Notes du cours de remise à niveau "Programmation, sémantique, typage"

Xavier Leroy
Xavier.Leroy@inria.fr

15 septembre 2005
http://cristal.inria.fr/~xleroy/mpri/prog/

Table des matières

Chapitre 1

Rappels de programmation en Caml

This part of the manual is a tutorial introduction to the Objective Caml language. A good familiarity with programming in a conventional languages (say, Pascal or C) is assumed, but no prior exposure to functional languages is required. The present chapter introduces the core language. Chapter 2 of the Objective Caml reference manual deals with the object-oriented features, and chapter 3 with the module system.

1.1 Basics

For this overview of Caml, we use the interactive system, which is started by running `ocaml` from the Unix shell, or by launching the `OCamlwin.exe` application under Windows. This tutorial is presented as the transcript of a session with the interactive system: lines starting with `#` represent user input; the system responses are printed below, without a leading `#`.

Under the interactive system, the user types Caml phrases, terminated by `;;`, in response to the `#` prompt, and the system compiles them on the fly, executes them, and prints the outcome of evaluation. Phrases are either simple expressions, or `let` definitions of identifiers (either values or functions).

```
# 1+2*3;;
- : int = 7

# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312

# let square x = x *. x;;
val square : float -> float = <fun>

# square(sin pi) +. square(cos pi);;
- : float = 1.
```

The Caml system computes both the value and the type for each phrase. Even function parameters need no explicit type declaration: the system infers their types from their usage in the function. Notice also that integers and floating-point numbers are distinct types, with distinct operators: `+` and `*` operate on integers, but `+.` and `*.` operate on floats.

```
# 1.0 * 2;;
This expression has type float but is here used with type int
```

Recursive functions are defined with the `let rec` binding:

```
# 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
```

1.2 Data types

In addition to integers and floating-point numbers, Caml offers the usual basic data types: booleans, characters, and character strings.

```
# (1 < 2) = false;;
- : bool = false

# 'a';;
- : char = 'a'

# "Hello world";;
- : string = "Hello world"
```

Predefined data structures include tuples, arrays, and lists. General mechanisms for defining your own data structures are also provided. They will be covered in more details later; for now, we concentrate on lists. Lists are either given in extension as a bracketed list of semicolon-separated elements, or built from the empty list `[]` (pronounce "nil") by adding elements in front using the `::` ("cons") operator.

```
# let l = ["is"; "a"; "tale"; "told"; "etc."];;
val l : string list = ["is"; "a"; "tale"; "told"; "etc."]

# "Life" :: l;;
- : string list = ["Life"; "is"; "a"; "tale"; "told"; "etc."]
```

As with all other Caml data structures, lists do not need to be explicitly allocated and deallocated from memory: all memory management is entirely automatic in Caml. Similarly, there is no explicit handling of pointers: the Caml compiler silently introduces pointers where necessary.

As with most Caml data structures, inspecting and destructuring lists is performed by pattern-matching. List patterns have the exact same shape as list expressions, with identifier representing unspecified parts of the list. As an example, here is insertion sort on a list:

```
# let rec sort lst =
#   match lst with
#   | [] -> []
#   | head :: tail -> insert head (sort tail)
# and insert elt lst =
#   match lst with
#   | [] -> [elt]
#   | head :: tail -> if elt <= head then elt :: lst else head :: insert elt tail
```

```
# ;;
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>

# sort l;;
- : string list = ["a"; "etc."; "is"; "tale"; "told"]
```

The type inferred for `sort`, `'a list -> 'a list`, means that `sort` can actually apply to lists of any type, and returns a list of the same type. The type `'a` is a *type variable*, and stands for any given type. The reason why `sort` can apply to lists of any type is that the comparisons (`=`, `<=`, etc.) are *polymorphic* in Caml: they operate between any two values of the same type. This makes `sort` itself polymorphic over all list types.

```
# sort [6;2;5;3];;
- : int list = [2; 3; 5; 6]

# sort [3.14; 2.718];;
- : float list = [2.718; 3.14]
```

The `sort` function above does not modify its input list: it builds and returns a new list containing the same elements as the input list, in ascending order. There is actually no way in Caml to modify in-place a list once it is built: we say that lists are *immutable* data structures. Most Caml data structures are immutable, but a few (most notably arrays) are *mutable*, meaning that they can be modified in-place at any time.

1.3 Functions as values

Caml is a functional language: functions in the full mathematical sense are supported and can be passed around freely just as any other piece of data. For instance, here is a `deriv` function that takes any float function as argument and returns an approximation of its derivative function:

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

# let sin' = deriv sin 1e-6;;
val sin' : float -> float = <fun>

# sin' pi;;
- : float = -1.00000000013961143
```

Even function composition is definable:

```
# let compose f g = function x -> f(g(x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let cos2 = compose square cos;;
val cos2 : float -> float = <fun>
```

Functions that take other functions as arguments are called "functionals", or "higher-order functions". Functionals are especially useful to provide iterators or similar generic operations over a data structure. For instance, the standard Caml library provides a `List.map` functional that applies a given function to each element of a list, and returns the list of the results:

```
# List.map (function n -> n * 2 + 1) [0;1;2;3;4];;
- : int list = [1; 3; 5; 7; 9]
```

This functional, along with a number of other list and array functionals, is predefined because it is often useful, but there is nothing magic with it: it can easily be defined as follows.

```
# let rec map f l =
#   match l with
#   | [] -> []
#   | hd :: tl -> f hd :: map f tl;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

1.4 Records and variants

User-defined data structures include records and variants. Both are defined with the `type` declaration. Here, we declare a record type to represent rational numbers.

```
# type ratio = {num: int; denum: int};;
type ratio = { num : int; denum : int; }
```

```
# let add_ratio r1 r2 =
#   {num = r1.num * r2.denum + r2.num * r1.denum;
#    denum = r1.denum * r2.denum};;
val add_ratio : ratio -> ratio -> ratio = <fun>
```

```
# add_ratio {num=1; denum=3} {num=2; denum=5};;
- : ratio = {num = 11; denum = 15}
```

The declaration of a variant type lists all possible shapes for values of that type. Each case is identified by a name, called a constructor, which serves both for constructing values of the variant type and inspecting them by pattern-matching. Constructor names are capitalized to distinguish them from variable names (which must start with a lowercase letter). For instance, here is a variant type for doing mixed arithmetic (integers and floats):

```
# type number = Int of int | Float of float | Error;;
type number = Int of int | Float of float | Error
```

This declaration expresses that a value of type `number` is either an integer, a floating-point number, or the constant `Error` representing the result of an invalid operation (e.g. a division by zero).

Enumerated types are a special case of variant types, where all alternatives are constants:

```
# type sign = Positive | Negative;;
type sign = Positive | Negative
```

```
# let sign_int n = if n >= 0 then Positive else Negative;;
val sign_int : int -> sign = <fun>
```

To define arithmetic operations for the `number` type, we use pattern-matching on the two numbers involved:

```
# let add_num n1 n2 =
#   match (n1, n2) with
#   | (Int i1, Int i2) ->
#       (* Check for overflow of integer addition *)
#       if sign_int i1 = sign_int i2 && sign_int(i1 + i2) <> sign_int i1
#       then Float(float i1 +. float i2)
#       else Int(i1 + i2)
#   | (Int i1, Float f2) -> Float(float i1 +. f2)
#   | (Float f1, Int i2) -> Float(f1 +. float i2)
#   | (Float f1, Float f2) -> Float(f1 +. f2)
#   | (Error, _) -> Error
#   | (_, Error) -> Error;;
val add_num : number -> number -> number = <fun>

# add_num (Int 123) (Float 3.14159);;
- : number = Float 126.14159
```

The most common usage of variant types is to describe recursive data structures. Consider for example the type of binary trees:

```
# type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

This definition reads as follow: a binary tree containing values of type `'a` (an arbitrary type) is either empty, or is a node containing one value of type `'a` and two subtrees containing also values of type `'a`, that is, two `'a btree`.

Operations on binary trees are naturally expressed as recursive functions following the same structure as the type definition itself. For instance, here are functions performing lookup and insertion in ordered binary trees (elements increase from left to right):

```
# let rec member x btree =
#   match btree with
#   | Empty -> false
#   | Node(y, left, right) ->
#       if x = y then true else
#       if x < y then member x left else member x right;;
val member : 'a -> 'a btree -> bool = <fun>

# let rec insert x btree =
#   match btree with
#   | Empty -> Node(x, Empty, Empty)
#   | Node(y, left, right) ->
#       if x <= y then Node(y, insert x left, right)
#                 else Node(y, left, insert x right);;
val insert : 'a -> 'a btree -> 'a btree = <fun>
```

1.5 Imperative features

Though all examples so far were written in purely applicative style, Caml is also equipped with full imperative features. This includes the usual `while` and `for` loops, as well as mutable data structures such as arrays. Arrays are either given in extension between `[|` and `|]` brackets, or allocated and initialized with the `Array.create` function, then filled up later by assignments. For instance, the function below sums two vectors (represented as float arrays) componentwise.

