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 \( x \). \( a \), or applications \( a_1 \). \( a_2 \). In addition, core ML has a distinguished construction `let \( x = a_1 \) 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 `;;`. All programs and the output of the interpreter (when useful) are displayed in a pink background. Input is marked with a vertical bar on the left margin, usually in green, except for incorrect programs that that uses a red mark. Here is an example. 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.

```ocaml
core.ml

```-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>

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 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:

```ocaml
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:

```ocaml
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!

```ocaml
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:

```ocaml
let derivative dx f x = function x -> (f(x +. dx) -. f(x)) /. dx;;
val derivative : float -> (float -> float) -> float -> float = <fun>

Then, the third derivative 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.

```ocaml
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
\[ \text{let } x_1 = a_1 ;; \text{ let } x_2 = a_2 ;; a \] into expressions of the form \[ \text{let } x_1 = a_1 \text{ in let } x_2 = a_2 \text{ in } a. \] 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 \( c \in C \) that are partitioned into constructors \( C \in C^+ \) and primitives \( f \in C \). Constants also come with an arity, that is, we assume a mapping \( \text{arity} \) from \( C \) to \( \mathbb{IN} \). For instance, integers and booleans are constructors of arity 0, pair is a constructor of arity 2, arithmetic operations, such as + or \( \times \) 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 ::= x \mid x. a \mid a a \quad \text{\(-calculus\)} \quad c \mid \text{let } x = a \text{ in } a \quad c ::= C \quad \begin{array}{c} \text{constructors} \\
\text{primitives}
\end{array}
\]

### 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`:

```ocaml
type name = Name of string | Int of int ;
type constant = { name : name ; constr : bool ; arity : int } ;
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.

```ocaml
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.

```ocaml
let e =
  App {Fun (\"x\", App (App (times, Var "x"), Var "x")),
       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 \( (x. x \* x) ((x. x+1) 2) \) 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:

\[
\begin{align*}
  v &::= \quad x, a \mid c^n v_1 \ldots v_n \quad \mid c^k v_1 \ldots v_k \quad k < n \\
  & \quad \text{Constructed values} \quad \text{Partially applied constants}
\end{align*}
\]

In fact, a partially applied constant \( c^n v_1 \ldots v_k \) behaves as the function \( x_{k+1} \ldots x_n \mapsto c^k v_1 \ldots v_k x_{k+1} \ldots x_n \), 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 \( \text{evaluated} \) defined on expressions:

\[
\begin{align*}
  \text{let rec evaluated = function} \\
  &\quad \text{Fun (_, _)} \to \text{true} \\
  &\quad u \to \text{partial_application 0 u} \\
  \text{and partial_application n = function} \\
  &\quad \text{Const c} \to (c.\text{constr} \mid c.\text{arity} > n) \\
  &\quad \text{App (u, v)} \to (\text{evaluated v} \&\& \text{partial_application (n+1) u})
\end{align*}
\]
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:

\[(x, a) v \quad a [x \ v] \quad (\_v)\]

\[\text{let } x = v \text{ in } a \quad a [x \ v] \quad (\text{Let}_v)\]

\[f^r v_1 \ldots v_n \quad a \quad (f^r v_1 \ldots v_n, a) \quad f\]

Redexes of the latter form, which describe how to reduce primitives, are also called delta rules. We write \(f \subset (\_\_\) for the union \(f \subset \bigcup (\_\_\) \_\). For instance, the rule (\( + \)) is the relation \(\{(p + q, p+q) \mid p, q \text{ IN}\}\) where \(n\) 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 \_\_\-rules can be implemented straightforwardly.

```ocaml
exception Reduce;;
let delta_bin_arith op code = function
  | App (App (Const { name = Name _, arity = 2} as c, _ v) ->
    if c = op ->
    | _ -> raise Reduce;;
let delta_plus = delta_bin_arith ( + );;
let delta_times = delta_bin_arith ( * );;
let delta_rules = [ delta_plus ; delta_times ];;

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

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

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

To implement (\( \_v \)), 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 \_\_\-conversion to avoid variable capture.

