards among a `hand`. This function is defined by induction. If the `hand` is empty, there is no solution. If the first element of the `hand` is a `Joker`, we search for sets of `k-1` similar elements in the rest of the `hand` and add the `Joker` in front of each set. Otherwise, the first element of `hand` is a regular card `h`: we first search for the set of all elements matching the card `h` in the rest of the hand; this set constitutes a solution if its size if at least `k`; then, we add all other solutions among the rest of the hand that disagree with `h`. Otherwise, the solutions are only those that disagree with `h`.

```
let rec find_similar k hand =
  match hand with
   | []  -> []
   | Joker :: t ->
       List.map (fun p -> Joker::p) (find_similar (k − 1) t)
   | h :: t ->
       let similar_to_h = h :: List.find_all (agrees_with h) t in
       let others =
         find_similar k (List.find_all (disagrees_with h) t) in
       if List.length similar_to_h >= k then similar_to_h :: others
       else others;;
let find_carre = find_similar 4;;
```

Here is an example of search:

```
find_carre
 [ king Spade; Joker; king Diamond; Joker; king Heart;
   king Club; card Queen Spade; card Queen Club; club_jack ];;
```

```
− : card list list =
[[Card {suit=Spade; name=King}; Joker; Card {suit=Diamond; name=King};
  Joker; Card {suit=Heart; name=King}; Card {suit=Club; name=King}];
 [Joker; Joker; Card {suit=Spade; name=Queen};
  Card {suit=Club; name=Queen}]]
```

Exercise 15, page 56

It would still be correct if τ_i contains a variable that does not belong to $\overline{\alpha}$, since this variable would not be generalized in the types of constructors and destructors. (Of course, it would be unsafe to generalize such a variable: for instance, one could then define **type** $g = C^g$ `of` α with $C^g \colon \forall\alpha.\, \alpha \rightarrow g$ and $f : \forall\alpha, \beta.\, g \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$ and assign any type to the expression $e \overset{\triangle}{=}$ `match` $C^g\ 1$ `with` $C^g\ y \Rightarrow y \equiv f\ (C^g\ 1)\ (\lambda y.y)$, which reduces to the integer 1.)

Conversely, it is safe, although strange and useless, that $\overline{\alpha}$ contains superfluous variables. Consider for instance the definition **type** $g(\alpha) = C^g$ `of int`. Then $g 1$ would have type $g(\alpha)$ for any type α.

Note that the latter is allowed in OCaml, while the former is rejected.

Exercise 16, page 56

We may use the following type definition: **type** `bool` $=$ `True of unit` $\mid$ `False of unit` and see the expression `if` a `then` a_1 `else` a_2 as syntactic sugar for $\texttt{match}_{\texttt{bool}}$ a `with True` $x \Rightarrow a_1 \mid$ `False` $x \Rightarrow a_2$.

Exercise 17, page 56

The generalization is to allow constructors of any arity.

$$\texttt{type } g(\overline{\tau}) = C^g \texttt{ of } \overline{\tau}_i \mid \ldots C^g \texttt{ of } \overline{\tau}_n$$

This is rather easy and left as an exercise. Then, one could define:

$$\begin{array}{l}
\textbf{type } (\alpha_1, \alpha_2)\ (\_ * \_) = (\_, \_) \texttt{ of } (\alpha_1,\ \alpha_2) \\
\texttt{fst} \triangleq \lambda z.(F_* \ z \ (\lambda x.\lambda y.x)) \\
\texttt{snd} \triangleq \lambda z.(F_* \ z \ (\lambda x.\lambda y.y))
\end{array}$$

Exercise 18, page 57

```
type value = Value of (value -> value);;
let fold f = Value f
let unfold (Value f) = f;;
```

Exercise 18 (continued)

$$\begin{array}{rl}
\llbracket x \rrbracket & = x \\
\llbracket \lambda x.a \rrbracket & = \texttt{fold}\ (\lambda x.\llbracket a \rrbracket) \\
\llbracket a_1\ a_2 \rrbracket & = \texttt{unfold}\ (\llbracket a_1 \rrbracket\ \llbracket a_2 \rrbracket)
\end{array}$$

Exercise 18 (continued)

Here, let us use the compiler! For sake of readability, we abbreviate `fold` and `unfold`.

```
let ( ! ) f = fold f and ( @ ) a1 a2 = unfold a1 a2;;
let fix =
 !(fun f' ->
    !( fun f -> !(fun x -> f' @ (f @ f) @ x))
     @ !(fun f -> !(fun x -> f' @ (f @ f) @ x))
  );;
```