```
# let add_vect v1 v2 =
#   let len = min (Array.length v1) (Array.length v2) in
#   let res = Array.create len 0.0 in
#   for i = 0 to len - 1 do
#     res.(i) <- v1.(i) +. v2.(i)
#   done;
#   res;;
val add_vect : float array -> float array -> float array = <fun>

# add_vect [| 1.0; 2.0 |] [| 3.0; 4.0 |];;
- : float array = [|4.; 6.|]
```

Record fields can also be modified by assignment, provided they are declared `mutable` in the definition of the record type:

```
# type mutable_point = { mutable x: float; mutable y: float };;
type mutable_point = { mutable x : float; mutable y : float; }

# let translate p dx dy =
#   p.x <- p.x +. dx; p.y <- p.y +. dy;;
val translate : mutable_point -> float -> float -> unit = <fun>

# let mypoint = { x = 0.0; y = 0.0 };;
val mypoint : mutable_point = {x = 0.; y = 0.}

# translate mypoint 1.0 2.0;;
- : unit = ()

# mypoint;;
- : mutable_point = {x = 1.; y = 2.}
```

Caml has no built-in notion of variable – identifiers whose current value can be changed by assignment. (The `let` binding is not an assignment, it introduces a new identifier with a new scope.) However, the standard library provides references, which are mutable indirection cells (or one-element arrays), with operators `!` to fetch the current contents of the reference and `:=` to assign the contents. Variables can then be emulated by `let`-binding a reference. For instance, here is an in-place insertion sort over arrays:

```
# let insertion_sort a =
#   for i = 1 to Array.length a - 1 do
#     let val_i = a.(i) in
#     let j = ref i in
#     while !j > 0 && val_i < a.(!j - 1) do
```

```
#       a.(!j) <- a.(!j - 1);
#       j := !j - 1
#     done;
#     a.(!j) <- val_i
#   done;;
val insertion_sort : 'a array -> unit = <fun>
```

References are also useful to write functions that maintain a current state between two calls to the function. For instance, the following pseudo-random number generator keeps the last returned number in a reference:

```
# let current_rand = ref 0;;
val current_rand : int ref = {contents = 0}

# let random () =
#   current_rand := !current_rand * 25713 + 1345;
#   !current_rand;;
val random : unit -> int = <fun>
```

Again, there is nothing magic with references: they are implemented as a one-field mutable record, as follows.

```
# type 'a ref = { mutable contents: 'a };;
type 'a ref = { mutable contents : 'a; }

# let (!) r = r.contents;;
val ( ! ) : 'a ref -> 'a = <fun>

# let (:=) r newval = r.contents <- newval;;
val ( := ) : 'a ref -> 'a -> unit = <fun>
```

In some special cases, you may need to store a polymorphic function in a data structure, keeping its polymorphism. Without user-provided type annotations, this is not allowed, as polymorphism is only introduced on a global level. However, you can give explicitly polymorphic types to record fields.

```
# type idref = { mutable id: 'a. 'a -> 'a };;
type idref = { mutable id : 'a. 'a -> 'a; }

# let r = {id = fun x -> x};;
val r : idref = {id = <fun>}

# let g s = (s.id 1, s.id true);;
val g : idref -> int * bool = <fun>

# r.id <- (fun x -> print_string "called id\n"; x);;
- : unit = ()

# g r;;
called id
called id
- : int * bool = (1, true)
```

1.6 Exceptions

Caml provides exceptions for signalling and handling exceptional conditions. Exceptions can also be used as a general-purpose non-local control structure. Exceptions are declared with the `exception` construct, and signalled with the `raise` operator. For instance, the function below for taking the head of a list uses an exception to signal the case where an empty list is given.

```
# exception Empty_list;;
exception Empty_list

# let head l =
#   match l with
#   | [] -> raise Empty_list
#   | hd :: tl -> hd;;
val head : 'a list -> 'a = <fun>

# head [1;2];;
- : int = 1

# head [];;
Exception: Empty_list.
```

Exceptions are used throughout the standard library to signal cases where the library functions cannot complete normally. For instance, the `List.assoc` function, which returns the data associated with a given key in a list of (key, data) pairs, raises the predefined exception `Not_found` when the key does not appear in the list:

```
# List.assoc 1 [(0, "zero"); (1, "one")];;
- : string = "one"

# List.assoc 2 [(0, "zero"); (1, "one")];;
Exception: Not_found.
```

Exceptions can be trapped with the `try`...`with` construct:

```
# let name_of_binary_digit digit =
#   try
#     List.assoc digit [0, "zero"; 1, "one"]
#   with Not_found ->
#     "not a binary digit";;
val name_of_binary_digit : int -> string = <fun>

# name_of_binary_digit 0;;
- : string = "zero"

# name_of_binary_digit (-1);;
- : string = "not a binary digit"
```

The `with` part is actually a regular pattern-matching on the exception value. Thus, several exceptions can be caught by one `try`...`with` construct. Also, finalization can be performed by trapping all exceptions, performing the finalization, then raising again the exception:

```
# let temporarily_set_reference ref newval funct =
#   let oldval = !ref in
#   try
#     ref := newval;
#     let res = funct () in
#     ref := oldval;
#     res
#   with x ->
#     ref := oldval;
#     raise x;;
val temporarily_set_reference : 'a ref -> 'a -> (unit -> 'b) -> 'b = <fun>
```

1.7 Symbolic processing of expressions

We finish this introduction with a more complete example representative of the use of Caml for symbolic processing: formal manipulations of arithmetic expressions containing variables. The following variant type describes the expressions we shall manipulate:

```
# type expression =
#     Const of float
#   | Var of string
#   | Sum of expression * expression    (* e1 + e2 *)
#   | Diff of expression * expression   (* e1 - e2 *)
#   | Prod of expression * expression   (* e1 * e2 *)
#   | Quot of expression * expression   (* e1 / e2 *)
# ;;
type expression =
    Const of float
  | Var of string
  | Sum of expression * expression
  | Diff of expression * expression
  | Prod of expression * expression
  | Quot of expression * expression
```

We first define a function to evaluate an expression given an environment that maps variable names to their values. For simplicity, the environment is represented as an association list.

```
# exception Unbound_variable of string;;
exception Unbound_variable of string
```

```
# let rec eval env exp =
#   match exp with
#   | Const c -> c
#   | Var v ->
#       (try List.assoc v env with Not_found -> raise(Unbound_variable v))
#   | Sum(f, g) -> eval env f +. eval env g
#   | Diff(f, g) -> eval env f -. eval env g
```

```
#   | Prod(f, g) -> eval env f *. eval env g
#   | Quot(f, g) -> eval env f /. eval env g;;
val eval : (string * float) list -> expression -> float = <fun>

# eval [("x", 1.0); ("y", 3.14)] (Prod(Sum(Var "x", Const 2.0), Var "y"));;
- : float = 9.42
```

Now for a real symbolic processing, we define the derivative of an expression with respect to a variable `dv`:

```
# let rec deriv exp dv =
#   match exp with
#   | Const c -> Const 0.0
#   | Var v -> if v = dv then Const 1.0 else Const 0.0
#   | Sum(f, g) -> Sum(deriv f dv, deriv g dv)
#   | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
#   | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
#   | Quot(f, g) -> Quot(Diff(Prod(deriv f dv, g), Prod(f, deriv g dv)),
#                        Prod(g, g))
# ;;
val deriv : expression -> string -> expression = <fun>

# deriv (Quot(Const 1.0, Var "x")) "x";;
- : expression =
Quot (Diff (Prod (Const 0., Var "x"), Prod (Const 1., Const 1.)),
 Prod (Var "x", Var "x"))
```

1.8 Pretty-printing and parsing

As shown in the examples above, the internal representation (also called *abstract syntax*) of expressions quickly becomes hard to read and write as the expressions get larger. We need a printer and a parser to go back and forth between the abstract syntax and the *concrete syntax*, which in the case of expressions is the familiar algebraic notation (e.g. `2*x+1`).

For the printing function, we take into account the usual precedence rules (i.e. `*` binds tighter than `+`) to avoid printing unnecessary parentheses. To this end, we maintain the current operator precedence and print parentheses around an operator only if its precedence is less than the current precedence.

```
# let print_expr exp =
#   (* Local function definitions *)
#   let open_paren prec op_prec =
#     if prec > op_prec then print_string "(" in
#   let close_paren prec op_prec =
#     if prec > op_prec then print_string ")" in
#   let rec print prec exp =     (* prec is the current precedence *)
#     match exp with
#     | Const c -> print_float c
```

```
#     | Var v -> print_string v
#     | Sum(f, g) ->
#         open_paren prec 0;
#         print 0 f; print_string " + "; print 0 g;
#         close_paren prec 0
#     | Diff(f, g) ->
#         open_paren prec 0;
#         print 0 f; print_string " - "; print 1 g;
#         close_paren prec 0
#     | Prod(f, g) ->
#         open_paren prec 2;
#         print 2 f; print_string " * "; print 2 g;
#         close_paren prec 2
#     | Quot(f, g) ->
#         open_paren prec 2;
#         print 2 f; print_string " / "; print 3 g;
#         close_paren prec 2
#   in print 0 exp;;
val print_expr : expression -> unit = <fun>

# let e = Sum(Prod(Const 2.0, Var "x"), Const 1.0);;
val e : expression = Sum (Prod (Const 2., Var "x"), Const 1.)

# print_expr e; print_newline();;
2. * x + 1.
- : unit = ()

# print_expr (deriv e "x"); print_newline();;
2. * 1. + 0. * x + 0.
- : unit = ()
```

Parsing (transforming concrete syntax into abstract syntax) is usually more delicate. Caml offers several tools to help write parsers: on the one hand, Caml versions of the lexer generator Lex and the parser generator Yacc (see chapter 12 of the reference manual), which handle LALR(1) languages using push-down automata; on the other hand, a predefined type of streams (of characters or tokens) and pattern-matching over streams, which facilitate the writing of recursive-descent parsers for LL(1) languages. An example using `ocamllex` and `ocamlyacc` is given in chapter 12 of the reference manual. Here, we will use stream parsers. The syntactic support for stream parsers is provided by the Camlp4 preprocessor, which can be loaded into the interactive toplevel via the `#load` directive below.

```
# #load "camlp4o.cma";;
Camlp4 Parsing version 3.08.3+4 (2005-06-21)

# open Genlex;;

# let lexer = make_lexer ["("; ")"; "+"; "-"; "*"; "/"];;
val lexer : char Stream.t -> Genlex.token Stream.t = <fun>
```

For the lexical analysis phase (transformation of the input text into a stream of tokens), we use a "generic" lexer provided in the standard library module `Genlex`. The `make_lexer` function takes

a list of keywords and returns a lexing function that "tokenizes" an input stream of characters. Tokens are either identifiers, keywords, or literals (integer, floats, characters, strings). Whitespace and comments are skipped.