```ocaml
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', subst x v a'')
    else Let (y, subst x v a', subst x v a'')
  | Const c -> Const c;;
```

Then beta is straightforward:

```ocaml
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_reduce = 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, $x\cdot x$ 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:

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

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:

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

The evaluation relation is then the transitive closure of the small step reduction. 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, ie. to represent them as evaluation contexts of the host language, using its run-time stack. Typically, an evaluator would be defined as follows:

```ocaml
let eval_top_reduce a = try eval (top_reduce a) with Reduce -> a;;
let rec eval = 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;;
```

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 is 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.

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

Here, contexts are still implicit, and the redex is immediately filled back into the evaluation context. However, the `one_step` function can easily be decomposed into three operations: `find_redex` that returns an evaluation context and a term, the reduction per say, and the reconstruction of the result by fill the reducts back into the evaluation context. The simplest representation of contexts is to view them as functions form terms to terms as follows:

```ocaml
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.

```ocaml
let rec find_redex : expr -> context * expr =
  function
  | App (a1, a2) when not (evaluated a1) ->
    appL a2 ** find_redex a1
  | App (a1, a2) when not (evaluated a2) ->
    appR a1 ** find_redex a2
  | Let (x, a1, a2) when not (evaluated a1) ->
    letL x a2 ** find_redex 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.

```ocaml
let one_step a =
  let c, t = find_redex a in
c (top_reduce t);;
```

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

```ocaml
let rec eval a =
  try eval (one_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 $v$).

```ocaml
let $x = v$ in $a$ ( x. a) $v$
```

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

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

```
( x. (x * x)) (( x. (x + 1)) 2)

( x. (x * x)) (2 + 1)

( x. (x * x)) 3

(3 * 3)

9
```

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

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

**Exercise 1** [(**) Representing evaluation contexts] Above, evaluation context were left implicit, using the runtime stack, or functions from terms to terms. This exercise proposes an explicit representation of evaluation contexts as a data structure, so that can be explored by pattern matching.

However, contexts are represented upside-down so that they can be held by their hole so as to provide an efficient data structure for exploring and transforming terms. This follows 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.)

From the formal definition

```
E ::= [·] | E a | v E | let x = E in a
```

we deduce the following OCaml definition:

```ocaml
type context =
  Top
```
Values are a subset of expressions. However, the invariant cannot be captured directly by the type system. For simplicity, we identify values with expressions. A more secure implementation could use another type to represent values.

Note that the type context is linear, in the sense that constructors have at more one context subterm. This leads to two opposition representation of contexts. The naive representation of context let \( x = [\cdot] a_2 \) in \( a_3 \) is LetL \((x, \text{AppL (Top, } a_2), a_3\)\). However, we shall represent them upside-down by the term AppL (LetL (x, Top, a_3), a_2), 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 \( E [a] \) where \( E \) is in many ways if we do not impose that \( a \) is a redex. For instance, taking \( (a_1 a_2) a_3 \), allows the following decompositions

\[
[\cdot] \ [\text{let } x = a_1 a_2 \ \text{in } a_3]
\]

\[
(\text{let } x = [\cdot] \ \text{in } a_3) \ [a_1 a_2]
\]

\[
(\text{let } x = [\cdot] a_2 \ \text{in } a_3) \ [a_1]
\]

\[
(\text{let } x = a_1 [\cdot] \ \text{in } a_3) \ [a_2]
\]

(The last decomposition is only allowed if \( a_1 \) is a value.) These decompositions are them described by the following OCaml expressions, composed of a pair whose left-hand side is the context, and whose right-hand side is the term to be placed in the hole:

\[
\text{Hole, } \quad \text{LetL (x, Hole, a_3), App (a_1, a_2)}
\]

\[
\text{AppL (LetL (Hole, a_2), a_3), a_1}
\]

\[
\text{AppR (LetL (Hole, a_1), a_3), a_2}
\]

We can represent these term 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 \( \text{fill} \) of type \( \text{context} * \text{expr} \rightarrow \text{expr} \) that takes a pair \((E,a)\) of an evaluation context and an expression and returns the expression \( E[a] \) obtained by filling the context with the expression.

Define a function \( \text{down} \) of type \( \text{context} * \text{expr} \rightarrow \text{context} * \text{expr} \) that given a pair \((E,a)\) searches for a sub-context \( E' \) of \( E \) in evaluation position (and the residual term \( a' \) at that position); it raises an exception \( \text{Value} \) if \( a \) is already a value (actually for convenience, the exception could carry the functional degree of this value, i.e. the number of arguments this value can be applied to while still remaining a value).

\[\text{Answer}\]
Starting with \((\text{Top}, a)\) allows to find the first position \((E_0, a_0)\) where reduction may occur, transforming \(a_0\) into \(a_0'\). However, after reduction, one wish to find the next evaluation position \((E_n, a_n)\) given \((E_{n-1}, a_{n-1}')\) and knowing that \(E_{n-1}\) was an evaluation context.

Define a function \(\text{next}\) that behaves as \(\text{down}\), except that \(E'\), may not be a sub-context of \(E\), but also obtained by re-folding \((E,a)\) as much as necessary.

Finally, define the one-step reduction, and check the evaluation steps of the program \(e\) given above and recover the function \(\text{reduce}\) of type \(\text{expr} \rightarrow \text{expr}\) that reduces an expression to a value.

Write a pretty printer for expressions and contexts, and use it to trace evaluation steps, automatically.

### 1.3.2 Properties of the reduction

The strategy we gave is \(\text{call-by-value}\): the rule \(\text{(v)}\) only applies when the argument of the application has been reduced to value. Another simple reduction strategy is \(\text{call-by-name}\). Here, applications are reduced before the arguments. To obtain a \(\text{call-by-name}\) strategy, rules \(\text{(v)}\) and \(\text{(Let v)}\) 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).

\[
\begin{align*}
(\ x, a) a' & \rightarrow a [x \ a'] \quad \text{(n)} \\
\text{let x = a' in a} & \rightarrow a [x \ a'] \quad \text{(Let n)}
\end{align*}
\]

Simultaneously, we must restrict evaluation contexts to prevent reductions of the arguments before the reduction of the application itself; actually, it suffices to remove \(\nu E\) and \(\text{let x = E in a}\) 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 \(-\text{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:}

\[
E_n ::= \cdots | (f^k v_1 \cdots v_{k-1} E_k a_{k+1} \cdots a_n)
\]

However, in a \(\text{call-by-name}\) semantics, one may wish to have constants such as \(\text{fst}\) 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:

\[
\begin{align*}
(\ x, x + x) ((\ x, (x + 1)) 2) & \rightarrow (\ n) \\
(((\ x, (x + 1)) 2) * ((\ x, (x + 1)) 2)) & \rightarrow (\ n) \\
(2 + 1) * ((\ x, (x + 1)) 2) & \rightarrow (\ n) \\
3 * ((\ x, (x + 1)) 2) & \rightarrow (\ +) \\
3 * (2 + 1) & \rightarrow (\ n) \\
3 * 3 & \rightarrow (\ +) \\
9 & \rightarrow (\ *)
\end{align*}
\]

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 \(\text{call-by-need}\) or \(\text{lazy}\) 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 \( a^* \), then there is a unique sequence \( a = a_0 \ldots a_1 \ldots a_n = a^* \). 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:

\[
E ::= [\cdot] \mid E a \mid a E \text{ even if } a \text{ is not reduced} \mid \text{let } x = E \text{ in } a
\]

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:

\[
\begin{align*}
\text{normal evaluation} & \quad a_n \quad v \\
\text{run-time error} & \quad a \ldots a_n \quad a_n \quad v \\
\text{loop} & \quad a_n \quad \ldots
\end{align*}
\]

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 \((x, y)\) 1 does not reduce to a value.