Exercise 19, page 59

$$\begin{aligned}
&\textbf{type } h'_1(\overline{\alpha}, \alpha_2) = \tau_1[\alpha_2/h_2(\overline{\alpha})]\\
&\textbf{type } h_2(\overline{\alpha}) = \tau_2[h'_1(\overline{\alpha}, h_2(\overline{\alpha}))/h_1(\overline{\alpha})]\\
&\textbf{type } h_1(\overline{\alpha}) = h'_1(\overline{\alpha}, h_2(\overline{\alpha}))
\end{aligned}$$

Exercise 20, page 64

The function `fix` should take as argument a function `f'` and return a function `f` so that `f x` is equal to `f' f x`. The solution is to store `f` in a reference `r`. We temporarily create the reference `r` with a dummy function (that is not meant to be used). Assuming that the reference will later contain the correct version of `f`, we can define `f` as `fun x -> f' !r x`. Hence, the following solution:

```
let fix f' =
  let r = ref (fun _ -> raise (Failure "fix")) in
  let f x = f' !r x in
  r := f; !r;;
```

```
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

Note that the exception should never be raised because the content of the reference is overridden by `f` before being reference is used.

We could also use the option type to initialized the content of the reference to `None` and replace it later with `Some f`. However, would not avoid raising an exception if the value of the reference where to be

`None` when being read, even though this situation should never occur dynamically in a correct implementation.

As an example, we define the factorial function:

```
let fact' fact n = if n > 0 then n * fact (n−1) else 1 in
fix fact' 6;;
```

```
− : int = 720
```

Exercise 22, page 66

The answer is positive. The reason is that exceptions hide the types of values that they communicate, which may be recursive types.

We first to define two inverse functions `fold` and `unfold`, using the following exception to mask types of values:

```
exception Hide of ((unit -> unit) -> (unit -> unit));;
```

```
let fold f = fun (x : unit) -> (raise (Hide f); ())
let unfold f = try (f(): unit); fun x -> x with Hide y -> y;;
```

```
val fold : ((unit -> unit) -> unit -> unit) -> unit -> unit = <fun>
val unfold : (unit -> unit) -> (unit -> unit) -> unit -> unit = <fun>
```

The two functions `fold` and `unfold` are inverse coercions between type $U \to U$ and U where U is $\texttt{unit} \to \texttt{unit}$: They can be used to embed any term of the untyped lambda calculus into a well-typed term, using the following well-known encoding:

$$\begin{aligned} [\![x]\!] &= x \\ [\![\lambda x.a]\!] &= \texttt{fold}\ (\lambda x.[\![a]\!]) \\ [\![a_1\ a_2]\!] &= \texttt{unfold}\ ([\![a_1]\!]\ [\![a_2]\!]) \end{aligned}$$

In particular, $[\![\texttt{fix}]\!]$ is well-typed.

Exercise 23, page 77

They differ when being inherited:

```
class cm1 = object inherit c1 method m = () end
class cm2 = object inherit c2 method m = () end;;
```

The method `c` of class `cm2` returns an object of type `c2` instead of `cm2`, as checked below:

```
((new cm1)#c : cm1);;
((new cm2)#c : cm2);;
```

Also, while the types `c1` and `c2` are equal, the type `cm1` is only a subtype of `cm2`.

Exercise 24, page 77

```
class backup =
  object (self)
    val mutable backup = None
    method save = backup <- Some (Oo.copy self)
    method restore =
      match backup with None -> self | Some x -> x
  end;;
```

Exercise 26, page 79

The only problem is the method `concat` that is a pseudo-binary method. There are two possible solutions. The first is not to make it a binary method, and let the class be parametric:

```
class ['a] ostring s = object (self)
  val s = s
  method repr = s
  method concat (t:'a) = {< s = s ^ t # repr >}
end;;
```

The second, more natural solution is to make concat a binary method by making the parameter be the self-type.

```
class ostring s = object (self : 'a)
  val s = s
  method repr = s
  method concat (t:'a) = {< s = s ^ t # repr >}
end;;
```

Exercise 28, page 84

For sake of readability, we only describe simplified rules covering all cases.

$$\frac{\langle p : \alpha_p; q : \tau_q; 1\rangle \doteq \langle p : \tau_p; r : \tau_r; 1\rangle \doteq e}{\langle p : \alpha_p; q : \tau_q; r : \tau_r; 1\rangle \doteq e \wedge \alpha_p \doteq \tau_p} \rightsquigarrow$$

$$\frac{\langle p : \tau_p; 1\rangle \doteq \langle p : \alpha_p; q : \tau_q; 0\rangle \doteq e}{\langle p : \tau_p; q : \tau_q; 0\rangle \doteq e \wedge \alpha_p \doteq \tau_p} \rightsquigarrow \qquad \frac{\langle p : \tau_p; 0\rangle \doteq \langle p : \alpha_p; 0\rangle \doteq e}{\langle p : \alpha_p; 0\rangle \doteq e \wedge \alpha_p \doteq \tau_p} \rightsquigarrow$$

$$\frac{\langle q : \tau_q; 0\rangle \doteq \langle r : \tau_r; 0\rangle \doteq e}{\perp} \rightsquigarrow$$

The generalization is obvious: the occurrences of $p : \alpha_p$, $p : \tau_p$, $q : \tau_q$, and $r : \tau_r$, can be replaced by finite mappings from labels to types P, P', Q, and R of disjoint domain except for P and P' of identical domain.

These (generalized) rules should be added to those for simple types, where rule Fail and Decompose are not extended to the object type constructor, nor the row constructors.

Exercise 33, page 128