```
# let token_stream = lexer(Stream.of_string "1.0 +x");;
val token_stream : Genlex.token Stream.t = <abstr>

# Stream.next token_stream;;
- : Genlex.token = Float 1.

# Stream.next token_stream;;
- : Genlex.token = Kwd "+"

# Stream.next token_stream;;
- : Genlex.token = Ident "x"
```

The parser itself operates by pattern-matching on the stream of tokens. As usual with recursive descent parsers, we use several intermediate parsing functions to reflect the precedence and associativity of operators. Pattern-matching over streams is more powerful than on regular data structures, as it allows recursive calls to parsing functions inside the patterns, for matching sub-components of the input stream. See the Camlp4 documentation for more details.

```
# let rec parse_expr = parser
#     [< e1 = parse_mult; e = parse_more_adds e1 >] -> e
# and parse_more_adds e1 = parser
#     [< 'Kwd "+"; e2 = parse_mult; e = parse_more_adds (Sum(e1, e2)) >] -> e
#   | [< 'Kwd "-"; e2 = parse_mult; e = parse_more_adds (Diff(e1, e2)) >] -> e
#   | [< >] -> e1
# and parse_mult = parser
#     [< e1 = parse_simple; e = parse_more_mults e1 >] -> e
# and parse_more_mults e1 = parser
#     [< 'Kwd "*"; e2 = parse_simple; e = parse_more_mults (Prod(e1, e2)) >] -> e
#   | [< 'Kwd "/"; e2 = parse_simple; e = parse_more_mults (Quot(e1, e2)) >] -> e
#   | [< >] -> e1
# and parse_simple = parser
#     [< 'Ident s >] -> Var s
#   | [< 'Int i >] -> Const(float i)
#   | [< 'Float f >] -> Const f
#   | [< 'Kwd "("; e = parse_expr; 'Kwd ")" >] -> e;;
val parse_expr : Genlex.token Stream.t -> expression = <fun>
val parse_more_adds : expression -> Genlex.token Stream.t -> expression =
  <fun>
val parse_mult : Genlex.token Stream.t -> expression = <fun>
val parse_more_mults : expression -> Genlex.token Stream.t -> expression =
  <fun>
val parse_simple : Genlex.token Stream.t -> expression = <fun>

# let parse_expression = parser [< e = parse_expr; _ = Stream.empty >] -> e;;
val parse_expression : Genlex.token Stream.t -> expression = <fun>
```

Composing the lexer and parser, we finally obtain a function to read an expression from a character string:

```
# let read_expression s = parse_expression(lexer(Stream.of_string s));;
val read_expression : string -> expression = <fun>

# read_expression "2*(x+y)";;
- : expression = Prod (Const 2., Sum (Var "x", Var "y"))
```

A small puzzle: why do we get different results in the following two examples?

```
# read_expression "x - 1";;
- : expression = Diff (Var "x", Const 1.)

# read_expression "x-1";;
Exception: Stream.Error "".
```

Answer: the generic lexer provided by `Genlex` recognizes negative integer literals as one integer token. Hence, `x-1` is read as the token `Ident "x"` followed by the token `Int(-1)`; this sequence does not match any of the parser rules. On the other hand, the second space in `x - 1` causes the lexer to return the three expected tokens: `Ident "x"`, then `Kwd "-"`, then `Int(1)`.

1.9 Standalone Caml programs

All examples given so far were executed under the interactive system. Caml code can also be compiled separately and executed non-interactively using the batch compilers `ocamlc` or `ocamlopt`. The source code must be put in a file with extension `.ml`. It consists of a sequence of phrases, which will be evaluated at runtime in their order of appearance in the source file. Unlike in interactive mode, types and values are not printed automatically; the program must call printing functions explicitly to produce some output. Here is a sample standalone program to print Fibonacci numbers:

```
(* File fib.ml *)
let rec fib n =
  if n < 2 then 1 else fib(n-1) + fib(n-2);;
let main () =
  let arg = int_of_string Sys.argv.(1) in
  print_int(fib arg);
  print_newline();
  exit 0;;
main ();;
```

`Sys.argv` is an array of strings containing the command-line parameters. `Sys.argv.(1)` is thus the first command-line parameter. The program above is compiled and executed with the following shell commands:

```
$ ocamlc -o fib fib.ml
$ ./fib 10
89
$ ./fib 20
10946
```

Chapitre 2

Sémantique opérationnelle

2.1 Généralités sur la sémantique

La sémantique d'un langage de programmation consiste à définir formellement la signification des programmes écrits dans ce langage, c'est-à-dire à caractériser mathématiquement les calculs décrits par un programme et les résultats qu'il produit.

La sémantique d'un langage est complémentaire de sa syntaxe : la syntaxe décrit comment représenter les programmes sous forme de suite de caractères, et symétriquement comment relire ces suites de caractères sous forme d'un enchaînement d'opérations élémentaires du langage. Par exemple, comment passer de la chaîne `1+2` à "l'opérateur d'addition appliqué aux deux constantes entières 1 et 2". La sémantique décrit comment passer de cette représentation abstraite du programme à son résultat – dans l'exemple, l'entier 3 (en général).

Généralement, les standards et textes de référence sur les langages de programmation définissent la sémantique d'un langage de manière informelle, en anglais ou dans une autre langue naturelle. De telles définitions sont souvent partielles ou ambiguës. Pour éviter cela, la sémantique formelle utilise les mathématiques au lieu de la langue naturelle pour définir de manière entièrement précise le comportement des programmes. Connaître la sémantique formelle d'un langage est crucial dans plusieurs situations :

- Pour prouver un programme écrit dans ce langage par-rapport à une spécification mathématique des résultats attendus.
- Pour développer des outils de programmation corrects pour ce langage : interprètes, compilateurs, analyseurs statiques, ...

De nombreuses approches de la sémantique formelle ont été étudiées. Sans être exhaustif, citons-en 4.

Sémantique axiomatique (logique de Hoare) La sémantique axiomatique décrit le comportement des programmes impératifs par l'intermédiaire des propriétés (formules logiques) entre les valeurs des variables qui sont vraies aux différents points du programme. Par exemple, la sémantique de l'affectation

$$\texttt{x := x + y}$$

est donnée par la relation logique

$$\{P(x+y)\} \quad \texttt{x := x + y} \quad \{P(x)\}$$

qui se lit ainsi : “la propriété P de la valeur x de la variable `x` est vraie juste après exécution de `x := x + y` seulement si cette même propriété P est vraie de $x + y$ (la valeur de `x` plus celle de `y`) juste avant cette exécution”. Dans le “triplet de Hoare” $\{P\}$ `instr` $\{Q\}$, les formules P et Q sont appelées respectivement précondition et postcondition de l'instruction `instr`.

Exemple : Voici un exemple de programme impératif annoté avec des préconditions et postconditions :

$$\{\ x = N\ \wedge\ N \geq 0\ \}$$

```
y := 1 ;
```

$$\{\ x \geq 0\ \wedge\ y = (N - x)!\ \}$$

```
while (x > 0) do
```

$$\{\ x > 0\ \wedge\ y = (N - x)!\ \}$$

```
    y := y * x ;
```

$$\{\ x > 0\ \wedge\ y = (N - x + 1)!\ \}$$

```
    x := x - 1 ;
```

$$\{\ x \geq 0\ \wedge\ y = (N - x)!\ \}$$

```
done
```

$$\{\ x = 0\ \wedge\ y = N!\ \}$$

Ce code calcule donc bien la factorielle de `x` pourvu que `x` soit positif ou nul à l'entrée.

La sémantique axiomatique est très utile pour prouver la correction de programmes, tout particulièrement de programmes impératifs. Elle n'est pas très utile en revanche comme guide pour implémenter des interpréteurs ou des compilateurs, car elle ne dit pas directement comment le programme mène ses calculs.

Sémantique dénotationnelle La sémantique dénotationnelle associe à chaque programme, instruction ou sous-expression sa *dénotation*, c'est-à-dire un objet mathématique qui représente le résultat du calcul. Par exemple, pour le langage des expressions arithmétiques avec une variable `x` ($e ::= \texttt{x} \mid n \mid e + e \mid e * e \mid \ldots$), la dénotation $[\![e]\!]$ d'une expression e est la fonction mathématique qui envoie la valeur entière de `x` sur la valeur entière correspondante de e :

$$\begin{aligned}
[\![\texttt{x}]\!] &= x \mapsto x \\
[\![n]\!] &= x \mapsto n \\
[\![e_1 + e_2]\!] &= x \mapsto ([\![e_1]\!](x) + [\![e_2]\!](x)) \\
[\![e_1 * e_2]\!] &= x \mapsto ([\![e_1]\!](x) \times [\![e_2]\!](x))
\end{aligned}$$

Exemple : La dénotation de `(x + 1) * 2` est donc $x \mapsto (x + 1) \times 2$.

Pour des langages plus riches comme les langages fonctionnels et le λ-calcul, la sémantique dénotationnelle est plus difficile : il faut construire un ensemble adéquat D de dénotations. (Cet ensemble s'appelle le domaine de la sémantique.) Avec une interprétation naïve des λ-abstractions comme des fonctions mathématiques de D dans D, il faudrait que l'ensemble D contienne l'ensemble $D \to D$, ce qui n'est pas possible. Il faut donc recourir à des dénotations ayant une structure mathématique plus complexe, en particulier les domaines de Scott et les stratégies en théorie des jeux. Voir le cours “Modèles des langages de programmation” (cours 2-1).

Sémantique par traduction ; sémantique dénotationnelle à la Strachey Un moyen indirect de définir la sémantique d'un langage est de donner une traduction de ce langage dans un langage dont la sémantique est déjà connue. Historiquement, le λ-calcul a souvent été utilisé comme langage cible pour ce faire. Ce type de sémantique par traduction vers le λ-calcul s'appelle sémantique dénotationnelle "à la Strachey", par opposition à la sémantique dénotationnelle "à la Scott" décrite au paragraphe précédent. Si on dispose d'une traduction "à la Strachey" d'un langage vers le λ-calcul, on peut composer cette traduction avec une sémantique dénotationnelle "à la Scott" du λ-calcul, obtenant ainsi une sémantique dénotationnelle "à la Scott" du langage de départ.

Sémantique opérationnelle La sémantique opérationnelle s'attache à décrire mathématiquement le déroulement de l'exécution d'un programme, c'est-à-dire l'enchaînement des calculs élémentaires effectués par le programme. Le résultat final (la valeur d'une expression p.ex.) se déduit de cet enchaînement. Cette approche se distingue des sémantiques dénotationnelles et axiomatiques sur deux points principaux : 1- ces dernières exposent beaucoup moins l'enchaînement des calculs ; 2- elles utilisent des objets mathématiques plus riches que la sémantique opérationnelle, qui n'utilise que des objets syntaxiques (termes) proches de la syntaxe du programme.

Dans le reste de ce cours, nous étudierons deux formes de sémantique opérationnelle :

- La sémantique "à grands pas" (*big-step semantics*), aussi appelée "sémantique naturelle". Elle se présente sous forme d'une relation entre les expressions du langage et les valeurs en lesquelles elles s'évaluent. On peut la voir comme une présentation mathématique d'un interprète pour le langage considéré.
- La sémantique "à petits pas" (*small-step semantics*), aussi appelée "sémantique à réductions". Elle se présente comme une suite de réécritures du programme, chaque réécriture correspondant à une étape élémentaire de calcul. Elle généralise la notion de réduction des termes du λ-calcul.

Comme langage d'étude, nous utiliserons mini-ML, un petit langage fonctionnel, sous-ensemble de Caml. Parmi les cours du M2 qui utilisent beaucoup de sémantiques opérationnelles, citons 2-3 (Concurrence) et 2-4 (Langages de programmation).

2.2 Le langage mini-ML

Le langage mini-ML se compose d'expressions (notées a, a_1, a', ...) dont la syntaxe abstraite est la suivante :

Expressions :	$a ::= x$		identificateur (nom de variable)
	$\mid c$		constante
	$\mid op$		opération primitive
	$\mid \texttt{fun}\ x \to a$		abstraction de fonction
	$\mid a_1\ a_2$		application de fonction
	$\mid (a_1, a_2)$		construction d'une paire
	$\mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$		liaison locale

Nous omettons les détails de la syntaxe concrète (forme des identificateurs, parenthèses, priorités des opérations, ...).

La classe c contient des constantes comme par exemple des constantes entières 0, 1, 2, -1, ..., les booléens `true`, `false`, ou des chaînes littérales `"foo"`, ... La classe *op* contient des symboles d'opérations primitives, comme l'addition entière `+`, les projections `fst` et `snd` pour accéder aux composantes d'une paire, etc.

Exemples d'expressions :