```ocaml
val e : 'e
val x : 'a
val y : 'b

val eval : 'a -> 'e

evaluated (eval e);;
val (x, y) : ('a, 'b) * int
val (app, fun, var) : 'a -> 'b -> int

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 \( \vdash a \; r \) 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 \( x. \; a, \; e \) of a function and an environment (in which the function should be executed). Finally, answers are values or plus a distinguished answer \( \text{error} \).

\[
\begin{align*}
\mathcal{V} & \equiv \emptyset \mid x \mid v \\
v & \equiv x \cdot a, \mid \mathcal{C}^{v_1 \ldots v_k} \mid \mathcal{C} \quad \text{constructed values} \\
r & \equiv v \mid \text{error}
\end{align*}
\]

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

An inference rule written

\[
P_1 \ldots P_n \quad \Rightarrow C
\]

is composed of premises \( P_1, \ldots, P_n \) and a conclusion \( C \) and should be read as the implication: \( P_1 \ldots P_n \Rightarrow C \); 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.

**Figure 1.1: Big step reduction rules for Core ML**

- **Eval-Const**
  \( \vdash a \; v \quad \vdash C^1 \; a \; C^1 \; v \)

- **Eval-Const-Error**
  \( \vdash a \; \text{error} \quad \vdash c \; a \; c \; \text{error} \)

- **Eval-Prim**
  \( \vdash a \; v \; f^1 \; v \; v' \)
  \( \vdash f^1 \; a \; v' \)

- **Eval-Prim-Error**
  \( \vdash a \; v \; f^1 \; v \; \neg \; v' \)
  \( \vdash f^1 \; a \; \text{error} \)

- **Eval-Var**

- **Eval-Fun**
Rules can be classified into 3 categories:

- Proper evaluation rules: eg. Eval-Fun, Eval-App, describe the evaluation process itself.

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 $\vdash a \ v$. 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 $\parallel a$.

**Implementation notes**

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

```ocaml
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 advantage 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.

```ocaml

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

```ocaml
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.

```ocaml
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
    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.)

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

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, ie. 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 (ie. 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 \( G \). 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.

\[
\text{Types} ::= \ |
g^n \ |
g^n(1, \ldots, n)
\]

Indeed, functional types, ie. the type of functions play a crucial role. Thus, we assume that there is a distinguished type symbol of arity 2, the right arrow 
\( \to \) in \( G \); we also write  \( \to \) for  \( (, ) \). We write \( \text{ftv}(\cdot) \) the set of type variables occurring in \( \cdot \).

Types of programs are given under typing assumptions, also called typing environments, which are partial mappings from program variables 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 \( A, x : a \) for the function that behaves as \( A \) except for \( x \) that is mapped to \( a \) (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 \( A \vdash a : \) and called typing judgments, between type environments \( A \), programs \( a \), and types . We summarize all these definitions (expanding the arrow types) in figure 1.2.

**Figure 1.2: Summary of types, typing environments and judgments**

| Types | ::= | \( g^n \ |
g^n(1, \ldots, n) \) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Typing environments</td>
<td>( A ::= \emptyset</td>
<td>A, z : )</td>
</tr>
<tr>
<td></td>
<td>( z ::= x</td>
<td>c )</td>
</tr>
<tr>
<td>Typing judgments</td>
<td>( A \vdash a : )</td>
<td></td>
</tr>
</tbody>
</table>

Typing judgments are defined as the smallest relation satisfying the inference rules of figure 1.3.
Figure 1.3: Typing rules for simple types

<table>
<thead>
<tr>
<th>Var-Const</th>
<th>Fun</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z )</td>
<td>( A, x : \vdash a : ' )</td>
</tr>
<tr>
<td>( \text{dom} (A) )</td>
<td>( A \vdash x. a : ' )</td>
</tr>
</tbody>
</table>

\( A \vdash a : ' \)

\( A \vdash a' : ' \)

\( A \vdash a.a' : ' \)

(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 assumption.

Assumption 0  [Initial environment] The initial type environment \( A_0 \) has the set of constants for domain, and respects arities. That is, for any \( C^n \) \( \text{dom} (A_0) \) then \( A_0(C^n) \) is of the form \( 1 \ldots n \ldots 0 \).