```
exit 0;;
```

```
exit 1;;
```

Exercise 34, page 132

```
for i = 1 to Array.length Sys.argv - 1
do print_string Sys.argv.(i); print_char ' ' done;
print_newline();;
```

Exercise 35, page 133

```
let echo chan =
  try while true do print_endline (input_line chan) done
  with End_of_file -> ();;

if Array.length Sys.argv <= 1 then echo stdin
else
  for i = 1 to Array.length Sys.argv - 1
  do
    let chan = open_in Sys.argv.(i) in
    echo chan;
    close_in chan
  done;;
```

Exercise 35 (continued)

```
let pattern =
  if Array.length Sys.argv < 2 then
    begin
      print_endline "Usage: grep REGEXP file1 .. file2";
      exit 1
    end
  else
    Str.regexp Sys.argv.(1);;

let process_line l =
  try let _ = Str.search_forward pattern l 0 in print_endline l
  with Not_found -> ()

let process_chan c =
  try while true do process_line (input_line c) done
  with End_of_file -> ();;

let process_file f =
  let c = open_in f in process_chan c; close_in c;;

let () =
  if Array.length Sys.argv > 2 then
```

```
    for i = 2 to Array.length Sys.argv - 1
    do process_file  Sys.argv.(i) done
  else
    process_chan stdin;;
```

Exercise 36, page 133

```
type count = {
  mutable chars : int;
  mutable lines : int;
  mutable words : int;
};;

let new_count() = {chars = 0; lines = 0; words = 0};;
let total = new_count();;

let cumulate wc =
  total.chars <- total.chars + wc.chars;
  total.lines <- total.lines + wc.lines;
  total.words <- total.words + wc.words;;

let rec counter ic iw wc =
  let c = input_char ic in
  wc.chars <- wc.chars + 1;
  match c with
  | ' ' | '\t' ->
      if iw then wc.words <- wc.words + 1 else ();
      counter ic false wc
  | '\n' ->
      wc.lines <- wc.lines + 1;
      if iw then wc.words <- wc.words + 1 else ();
      counter ic false wc
  | c ->
      counter ic true wc;;

let count_channel ic wc =
  try counter ic false wc with
```

```
  | End_of_file -> cumulate wc; close_in ic;;

let output_results s wc =
  Printf.printf ”%7d%8d%8d %s\n” wc.lines  wc.words wc.chars  s;;

let count_file file_name =
  try
    let ic = open_in file_name in
    let wc = new_count() in
    count_channel ic wc;
    output_results file_name wc;
  with Sys_error s -> print_string s; print_newline(); exit 2;;

let main () =
  let nb_files = Array.length Sys.argv − 1 in
  if nb_files > 0 then
    begin
      for i = 1 to nb_files do
        count_file Sys.argv.(i)
      done;
      if nb_files > 1 then output_results ”total” total;
    end
  else
    begin
      let wc = new_count() in
      count_channel stdin wc;
      output_results ”” wc;
    end;
  exit 0;;