```
+ (3, 2)
```
le calcul de 3 plus 2

```
3 + 2
```
le même, avec notation infixe pour le `+`

```
fun x → + (x, 1)
```
la fonction "successeur"

```
fun f → fun g → fun x → f(g x)
```
la composition de fonctions

```
let double = fun f → fun x → f(f x) in
let fois2 = fun x → + (x, x) in
let fois4 = double fois2 in
  double fois4
```
la fonction "fois 16"

D'autres constructions essentielles des langages de programmation peuvent également être présentées sous formes d'opérateurs prédéfinis. Par exemple, l'expression conditionnelle `if` $a_1 =$ 0 `then` a_2 `else` a_3 peut être vue comme l'application d'un opérateur `ifzero` à $(a_1, (a_2, a_3))$.

Plus surprenant : la définition de fonctions récursives peut également être ajoutée à mini-ML sous la forme de l'opérateur de point fixe `fix` (aussi noté Y dans la littérature du λ-calcul) : intuitivement, `fix(fun` $f \to a$) est la fonction f qui vérifie $f = a$. Par exemple, la fonction factorielle s'écrit

```
fix(fun fact → fun n → if n = 0 then 1 else n * fact(n-1))
```

et la fonction `power`$(f)(n)$, qui calcule la composée $f \circ \cdots \circ f$ (n compositions), s'écrit :