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 (ie. their associated delta-rules) are chosen accordingly.

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

\( (a \ a') \) \( (A, ) \) \( (A \vdash a : A \vdash a' : ) \)

The relation \( \vdash \) relates the set of typings of two programs programs, regardless of their dynamic properties.

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


Theorem 1  [Subject reduction] Reduction preserves typings

Assumption 2  [Progress for constants] The -reduction is well-defined. If \( A_0 \vdash f^n v_1 \ldots v_n : \), then \( f^n v_1 \ldots v_n \text{dom} ( ) f \)

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 \( \text{let } x = a_1 \text{ in } a_2 \overset{\text{def}}{=} (x. a_2) a_1 \). Hence, we could derived the following typing rule, so as to type those nodes directly:

\( A \vdash a_1 : l \quad A, x : l \vdash a_2 : 2 \)

\( A \vdash \text{let } x = a_1 \text{ in } a_2 : \)
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 \( \tau \), find all substitutions such that \( (A) \vdash a : \tau \). A solution is a principal solution of a problem \( P \) if all other solutions are instances of \( \sigma \), i.e. are of the form \( \sigma \) for some substitution \( \sigma \).

**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 \( (A, a, \tau) \).

**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 \( \tau \) 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 \( A \ a : \tau \) for type inference problems. Note that \( A \ a : \tau \) 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 is a solution to the type inference problem \( A \ a : \tau \) if \( (A) \vdash a : \tau \). 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 \( (A) \vdash a : \tau \) for any substitution.

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.

\[
U ::= A \ a : \tau \mid 1 = \ldots n \mid U \ U \mid U \mid T
\]

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 \( T \) and an unsolvable problem \( U \), although these are definable.

It is convenient to identify constraint problems modulo the following equivalences, which obviously preserve the sets of solutions: the symbol \( \sigma \) is commutative and associative. The constraint problem \( T \) is absorbing and \( T \) is neutral for \( U \), that is \( U \) and \( U \ T = T \). We also treat \( U \) modulo renaming of bound variables, and extrusion of quantifiers; that is, if \( x \) is not free in \( U \), then \( U \) and \( U \ U' = (U \ U') \).

**Figure 1.4: Simplification of type inference problems**

<table>
<thead>
<tr>
<th>I-Var-Fail</th>
<th>I-Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( x ) ( \text{dom} \ (A) )</td>
<td></td>
</tr>
</tbody>
</table>
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 $U \rightarrow U'$ (or

$$U \rightarrow U'$$

) 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.)

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

\[
A \quad \text{let } x = a_1 \text{ in } a_2 : \\

\rightarrow \\
A \quad (x, a_2) a_1 : \\
\]

Let-Mono

\[
A \quad \text{let } x = a_1 \text{ in } a_2 : \\

\rightarrow \\
. (A \ a_1 : \ A, x : \ a_2 : ) \\
\]

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 -> unify (type_of_const c) t
  | Var x -> unify (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;
    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 $\cdot T$, or $\cdot U$ 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 $1 = ... = n = 1$ or $1 = 1$ for short.) Most-general solutions can be obtained straightforwardly from normal forms (that are not $\cdot$).

The first step is to rearrange multi-equations of $U$ into the conjunction $1 = 1 \cdots n = n$ such that a variable of $j$ never occurs in $i$ for $i \neq j$. (Remark, that since $U$ is in normal form, hence completely merged, variables $1, \ldots, n$ are all distinct.) If no such ordering can be found, then there is a cycle and the problem has no solution. Otherwise, the composition $(1 \mid 1 \mid \cdots \mid n \mid n)$ is a principal solution.

For instance, the unification problem $(g_1 \mid 1) = 2 \mid g_2$ can be reduced to the equivalent problem $1 = g_2 \quad 2 = (g_1 \mid 1)$, which is in a solved form. Then, $\{ 1 \mid g_2, \quad 2 \mid (g_1 \mid g_2) \}$ is a most general solution.