main();;
```

Bibliography

[1] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer, 1997.

[2] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[3] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM press, 1993.

[4] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, August 1998.

[5] Davide Ancona and Elena Zucca. A primitive calculus for module systems. In Gopalan Nadathur, editor, *PPDP'99 - International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 62–79. Springer-Verlag, 1999.

[6] Hans P. Barendregt. *The Lambda-Calulus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North-Holland, 1984.

[7] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. Modules, objets et calcul formel. In *Actes des Journées Francophones des Langages Applicatifs*. INRIA, 1999.

[8] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pages 302–315, July 1997.

[9] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[10] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, Brussels, July 1998.

[11] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing surveys*, 17(4):471–522, 1985.

[12] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[13] Craig Chambers. The Cecil Language: Specification & Rationale. Technical Report 93-03-05, University of Washington, 1993.

[14] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *proceedings of the conference Lisp and Functional Programming, LFP'86*. ACM Press, August 1986. Also appears as INRIA Research Report RR-529, May 1986.

[15] Guy Cousineau and Michel Mauny. *Approche fonctionnelle de la programmation*. Ediscience, 1995.

[16] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[17] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Proceedings of the 22th ACM Conference on Principles of Programming Languages*, January 1995.

[18] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *International Conference on Functional Programming 96*, pages 262–273. ACM Press, 1996.

[19] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, 1995.

[20] Kathleen Fisher and John Reppy. The design of a class mechanism for Moby. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Languages, design and Implementations*, pages 37–49, Atlanta, May 1999. ACM SIGPLAN, acm press.

[21] Kathleen Fisher and John Reppy. Extending Moby with inheritance-based subtyping. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, 2000.

[22] Alexandre Frey and François Bourdoncle. The Jazz home page. Free software available at `http://www.cma.ensmp.fr/jazz/index.html`.

[23] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, September 1998.

[24] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. In *International Symposium on Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 20–46. Springer, September 1997.

[25] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[26] Thérèse Accart Hardin and Véronique Donzeau-Gouge Viguié. *Concepts et outils de programmation — Le style fonctionnel, le style impératif avec CAML et Ada.* Interéditions, 1992.

[27] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *ACM Symposium on Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[28] F. Henglein. *Polymorphic Type Inference and Semi-Unification.* PhD thesis, Courant Institute of Mathematical Sciences, New York University., 1989.

[29] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus.* Volume 1 of London Mathematical Society Student texts. Cambridge University Press, 1986.

[30] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press, 2000.

[31] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse de doctorat d'état, Université Paris 7, 1976.

[32] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[33] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[34] Trevor Jim. *Principal typings and type inference.* PhD thesis, Massachusetts Institute of Technology, 1996.

[35] Trevor Jim. What are principal typings and what are they good for? In Principles of Programming Languages, pages 42–53, 1996.

[36] Mark P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, November 1994.

[37] Mark P. Jones and Simon Peyton Jones. Lightweight extensible records for haskell. In *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical report, 1999.

[38] Simon Peyton Jones and John Hughes. Report on the programming language Haskell 98. Technical report, `http://www.haskell.org`, 1999.

[39] Gilles Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Science*, pages 22–39, 1987.

[40] Claude Kirchner and Jean-Pierre Jouannaud. Solving equations in abstract algebras: a rule-based survey of unification. Research Report 561, Université de Paris Sud, Orsay, France, April 1990.

[41] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

[42] Xavier Leroy. Polymorphic typing of an algorithmic language. Research report 1778, INRIA, 1992.

[43] Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. Thèse de doctorat, Université Paris 7, 1992.

[44] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *ACM Symposium on Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[45] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[46] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[47] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[48] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In D. Sannella, editor, *Programming languages and systems – ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

[49] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[50] Robin Milner and Mads Tofte. *Commentary on Standard ML.* The MIT Press, 1991.

[51] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised).* The MIT Press, 1997.

[52] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[53] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23th ACM Conference on Principles of Programming Languages*, pages 54–67, January 1996.

[54] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.

[55] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–146, June 1995.

[56] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages*, 2001.

[57] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1996.

[58] Lawrence C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[59] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). Summary in *ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico.*

[60] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM Conference on Principles of Programming Languages*, 1998. Full version available as Indiana University CSCI Technical Report 493.

[61] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[62] François Pottier. Simplifying subtyping constraints: a theory. To appear in *Information & Computation*, August 2000.

[63] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.

[64] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, number 789 in Lecture Notes in Computer Science, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

[65] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1994.

[66] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the

proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[67] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[68] Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999.

[69] Claudio V. Russo. *Types for modules*. PhD thesis, University of Edinburgh, 1998.

[70] Simon Thompson. *Haskell: the craft of functional programming*. Addison-Wesley, 1999.

[71] Jérôme Vouillon. Combining subsumption and binary methods: An object calculus with views. In *ACM Symposium on Principles of Programming Languages*. ACM Press, 2000.

[72] Jérôme Vouillon. *Conception et réalisation d'une extension du langage ML avec des objets*. Thèe de doctorat, Université Paris 7, October 2000.

[73] Mitchell Wand. Complete type inference for simple objects. In D. Gries, editor, *Second Symposium on Logic In Computer Science*, pages 207–276, Ithaca, New York, June 1987. IEEE Computer Society Press.

[74] Pierre Weis and Xavier Leroy. *Le langage Caml*. Dunod, 1999.

[75] Joe B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

[76] Joe B. Wells. The essence of principal typings. In *Proceedings of the 29th International Colloquim on Automata, Languages, and Programming*, LNCS. Springer-Verlag, 2002.

[77] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

List of all exercises

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Appendix A

Index