```
fix(fun power → fun f → fun n →
      if n = 0 then (fun x → x)
               else (fun x → power(f)(n-1)(f(x))))
```

Plus généralement, les définitions de fonctions récursives en Caml, notées

`let rec` $f\ x = a_1$ `in` a_2

s'expriment en mini-ML par

`let` $f =$ `fix(fun` $x \to a_1$`)` `in` a_2

Exercice 2.1 *Caml permet également des définitions de fonctions mutuellement récursives :*

`let rec` $f\ x = a_1$ `and` $g\ y = a_2$ `in` a_3

Comment les traduire en mini-ML ?

2.3 Sémantique opérationnelle "à grands pas" de mini-ML

2.3.1 Les valeurs

La sémantique opérationnelle que nous donnons ici se présente comme une relation entre expressions a et valeurs v, notée $a \xrightarrow{v} v$ (lire : "l'expression a s'évalue en la valeur v"). Les valeurs v sont décrites par la grammaire suivante :

Valeurs :	$v ::= \texttt{fun}\ x \to a$		valeurs fonctionnelles
	$\mid c$		valeurs constantes
	$\mid op$		primitives non appliquées
	$\mid (v_1, v_2)$		paire de deux valeurs

(Remarquons qu'ici les valeurs sont un sous-ensemble des expressions ; ce n'est pas toujours le cas dans ce style de sémantique.)

2.3.2 Rappels sur les règles d'inférence

Nous allons utiliser des *règles d'inférence* pour définir la relation d'évaluation, ainsi que la plupart des relations utilisées dans ce cours. Une définition par règles d'inférence se compose d'axiomes A et d'implications de la forme

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{P}$$

qui se lisent "si $P_1, \dots, P_n$ sont vraies, alors P est vraie". Une autre lecture de la règle d'inférence ci-dessus est comme l'implication $P_1 \wedge \dots \wedge P_n \Rightarrow P$.

Les règles d'inférence peuvent contenir des variables libres, qui sont implicitement quantifiées universellement en tête de la règle. Par exemple, l'axiome $A(x)$ signifie $\forall x. A(x)$; la règle

$$\frac{P_1(x) \quad P_2(y)}{P(x, y)}$$

signifie $\forall x, y.\ P_1(x) \wedge P_2(y) \Rightarrow P(x, y)$.

Étant donné un ensemble de règles d'inférence portant sur une ou plusieurs relations, on définit ces relations comme les plus petites relations qui satisfont les règles d'inférence. Par exemple, voici des règles sur des prédicats $\texttt{Pair}(n)$ et $\texttt{Impair}(n)$:

$$\texttt{Pair}(0) \qquad \frac{\texttt{Impair}(n)}{\texttt{Pair}(n+1)} \qquad \frac{\texttt{Pair}(n)}{\texttt{Impair}(n+1)}$$

Il faut les lire comme les conditions suivantes portant sur les prédicats `Pair` et `Impair` :

$$\begin{array}{l} \texttt{Pair}(0) \\ \forall n.\ \texttt{Impair}(n) \Rightarrow \texttt{Pair}(n+1) \\ \forall n.\ \texttt{Pair}(n) \Rightarrow \texttt{Impair}(n+1) \end{array}$$

Il y a de nombreux prédicats sur les entiers qui satisfont ces conditions, par exemple $\texttt{Pair}(n)$ et $\texttt{Impair}(n)$ vrais pour tout n. Cependant, les plus petits prédicats (les prédicats vrais les moins

souvent) vérifiant ces conditions sont $\texttt{Pair}(n) = (n \bmod 2 = 0)$ et $\texttt{Impair}(n) = (n \bmod 2 = 1)$. Les règles d'inférence définissent donc `Pair` et `Impair` comme étant ces deux prédicats.

Le plus petit prédicat satisfaisant un ensemble de règles d'inférence existe toujours : c'est la conjonction (infinie) de tous les prédicats satisfaisant les règles. Cependant, cette caractérisation n'est pas très commode à manipuler. Heureusement, une caractérisation beaucoup plus maniable existe, à base de dérivations.

Une *dérivation* (encore appelée *arbre de preuve*) dans un système de règles d'inférence est un arbre portant aux feuilles des instances des axiomes et aux noeuds des conclusions de règles d'inférence dont les hypothèses sont justifiées par les fils du noeud dans l'arbre. La conclusion de la dérivation est le noeud racine de l'arbre. On représente généralement les dérivations par des "empilements" d'instances de règles d'inférence, avec la conclusion de la dérivation en bas. Par exemple, voici une dérivation qui conclut `Impair(3)` dans le système de règles ci-dessus :

$$\dfrac{\dfrac{\dfrac{\texttt{Pair}(0)}{\texttt{Impair}(1)}}{\texttt{Pair}(2)}}{\texttt{Impair}(3)}$$

Les dérivations caractérisent exactement les plus petits prédicats vérifiant un ensemble de règles d'inférence. Dans l'exemple, notons

$$\begin{aligned} P_0(n) &= \text{il existe une dérivation qui conclut } \texttt{Pair}(n) \\ I_0(n) &= \text{il existe une dérivation qui conclut } \texttt{Impair}(n) \end{aligned}$$

On peut montrer que P_0 et I_0 sont les plus petits prédicats qui satisfont les règles d'inférence pour `Pair` et `Impair`.

Exercice 2.2 *Pour démontrer formellement ce résultat, montrer que :*

1. *P_0 et I_0 satisfont les règles d'inférence.*
2. *Pour tous prédicats P et I qui satisfont les règles d'inférence, $P_0(n) \Rightarrow P(n)$ et $I_0(n) \Rightarrow I(n)$.*

L'intérêt des dérivations est qu'il s'agit de structures syntaxiques finies (de simples arbres) sur lesquels on peut raisonner par récurrence : récurrence sur la taille de la dérivation ou sur la structure de la dérivation. Nous verrons des exemples de telles récurrences dans le reste du chapitre.

2.3.3 Les règles d'évaluation

Nous définissons la relation $a \xrightarrow{v} v$ (lire : "l'expression a s'évalue en la valeur v") comme la plus petite relation satisfaisant les axiomes et les règles d'inférence suivants :

$$c \xrightarrow{v} c \ (1) \qquad op \xrightarrow{v} op \ (2) \qquad (\texttt{fun}\ x \to a) \xrightarrow{v} (\texttt{fun}\ x \to a) \ (3)$$

$$\frac{a_1 \xrightarrow{v} (\texttt{fun}\ x \to a) \quad a_2 \xrightarrow{v} v_2 \quad a[x \leftarrow v_2] \xrightarrow{v} v}{a_1\ a_2 \xrightarrow{v} v} \ (4) \qquad \frac{a_1 \xrightarrow{v} v_1 \quad a_2 \xrightarrow{v} v_2}{(a_1, a_2) \xrightarrow{v} (v_1, v_2)} \ (5)$$

$$\frac{a_1 \xrightarrow{v} v_1 \quad a_2[x \leftarrow v_1] \xrightarrow{v} v}{(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) \xrightarrow{v} v} \ (6)$$

On a noté $a[x \leftarrow v]$ l'expression obtenue en substituant chaque occurrence de x libre dans a par v. Ainsi, $(+\ (x,1))[x \leftarrow 2]$ est $+(2,1)$, mais $(\texttt{fun}\ x \to x)[x \leftarrow 2]$ est $(\texttt{fun}\ x \to x)$.

Les règles 1, 2, 3 expriment que les constantes, les opérateurs et les fonctions sont déjà entièrement évalués : il n'y a rien à faire lors de l'évaluation. La règle 4 exprime que pour évaluer une application $a_1\ a_2$, il faut évaluer a_1 et a_2. Si la valeur de a_1 est une fonction $\texttt{fun}\ x \to a$ (règle 4), le résultat de l'application est la valeur de a après substitution du paramètre formel x par l'argument effectif v_2 (la valeur de a_2). L'évaluation des paires (règle 5) est évidente : on évalue les deux sous-expressions et on construit la paire des deux valeurs obtenues. Enfin, pour une construction $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$, la règle 6 exprime qu'il faut d'abord évaluer a_1, puis substituer x par sa valeur dans a_2 et poursuivre avec l'évaluation de a_2.

Pour être complet, il faut ajouter des règles pour chaque opérateur *op* qui nous intéresse, décrivant l'évaluation des applications de cet opérateur. Par exemple, pour `+`, `fst`, `snd` nous avons les règles

$$\frac{a_1 \xrightarrow{v} +\quad a_2 \xrightarrow{v} (n_1, n_2)\quad n_1, n_2 \text{ constantes entières et } n = n_1 + n_2}{a_1\ a_2 \xrightarrow{v} n} \quad (7)$$

$$\frac{a_1 \xrightarrow{v} \texttt{fst}\quad a_2 \xrightarrow{v} (v_1, v_2)}{a_1\ a_2 \xrightarrow{v} v_1} \quad (8) \qquad \frac{a_1 \xrightarrow{v} \texttt{snd}\quad a_2 \xrightarrow{v} (v_1, v_2)}{a_1\ a_2 \xrightarrow{v} v_2} \quad (9)$$

Exemple 1 : nous avons $(\texttt{fun}\ x \to +(x,1))\ 2 \xrightarrow{v} 3$, car la dérivation suivante est valide :

$$\dfrac{(\texttt{fun}\ x \to +(x,1)) \xrightarrow{v} (\texttt{fun}\ x \to +(x,1))\quad 2 \xrightarrow{v} 2\quad \dfrac{+ \xrightarrow{v} +\quad \dfrac{2 \xrightarrow{v} 2\quad 1 \xrightarrow{v} 1}{(2,1) \xrightarrow{v} (2,1)}}{+(2,1) \xrightarrow{v} 3}}{(\texttt{fun}\ x \to +(x,1))\ 2 \xrightarrow{v} 3}$$

Exemple 2 : Soit $a = 1\ 2$. Il n'existe pas de valeur v telle que $a \xrightarrow{v} v$. En effet, une dérivation de $a \xrightarrow{v} v$ devrait se terminer par une des règles 4, 7, 8 ou 9. Mais pour que ces règles s'appliquent, il faudrait que 1 s'évalue en une fonction (pour la règle 4) ou en une opération (pour les autres règles). Cependant, la seule valeur en laquelle 1 peut s'évaluer est 1, qui n'est rien de tout cela. Donc, il n'y a pas de dérivation de $a \xrightarrow{v} v$ pour tout v.

Exemple 3 : Soit $a = (\texttt{fun}\ f \to (f\ f))\ (\texttt{fun}\ f \to (f\ f))$. Il n'existe pas de valeur v telle que $a \xrightarrow{v} v$. En effet, notons $b = (\texttt{fun}\ f \to (f\ f))$. Une dérivation de $a \xrightarrow{v} v$ doit nécessairement se terminer ainsi :

$$\frac{b \xrightarrow{v} b\quad b \xrightarrow{v} b\quad (f\ f)[f \leftarrow b] \xrightarrow{v} v}{b\ b \xrightarrow{v} v}$$

Mais $(f\ f)[f \leftarrow b] = b\ b = a$, donc toute dérivation de $a \xrightarrow{v} v$ doit contenir une sous-dérivation de $a \xrightarrow{v} v$. Nous pouvons donc construire une suite infinie de dérivations de $a \xrightarrow{v} v$ de tailles strictement décroissantes. Ceci n'est bien sûr pas possible car les dérivations sont toutes finies.

2.3.4 Quelques propriétés de la relation d'évaluation

En guise d'entraînement à la preuve par récurrence sur une dérivation, nous allons montrer deux propriétés simples de la sémantique à grands pas : 1- le résultat est bien une valeur, et 2- l'évaluation est déterministe.

Proposition 2.1 (L'évaluation produit des valeurs closes) *Si $a \xrightarrow{v} v$, alors v est une valeur bien formée. De plus, si a est une expression close (sans variables libres), v est également close.*

Démonstration: Nous savons par hypothèse $a \xrightarrow{v} v$ qu'il existe une dérivation D de $a \xrightarrow{v} v$. Nous raisonnons donc par récurrence structurelle sur cette dérivation, et par cas sur la dernière règle d'évaluation utilisée.

Cas règle 1, règle 2, règle 3 : nous avons $v = c$, $v = op$ ou $v = \texttt{fun}\ x \to a$ respectivement, qui sont des valeurs bien formées. De plus, $v = a$, donc v est close si a l'est aussi.

Cas règle 4 : la dérivation d'évaluation est de la forme suivante :

$$\frac{\begin{array}{ccc}(D_1) & (D_2) & (D_3)\\ \vdots & \vdots & \vdots \\ a_1 \xrightarrow{v} \texttt{fun}\ x \to a & a_2 \xrightarrow{v} v_2 & a[x \leftarrow v_2] \xrightarrow{v} v\end{array}}{a_1\ a_2 \xrightarrow{v} v}$$

Par hypothèse de récurrence appliquée à la dérivation D_3, nous obtenons que v est une valeur. De plus, si $a = a_1\ a_2$ est close, a_1 et a_2 le sont aussi. Donc, par hypothèse de récurrence appliquée à D_1 et D_2, les valeurs v_2 et $\texttt{fun}\ x \to a$ sont closes. Il s'ensuit que $a[x \leftarrow v_2]$ est clos, et donc v est close par hypothèse de récurrence appliquée à D_3.

Cas règle 5 : nous avons $a = (a_1, a_2)$, $v = (v_1, v_2)$ et des sous-dérivations d'évaluation $a_1 \xrightarrow{v} v_1$ et $a_2 \xrightarrow{v} v_2$. Comme a est close, a_1 et a_2 le sont aussi. Par hypothèse de récurrence appliquée aux deux sous-dérivations d'évaluation, il vient que v_1 et v_2 sont deux valeurs bien formées et closes. Il en va donc de même pour v.

Cas règle 6 : nous avons $a = \texttt{let}\ x = a_1\ \texttt{in}\ a_2$ et des sous-dérivations d'évaluation $a_1 \xrightarrow{v} v_1$ et $a_2[x \leftarrow v_1] \xrightarrow{v} v$. En appliquant l'hypothèse de récurrence à cette dernière sous-dérivation, il vient v valeur bien formée. De plus, si a est close, a_1 et a_2 le sont aussi. Donc v_1 est close (par hypothèse de récurrence appliquée à la première sous-dérivation) et $a_2[x \leftarrow v_1]$ aussi. L'hypothèse de récurrence pour la seconde sous-dérivation montre donc que v est close.

Cas règles 7, 8, 9. Même raisonnements que dans les cas précédents. Remarquons que si (v_1, v_2) est une valeur bien formée et close, c'est le cas également pour v_1 et v_2. □

Passons maintenant à la propriété de déterminisme : une expression donnée ne peut pas s'évaluer en deux valeurs différentes.

Proposition 2.2 (Déterminisme de l'évaluation) *Si $a \xrightarrow{v} v$ et $a \xrightarrow{v} v'$, alors $v = v'$.*

Démonstration: Nous raisonnons par récurrence structurelle sur les dérivations D de $a \xrightarrow{v} v$ et D' de $a \xrightarrow{v} v'$, puis par cas sur la forme de l'expression a.

Cas a est une constante c. La seule règle d'évaluation qui s'applique à une constante est 1. Donc, les dérivations D et D' sont de la forme $c \xrightarrow{v} c$, et $v_1 = c$ et $v_2 = c$. D'où le résultat $v = v'$.

Cas a est un opérateur *op* ou une abstraction `fun` $x \to a$. Comme le cas précédent.

Cas a est une paire (a_1, a_2). Une seule règle d'évaluation peut s'appliquer à a : la règle 5. Donc, la dérivation D est nécessairement de la forme

$$\dfrac{\begin{array}{cc}(D_1) & (D_2)\\ \vdots & \vdots \\ a_1 \xrightarrow{v} v_1 & a_2 \xrightarrow{v} v_2\end{array}}{(a_1, a_2) \xrightarrow{v} (v_1, v_2)}$$

et de même D' est de la forme

$$\dfrac{\begin{array}{cc}(D'_1) & (D'_2)\\ \vdots & \vdots \\ a_1 \xrightarrow{v} v'_1 & a_2 \xrightarrow{v} v'_2\end{array}}{(a_1, a_2) \xrightarrow{v} (v'_1, v'_2)}$$

Mais D_1 est une sous-dérivation de D, et D'_1 une sous-dérivation de D'. Nous pouvons donc appliquer l'hypothèse de récurrence à l'expression a_1, aux valeurs v_1 et v'_1 et aux dérivations D_1 et D'_1 ; il s'ensuit que $v_1 = v'_1$. Appliquant de même l'hypothèse de récurrence à l'expression a_2, aux valeurs v_2 et v'_2 et aux dérivations D_2 et D'_2, nous obtenons $v_2 = v'_2$. Donc, $v = (v_1, v_2) = (v'_1, v'_2) = v'$, ce qui est le résultat attendu.

Cas a est une application $a_1\ a_2$. Quatre règles d'évaluation peuvent s'appliquer à a : les règles 4, 7, 8 ou 9. Donc, D et D' se terminent nécessairement par l'une de ces règles. Remarquons que ces cinq règles ont des prémisses de la forme $a_1 \xrightarrow{v} v_1$ et $a_2 \xrightarrow{v} v_2$ pour certaines valeurs de v_1 et v_2. Donc, D est nécessairement de la forme

$$\dfrac{\begin{array}{cc}(D_1) & (D_2)\\ \vdots & \vdots \\ a_1 \xrightarrow{v} v_1 & a_2 \xrightarrow{v} v_2\end{array} \quad \cdots}{a_1\ a_2 \xrightarrow{v} v}$$

et D' est aussi de la forme

$$\dfrac{\begin{array}{cc}(D'_1) & (D'_2)\\ \vdots & \vdots \\ a_1 \xrightarrow{v} v'_1 & a_2 \xrightarrow{v} v'_2\end{array} \quad \cdots}{a_1\ a_2 \xrightarrow{v} v'}$$

Comme D_1 est une sous-dérivation de D et D'_1 une sous-dérivation de D', nous pouvons appliquer l'hypothèse de récurrence à D_1 et D'_1. Il vient $v_1 = v'_1$. Procédant de même avec D_2 et D'_2, il vient $v_2 = v'_2$.

Nous pouvons maintenant raisonner par cas sur la forme de v_1, qui peut être une fonction ou un opérateur. Supposons par exemple $v_1 = \texttt{fun}\ x \to a$. Compte tenu de ce que $v_1 = v_1'$ et $v_2 = v_2'$, les dérivations D et D' sont donc de la forme

$$\dfrac{\begin{array}{ccc}(D_1) & (D_2) & (D_3)\\ \vdots & \vdots & \vdots \\ a_1 \xrightarrow{v} \texttt{fun}\ x \to a & a_2 \xrightarrow{v} v_2 & a[x \leftarrow v_2] \xrightarrow{v} v\end{array}}{a_1\ a_2 \xrightarrow{v} v}$$

$$\dfrac{\begin{array}{ccc}(D_1') & (D_2') & (D_3')\\ \vdots & \vdots & \vdots \\ a_1 \xrightarrow{v} \texttt{fun}\ x \to a & a_2 \xrightarrow{v} v_2 & a[x \leftarrow v_2] \xrightarrow{v} v'\end{array}}{a_1\ a_2 \xrightarrow{v} v'}$$

Appliquant une dernière fois l'hypothèse de récurrence aux dérivations D_3 et D_3', il vient $v = v'$ comme attendu.

Si v_1 est un opérateur $+$, `fst`, ou `snd`, nous concluons directement $v = v'$ par examen des règles, sans avoir besoin d'invoquer l'hypothèse de récurrence une troisième fois.

Cas a est $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$. Le résultat découle de l'hypothèse de récurrence par un raisonnement analogue au cas de l'application. □

Exercice 2.3 *Rédiger complètement le dernier cas de la preuve ci-dessus.*

Un corollaire de la proposition 2.2 est que l'évaluation de mini-ML est réellement une fonction partielle *eval* des expressions dans les valeurs : $eval(a)$, si défini, est l'unique valeur v telle que $a \xrightarrow{v} v$.

Nous avons cependant gardé une présentation relationnelle, car elle s'étend plus facilement avec des primitives non-déterministes. Par exemple, donnons-nous une primitive $\texttt{random}(n)$ qui renvoie un nombre aléatoire entre 0 et n. Sa règle d'évaluation est :

$$\dfrac{a_1 \xrightarrow{v} \texttt{random} \qquad a_2 \xrightarrow{v} n \qquad n \text{ entier et } 0 \leq m \leq n}{a_1(a_2) \xrightarrow{v} m}$$

Avec cette règle, on a $\texttt{random}(1) \xrightarrow{v} 0$ et $\texttt{random}(1) \xrightarrow{v} 1$, exprimant que 0 et 1 sont deux valeurs correctes pour cette expression.

2.3.5 Implémentation en Caml d'un interprète pour mini-ML

Nous allons maintenant programmer en Caml un interprète pour mini-ML qui suit à la lettre les règles d'évaluation ci-dessus. C'est une fonction d'évaluation qui prend une expression et renvoie sa valeur si elle existe.

Nous commençons par définir la syntaxe abstraite des expressions mini-ML sous forme d'un type concret de Caml :