**Figure 1.5: Unification rules for simple types**

<table>
<thead>
<tr>
<th>Merge</th>
<th>Decompose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1 = e_2$</td>
<td>$g(i)^{i \mid 1} = g(i)^{i \mid 1} = e$</td>
</tr>
<tr>
<td>$e_1 = e_2$</td>
<td>$g(i)^{i \mid 1} = e \quad i \mid (i = 1)$</td>
</tr>
</tbody>
</table>
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 by a variable and an additional equation \( = \). If it were applied alone, rule **Generalize** would reduce any unification problem into one that only contains small terms, *ie.* 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.

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 \( 1 = 2 = \ldots \) can be represented as a chain of indirections \( 1 \mid 2 \mid \ldots \), where \( \mid \) 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 can be represented by a variable and an equation \( = \), *ie.* an indirection \( \mid \).

A possible implementation for types in OCaml is:

```
type type_symbol = Tarrow | Tint

type texp = { mutable texp : node; mutable mark : int }
```

19 of 27 05/05/06 13:43
The field `mark` of type `texp` is used to mark nodes during recursive visits.

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

```ocaml
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, \(1 = 2 \land 2 = 2\) can be represented as the graph \(1 \mid (2 \mid 2)\) where \(2 \mid\).

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`).

```ocaml
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;;
```

We can now consider the implementation of unification itself. Remember that a type \(=\) is represented by an equation \(=\), 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 \(1 = 1\) and \(2 = 2\). If \(1\) and \(2\) are identical, then so must be \(1\) and \(2\) and the to equations, so the problem is already in solved form. Otherwise, let us consider the multi-equation \(e\) equal to \(1 = 2 = 1 = 2\). If \(1\) is a variable then \(e\) is effectively built by linking \(1\) to \(2\), and conversely if \(2\) 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 \(1 = 2 = 2\) and the equations \(e_i\)'s resulting from the decomposition of \(1 = 2\). The former is implemented by a link from \(1\) to \(2\). The later is implemented by recursive calls to the function `unify`. In case \(1\) and \(2\) are incompatible, then unification fails (rule `Fail`).