```
type expression =
  | Var of string
  | Const of int
  | Op of string
  | Fun of string * expression
  | App of expression * expression
  | Paire of expression * expression
  | Let of string * expression * expression
```

Nous utilisons le type `string` des chaînes de caractères pour représenter aussi bien les variables que les opérateurs.

Nous définissons maintenant une fonction de substitution `subst` a x b qui calcule l'expression $a[x \leftarrow b]$. Tout comme dans le λ-calcul, la substitution générale dans les termes de mini-ML est délicate car il faut éviter les captures de variables lorsque a est une fonction `fun` $x \to a'$ et x est libre dans b. Cependant, la proposition 2.1 montre que le terme b substitué est toujours clos (sans variables libres) lorsqu'on évalue des termes clos. Dans ce cas, la capture de variables est impossible et nous pouvons définir une substitution très simple, sans renommage de variables.

```
let rec subst a x b =
  match a with
  | Var v -> if v = x then b else a
  | Const n -> a
  | Op o -> a
  | Fun(v, a1) -> if v = x then a else Fun(v, subst a1 x b)
  | App(a1, a2) -> App(subst a1 x b, subst a2 x b)
  | Paire(a1, a2) -> Paire(subst a1 x b, subst a2 x b)
  | Let(v, a1, a2) -> Let(v, subst a1 x b, if v = x then a2 else subst a2 x b)
```

La fonction d'évaluation procède par cas sur l'expression, ce qui détermine une ou plusieurs règles d'évaluation qui peuvent s'appliquer, et appels récursifs pour effectuer les évaluations figurant en prémisses des règles :

```
let rec valeur a =
  match a with
  | Const c ->                (* règle 1 *)
      Const c
  | Op o ->                   (* règle 2 *)
      Op o
  | Fun(x, a) ->              (* règle 3 *)
      Fun(x, a)
  | App(a1, a2) ->            (* règles 4, 7, 8, 9 ou 10 *)
      begin match valeur a1 with
      | Fun(x, a) ->          (* règle 4 *)
          valeur (subst a x (valeur a2))
      | Op "+" ->             (* règle 7 *)
          let (Paire(Const n1, Const n2)) = valeur a2 in Const(n1 + n2)
      | Op "fst" ->           (* règle 8 *)
```

```
          let (Paire(v1, v2)) = valeur a2 in v1
      | Op "snd" ->           (* règle 9 *)
          let (Paire(v1, v2)) = valeur a2 in v2
      end
  | Paire(a1, a2) ->          (* règle 5 *)
      Paire(valeur a1, valeur a2)
  | Let(x, a1, a2) ->         (* règle 6 *)
      valeur (subst a2 x (valeur a1))
```

Il ne reste plus qu'à essayer :

```
# valeur (App(Op "+", Paire(Const 1, Const 2))) ; ;
- : expression = Const 3

# valeur (Paire(App(Op "+", Paire(Const 1, Const 2)),
                App(Op "+", Paire(Const 3, Const 4)))) ; ;
- : expression = Paire (Const 3, Const 7)