```ocaml
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

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;;
```

20 of 27
| 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 c = 
  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 
    visit t; 
  if !cycles <> [] then raise (Cycle !cycles);; 
let unify 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 
unify x (tarrow x x);;
```

```
Uncought 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 \( \forall \sigma . \sigma \) 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 \( \text{let } f = \lambda x . x \text{ in } f ( (x + f 1) ) \). However, this expression does not typecheck, since while any type \( \sigma \) can be chosen for \( \lambda x . x \), 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 \( \tau \) of a set of variables \( \alpha \) and a type \( \sigma \). We identify \( \varnothing \) with the empty type scheme \( \varnothing \). We use the letter \( \tau \) to represent type schemes. An instance of a type scheme \( \tau \) is a type of the form \( \tau[\sigma] \) obtained by a simultaneous substitution in \( \sigma \) of all quantified variables \( \alpha \) by simple types \( \sigma \) in \( \tau \). (Note that the notation \( \tau[\sigma] \) is an abbreviation for \( (\alpha \mapsto \sigma)() ) \).

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

\[
\text{ftv}(A) = \text{ftv}(A(z))
\]
The representation of type schemes is straightforward (although other representations are possible).

\[
\text{type scheme} = \text{texp list} \to \text{texp};
\]

**Exercise 4** [(*) Free type variables for recursive types] Implement the function \( \text{ftv_type} \) that computes \( \text{ftv}() \) for types (as a slight modification to the function \( \text{acyclic} \)).

**Answer**

Write a (simple version of a) function \( \text{type_instance} \) taking type scheme \( A \) as argument and returning a type instance of \( A \) obtained by renaming and stripping off the bound variables of \( A \) (you may assume that \( A \) 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 \( z : \) in typing contexts. Since we can see a type \( \) as trivial type scheme \( \emptyset \), we can assumes that all bindings are of the form \( z : \). Thus, we change rule \( \text{Var} \) to:

\[
\text{Var-Const} \\
A(z) = \\
A \mid z : \quad [\quad ]
\]

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:

\[
\text{Let} \\
A \mid a : \quad A, x : \quad (\text{ftv}(\) \setminus \text{ftv}(A)). \quad \mid a' : \quad'
\]

That is, the type found for the expression \( a \) bound to \( x \) must be generalized “as much as possible”, that is, with respect to all variables appearing in \( a \) 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: \( \alpha \)-bound variables remain monomorphic.

In summary, the set of typing rules of ML is composed of rules \( \text{Fun, App} \) from figure 1.3 plus rules \( \text{Var-Const} \) and \( \text{Let} \) 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 \( I-\text{Fun}, I-\text{App} \) do not need to be changed. The rewriting rule \( I-\text{Var} \) 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:

\[ I-\text{Var} \]

\[
\begin{align*}
\text{if} & \quad .\.: = A(x) \\
\text{and} & \quad \text{ftv}(\ ) = \emptyset
\end{align*}
\]

\[ A \quad x : \\
.\. = .\.
\]

The \textbf{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.

\[ I-\text{Let} \]

\[
\begin{align*}
\text{if} & \quad \text{ftv}(A), A \quad a_1 : \quad U \\
\text{and} & \quad U \text{ solved}, U
\end{align*}
\]

\[ A \quad \text{let } x = a_1 \text{ in } a_2 : 2 \\
A, x : \quad .\. U(\ ) |- a_2 : 2
\]

where \( U(\ ) \) is a principal solution of \( U \).

\section*{Implementation notes, file \texttt{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 \texttt{type_instance} to be used for the implementation of the rule \textbf{Var-Const}. We also need a function \texttt{generalizable} to compute generalizable variables \( \text{ftv}(t) \setminus \text{ftv}(A) \) from a type environment \( A \) and a type \( t \). The obvious implementation would compute \( \text{ftv}(\ ) \) and \( \text{ftv}(A) \) 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.

\textbf{Exercise 5} [\( ** \) Generalizable variables] Generalize the implementation of \texttt{ftv_type} so as to obtain a direct implementation of generalizable variables.

\textbf{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 \texttt{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:

\begin{verbatim}
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
\end{verbatim}
1.5 Recursion

So as to be Turing-complete, ML should allow a form of recursion. This is provided by the `let rec f = x. a1 in a2` form, which allows `f` to appear in `x. a1`, recursively. The recursive expression `x. a1` 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:

\[
\text{fix} : \quad 1,2.\ ((1\ 2)\ 1\ 2)\ 1\ 2
\]

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

\[
\text{fix} f v = f (\text{fix} f) v (\text{fix})
\]

Since `fix` is of arity 2, the expression `(fix f)` appearing on the right hand side of the rule `(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 -rule,

\[
\text{fix} f = f (\text{fix} f)
\]

the evaluation of `fix f v` would fall into an infinite loop (the active part is underlined):

\[
\text{fix} f v = f (\text{fix} f) v (\text{fix} f) v \ldots
\]

For convenience, we may use `let rec f = x. a1 in a2` as syntactic sugar for `let f = fix ( f. x. a1 ) in a2`.

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

\[
\text{fix}. ( f. x. f (f f) x) ( f. x. f (f f) x)
\]

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).
Exercise 8 [(**) Type soundness of the fix-point combinator] Check that the hypotheses 1 and 2 are satisfied for the fix-point combinator \( \text{fix} \).

Mutually recursive definitions

The language OCaml allows mutually recursive definitions. For example,

```ml
let rec f1 = x. a1 and f2 = x. a2 in a
```

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

```ml
let rec f1 = f2. x. let f1 = f1 f2 in a1

in

let rec f2 = x. let f2 = f2 in

let f1 = f1 f2 in

a2

in

a
```

This can be easily generalize to

```ml
let rec f1 = x. a1 and ... fn x. an in a
```

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

```ml
let rec f1 = x. a1 and f2 = x. a2 and f3 = x. a3 in a
```

Recursion and polymorphism

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

Use let-binding to write a shorter equivalent version of fix.

**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).

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.

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].