# valeur (App (Fun ("x", App(Op "+", Paire(Var "x", Const 1))), Const 2)) ; ;
- : expression = Const 3

# valeur (App (Const 1, Const 2)) ; ;
Exception : Match_failure ("miniML.ml", 32, 6).

# let b = Fun("x", App(Var "x", Var "x")) in valeur (App(b, b)) ; ;
Interrupted.
```

Dans le dernier exemple (correspondant à l'exemple 3), l'utilisateur a dû interrompre l'exécution car cette dernière ne termine pas. L'avant-dernier exemple (correspondant à l'exemple 2) montre un programme mal formé dont l'évaluation échoue sur une erreur, signalée comme une exception par Caml.

2.4 Sémantique opérationnelle "à petits pas" de mini-ML

Comme nous l'avons vu sur des exemples, la sémantique à grands pas ne distingue pas entre des expressions dont le calcul "plante" à cause d'un sous-terme absurde (comme 1 2) et les expressions dont le calcul "boucle", c'est-à-dire ne termine pas (comme $b\ b$ avec $b = (\texttt{fun}\ x \to x\ x)$). Dans les deux cas, il n'existe pas de valeur en laquelle l'expression s'évalue. Il y a cependant des circonstances où il serait utile de distinguer ces deux comportements dans la sémantique formelle. (C'est en particulier le cas pour prouver la sûreté d'un système de types, voir la section 3.5 et le cours 2-4.)

La sémantique opérationnelle à petits pas, aussi appelée sémantique à réductions, est une autre forme de sémantique opérationnelle qui expose plus finement les étapes intermédiaires du calcul. Elle se présente sous la forme d'une relation $a \to a'$, où a' est l'expression obtenue à partir de a par une seule étape de calcul. Pour évaluer complètement l'expression, il faut bien sûr enchaîner plusieurs étapes

$$a \to a_1 \to a_2 \to \ldots \to v$$

jusqu'à ce qu'on atteigne une valeur (une expression complètement évaluée). Ainsi, toutes les étapes intermédiaires du calcul – la progression du programme initial vers son résultat final – sont observables. De plus, on peut facilement distinguer les programmes qui "bouclent" de ceux qui "plantent" : les premiers correspondent à des suites infinies de réductions

$$a \to a_1 \to a_2 \to a_3 \dots$$

et les seconds à des suites finies qui arrivent sur une expression non réductible qui n'est pas une valeur :

$$a \to a_1 \to a_2 \to \dots \to a_n \text{ et } a_n \text{ irréductible, et } a_n \text{ n'est pas une valeur}$$

La notion de réduction que nous allons utiliser est proche, dans l'esprit, de la β-réduction dans le λ-calcul. Elle s'en écarte sur deux points :

- Mini-ML fournit des constructions supplémentaires par-rapport au λ-calcul : liaison `let`, constantes, opérateurs. Ces constructions nécessitent des règles de réduction supplémentaires.
- Notre relation de réduction impose une stratégie d'évaluation fixée : appel par valeur avec évaluation des sous-expressions de gauche à droite. La β-réduction du λ-calcul peut s'appliquer à n'importe quel sous-terme d'un λ-terme. Le choix des redex successivement réduits peut faire la différence entre une évaluation qui termine et une qui ne termine pas, ou entre une évaluation implémentable efficacement et une qui ne l'est pas. Pour définir complètement le comportement des programmes d'un langage, il est préférable de définir également la stratégie d'évaluation.

2.4.1 Définition de la relation de réduction

Pour définir la relation "se réduit en une étape" $\to$, on commence par se donner des axiomes pour la relation $\xrightarrow{\varepsilon}$ "se réduit en une étape en tête du terme". Les deux axiomes de base sont la β réduction pour les fonctions et pour le `let` :

$$\begin{aligned} (\texttt{fun}\ x \to a)\ v &\xrightarrow{\varepsilon} a\{x \leftarrow v\} \qquad (\beta_{fun}) \\ (\texttt{let}\ x = v\ \texttt{in}\ a) &\xrightarrow{\varepsilon} a\{x \leftarrow v\} \qquad (\beta_{let}) \end{aligned}$$

Notons que ces deux règles ne s'appliquent que si le substitué est une valeur v : cela reflète la stratégie "appel par valeur" que nous désirons.

On se donne aussi des axiomes pour les opérateurs. Ces axiomes s'appellent aussi des δ-règles et dépendent bien sûr des opérateurs considérés. Voici les δ-règles pour `+`, `fst`, `snd`, `ifzero`, et `fix` :

$$\begin{aligned} +\ (n_1, n_2) &\xrightarrow{\varepsilon} n \text{ si } n_1, n_2 \text{ entiers et } n = n_1 + n_2 \qquad (\delta_+) \\ \texttt{fst}\ (v_1, v_2) &\xrightarrow{\varepsilon} v_1 \qquad (\delta_{fst}) \\ \texttt{snd}\ (v_1, v_2) &\xrightarrow{\varepsilon} v_2 \qquad (\delta_{snd}) \\ \texttt{fix}\ (\texttt{fun}\ x \to a) &\xrightarrow{\varepsilon} a\{x \leftarrow \texttt{fix}\ (\texttt{fun}\ x \to a)\} \qquad (\delta_{fix}) \\ \texttt{ifzero 0 then}\ a_1\ \texttt{else}\ a_2 &\xrightarrow{\varepsilon} a_1 \qquad (\delta_{if}) \\ \texttt{ifzero}\ n\ \texttt{then}\ a_1\ \texttt{else}\ a_2 &\xrightarrow{\varepsilon} a_2 \text{ si } n \text{ entier, } n \neq 0 \qquad (\delta_{if'}) \end{aligned}$$

Bien sûr, on ne réduit pas toujours en tête de l'expression. Considérons par exemple

$$a = (\texttt{let}\ x = +(1,2)\ \texttt{in}\ +(x,3))$$

Aucun des axiomes ci-dessus ne s'applique à a. Pour évaluer a, il est clair qu'il faut commencer par réduire "en profondeur" la sous-expression $+(1,2)$. Cette notion de réduction en profondeur est exprimée par la règle d'inférence suivante :

$$\frac{a \xrightarrow{\varepsilon} a'}{E(a) \to E(a')}\ \text{(contexte)}$$

Dans cette règle, E est un *contexte d'évaluation*. Les contextes d'évaluation sont définis par la syntaxe abstraite suivante :

Contextes d'évaluation :

$E ::= [\,]$	évaluation en tête
$\mid E\ a$	évaluation à gauche d'une application
$\mid v\ E$	évaluation à droite d'une application
$\mid \texttt{let}\ x = E\ \texttt{in}\ a$	évaluation à gauche d'un `let`
$\mid (E, a)$	évaluation à gauche d'une paire
$\mid (v, E)$	évaluation à droite d'une paire

Rappels sur les contextes : Un contexte est une expression avec un "trou", noté $[\,]$. Par exemple, $+([\,],2)$. L'opération de base sur un contexte C est l'application $C(a)$ à une expression a. $C(a)$ est l'expression dénotée par C dans laquelle le "trou" $[\,]$ est remplacé par a. Par exemple, $+([\,],2)$ appliqué à 1 est l'expression $+(1,2)$.

Les contextes d'évaluation E ne sont pas n'importe quelle expression avec un trou. Par exemple, $(+(1,2),[\,])$ n'est pas un contexte d'évaluation, car le membre gauche de la paire n'est pas une valeur. L'idée est de forcer un certain ordre d'évaluation en restreignant les contextes. La syntaxe des contextes ci-dessus force une évaluation en appel par valeur et de gauche à droite (on évalue la sous-expression gauche d'une application, d'une paire ou d'un `let` avant la sous-expression droite).

Exemple 1 Considérons l'expression $a = (+(1,2),+(3,4))$. Il n'y a qu'une manière de l'écrire sous la forme $a = E_1(a_1)$ afin d'appliquer la règle (contexte) : il faut prendre $E_1 = ([\,],+(3,4))$ et $a_1 = +(1,2)$. Comme $a_1 \xrightarrow{\varepsilon} 3$, on obtient $a \to E_1(3) = (3,+(3,4))$. Cette dernière expression peut alors s'écrire sous la forme $E_2(a_2)$ avec $E_2 = (3,[\,])$ et $a_2 = +(3,4)$. Appliquant la règle (contexte), on obtient le résultat $E_2(7) = (3,7)$. On a donc obtenu la séquence de réductions suivante :

$$(+(1,2),+(3,4)) \to (3,+(3,4)) \to (3,7)$$

qui nous emmène de a vers la valeur $(3,7)$ en évaluant toujours la sous-expression gauche en premier. Si nous voulons commencer par évaluer la sous-expression droite $+(3,4)$, il faudrait écrire $a = E_3(a_3)$ avec $E_3(+(1,2),[\,])$ et $a_3 = +(3,4)$, mais cela n'est pas possible car E_3 n'est pas un contexte d'évaluation.

Exemple 2 Considérons l'expression $a = 1\ 2$. Aucune règle de réduction ne s'applique. On dit que a est en forme normale.

Exemple 3 Considérons l'expression $a = b\ b$ avec $b = \texttt{fun}\ x \to x\ x$. Puisque b est une valeur, nous avons $a \to (x\ x)[x \leftarrow b] = b\ b = a$. (Réduction en tête par la règle β_{fun}). Il y a donc une suite de réductions infinie $a \to a \to a \to \ldots$

Remarque On peut facilement changer l'ordre d'évaluation en modifiant uniquement la grammaire des contextes E. Par exemple, une évaluation de droite à gauche (comme implémenté par Caml) s'obtient en prenant :

Contextes d'évaluation (droite-gauche) :

$E ::=$	$[\,]$	évaluation en tête
	$\mid a\ E$	évaluation à droite d'une application
	$\mid E\ v$	évaluation à gauche d'une application
	$\mid \texttt{let}\ x = E\ \texttt{in}\ a$	évaluation à gauche d'un `let`
	$\mid (a, E)$	évaluation à droite d'une paire
	$\mid (E, v)$	évaluation à gauche d'une paire

Réduction multiple et formes normales On définit la relation $\xrightarrow{*}$ (lire : "se réduit en zéro, une ou plusieurs étapes") comme la fermeture réflexive et transitive de $\to$. C'est-à-dire, $a \xrightarrow{*} a'$ ssi $a = a'$ ou il existe $a_1, \ldots, a_n$ tels que $a \to a_1 \to \ldots \to a_n \to a'$.

On dit que a est en *forme normale* si $a \not\to$, c'est-à-dire s'il n'existe pas d'expression a' telle que $a \to a'$. Remarquons que les valeurs v sont en forme normale. Il y a aussi d'autres expressions qui sont en forme normale et qui ne sont pas des valeurs, comme p.ex. $1\ 2$. Nous caractérisons les expressions erronées (celles qui "plantent") comme les formes normales qui ne sont pas des valeurs. L'exemple typique est $1\ 2$.

2.4.2 Lien entre sémantique "grands pas" et "petits pas"

Nous allons montrer que les deux sémantiques que nous avons définies pour mini-ML, la "grands pas" et la "petits pas", sont équivalentes sur les expressions dont l'évaluation termine sur une valeur : pour toute expression a et toute valeur v, $a \xrightarrow{v} v$ si et seulement si $a \xrightarrow{*} v$. Un lemme technique pour commencer :

Proposition 2.3 (Passage au contexte des réductions) *Supposons $a \to a'$. Alors, pour toute expression b et toute valeur v,*

1. $a\ b \to a'\ b$
2. $v\ a \to v\ a'$
3. $\texttt{let}\ x = a\ \texttt{in}\ b \to \texttt{let}\ x = a'\ \texttt{in}\ b$

Démonstration: Par la règle (contexte), il existe un contexte E et des termes r, r' tels que

$$a = E(r) \qquad a' = E(r') \qquad r \xrightarrow{\varepsilon} r'$$

Considérons les contextes

$$E_1 = E\ b \qquad E_2 = v\ E \qquad E_3 = (\texttt{let}\ x = E\ \texttt{in}\ b)$$

Ce sont des contextes bien formés. Les réductions suivantes sont donc valides :

$$\frac{r \xrightarrow{\varepsilon} r'}{a\ b = E_1(r) \to E_1(r') = a'\ b} \qquad \frac{r \xrightarrow{\varepsilon} r'}{v\ a = E_2(r) \to E_2(r') = v\ a'}$$

$$\frac{r \xrightarrow{\varepsilon} r'}{\mathit{let}\ \mathtt{x} = \mathtt{a}\ \mathtt{in}\ \mathtt{b} = \mathtt{E_3(r)} \to \mathtt{E_3(r')} = \mathtt{let\ x = a'\ in\ b}}$$

C'est le résultat annoncé. □

Proposition 2.4 ("Grands pas" implique "petits pas") *Si $a \xrightarrow{v} v$, alors $a \xrightarrow{*} v$.*

Démonstration: Par récurrence sur la dérivation de l'évaluation $a \xrightarrow{v} v$ et par cas sur la dernière règle utilisée. Si c'est un des axiomes 1, 2, 3, le résultat est immédiat car $v = a$. Si c'est la règle 4, on a :

$$\frac{a \xrightarrow{v} (\texttt{fun}\ x \to c) \qquad b \xrightarrow{v} v_2 \qquad c[x \leftarrow v_2] \xrightarrow{v} v}{a\ b \xrightarrow{v} v}$$

Par hypothèse de récurrence appliquée aux trois sous-dérivations des prémisses, nous savons qu'il existe des suites de réductions de la forme

$$\begin{aligned} &a \to a_1 \to \ldots \to a_n \to v_1 = (\texttt{fun}\ x \to c) \\ &b \to b_1 \to \ldots \to b_p \to v_2 \\ &c[x \leftarrow v_2] \to c_1 \to \ldots \to c_q \to v \end{aligned}$$

En appliquant la proposition 2.3 aux réductions des deux premières suites, nous obtenons

$$\begin{aligned} &a\ b \to a_1\ b \to \ldots \to a_n\ b \to v_1\ b \\ &v_1\ b \to v_1\ b_1 \to \ldots \to v_1\ b_p \to v_1\ v_2 \end{aligned}$$

En enchaînant toutes ces réductions et en glissant au milieu une étape β_{fun}, il vient

$$a\ b \to \ldots \to v_1\ b \to \ldots \to v_1\ v_2 = (\texttt{fun}\ x \to c)\ v_2 \to c[x \leftarrow v_2] \to \ldots \to v$$

D'où le résultat attendu $a\ b \xrightarrow{*} v$.

Le raisonnement est le même pour les autres règles d'évaluation (5, 6, 7, 8, 9). □

Pour l'implication inverse ("petits pas" implique "grands pas"), nous avons besoin de deux lemmes sur la relation d'évaluation $\xrightarrow{v}$.

Proposition 2.5 (Les valeurs sont déjà évaluées) *$v \xrightarrow{v} v$ pour toute valeur v.*

Démonstration: Immédiat par récurrence structurelle sur la valeur v. □

Proposition 2.6 (Réduction et évaluation) *Si $a \to a'$ et $a' \xrightarrow{v} v$, alors $a \xrightarrow{v} v$.*

Démonstration: On montre d'abord le résultat pour une réduction de tête $a \xrightarrow{\varepsilon} a'$, en examinant les axiomes de réduction. Prenons comme exemple β_{fun} : on a $a = (\texttt{fun}\ x \to a_1)\ v_2$ et $a' = a_1[x \leftarrow v_2]$. On peut construire la dérivation suivante :

$$\frac{(\texttt{fun}\ x \to a_1) \xrightarrow{v} (\texttt{fun}\ x \to a_1) \qquad v_2 \xrightarrow{v} v_2 \qquad a_1[x \leftarrow v_2] \xrightarrow{v} v}{(\texttt{fun}\ x \to a_1)\ v_2 \xrightarrow{v} v}$$

(La prémisse de gauche est l'axiome 3 ; celle du milieu vient du lemme 2.5 ; celle de droite de l'hypothèse $a_1[x \leftarrow v_2] \xrightarrow{v} v$.) On a donc bien montré $a \xrightarrow{v} v$. La preuve pour les autres axiomes de réduction en tête est similaire.

Pour finir la preuve, il faut montrer que le résultat passe au contexte : si $a \xrightarrow{\varepsilon} a'$ et $E(a') \xrightarrow{v} v$, alors $E(a) \xrightarrow{v} v$. Cela se fait par récurrence structurelle sur E. Le cas de base $E = [\,]$ est le résultat précédent sur les réductions de tête. Faisons le cas $E = F\ a_2$ par exemple. Supposons donc $E(a') \xrightarrow{v} v$. Comme $E(a') = F(a')\ a_2$, nous avons donc une dérivation de la forme

$$\frac{F(a') \xrightarrow{v} \texttt{fun}\ x \to a_3 \qquad a_2 \xrightarrow{v} v_2 \qquad a_3\{x \leftarrow v_2\} \xrightarrow{v} v}{F(a')\ a_2 \xrightarrow{v} v}$$

Par hypothèse de récurrence, on a $F(a) \xrightarrow{v} \texttt{fun}\ x \to a_3$. On peut donc construire la dérivation

$$\frac{F(a) \xrightarrow{v} \texttt{fun}\ x \to a_3 \qquad a_2 \xrightarrow{v} v_2 \qquad a_3\{x \leftarrow v_2\} \xrightarrow{v} v}{F(a)\ a_2 \xrightarrow{v} v}$$

qui conclut $E(a) \xrightarrow{v} v$ comme attendu. La preuve pour les autres formes de contextes est similaire. □

Proposition 2.7 ("Petits pas" implique "grands pas") *Si $a \xrightarrow{*} v$, alors $a \xrightarrow{v} v$.*

Démonstration: Supposons $a \to a_1 \to \ldots a_n \to v$. Nous avons $v \xrightarrow{v} v$ par le lemme 2.5, puis $a_n \xrightarrow{v} v$ par le lemme 2.6, puis $a_{n-1} \xrightarrow{v} v$ toujours par le lemme 2.6, et ainsi de suite jusqu'à $a \xrightarrow{v} v$ par applications répétées du lemme 2.6. □

2.4.3 Implémentation en Caml d'un réducteur

Nous allons programmer en Caml la sémantique à petits pas de mini-ML, c'est-à-dire une fonction qui prend un terme a et renvoie a' tel que $a \to a'$ ou signale que a est en forme normale.

Commençons par une fonction qui teste si une expression est une valeur :

```
let rec est_une_valeur a =
  match a with
    Var v -> false
  | Const n -> true
```

```
    | Op o -> true
    | Fun(x, a1) -> true
    | App(a1, a2) -> false
    | Paire(a1, a2) -> est_une_valeur a1 && est_une_valeur a2
    | Let(v, a1, a2) -> false
```

Vient ensuite une fonction qui effectue une étape de réduction en tête (relation $\xrightarrow{\varepsilon}$), ou lève l'exception `Forme_normale` si aucune réduction en tête n'est possible.

```
exception Forme_normale

let réduction_tête a =
  match a with
    App(Fun(x, a1), a2) when est_une_valeur a2 -> (* beta_fun *)
      subst a1 x a2
  | Let(x, a1, a2) when est_une_valeur a1 -> (* beta_let *)
      subst a2 x a1
  | App(Op "+", Paire(Const n1, Const n2)) -> (* delta_plus *)
      Const(n1 + n2)
  | App(Op "-", Paire(Const n1, Const n2)) -> (* delta_moins *)
      Const(n1 - n2)
  | App(Op "*", Paire(Const n1, Const n2)) -> (* delta_fois *)
      Const(n1 * n2)
  | App(Op "fst", Paire(a1, a2))
    when est_une_valeur a1 && est_une_valeur a2 -> (* delta_fst *)
      a1
  | App(Op "snd", Paire(a1, a2))
    when est_une_valeur a1 && est_une_valeur a2 -> (* delta_snd *)
      a2
  | App(Op "fix", Fun(x, a2)) ->        (* delta_fix *)
      subst a2 x a
  | App(Op "ifzero", Paire(Const 0, Paire(a1, a2))) -> (* delta_if *)
      a1
  | App(Op "ifzero", Paire(Const n, Paire(a1, a2))) -> (* delta_if' *)
      a2
  | _ ->
      raise Forme_normale
```

Pour programmer la réduction en profondeur d'un terme a (relation $\rightarrow$), il faut savoir décomposer a en un contexte E et un terme r (le redex) tels que $a = E(r)$ et r se réduit en tête. C'est la partie la plus difficile de l'exercice.

Nous commençons par définir la représentation Caml des contextes. Une représentation élégante du contexte E est sous forme d'une fonction Caml `gamma` des expressions dans les expressions, telle que `gamma` appliquée à une expression a renvoie l'expression $E(a)$. Nous définissons les opérations de construction de tels contextes comme suit.

```
type contexte = expression -> expression
```

```
let trou = fun (a : expression) -> a    (* le contexte réduit à [] *)

let app_gauche gamma a2 =               (* le contexte (gamma a2) *)
  function (a : expression) -> App(gamma a, a2)
let app_droite v1 gamma =               (* le contexte (v1 gamma) *)
  function (a : expression) -> App(v1, gamma a)
let paire_gauche gamma a2 =             (* le contexte (gamma, a2) *)
  function (a : expression) -> Paire(gamma a, a2)
let paire_droite v1 gamma =             (* le contexte (v1, gamma) *)
  function (a : expression) -> Paire(v1, gamma a)
let let_gauche x gamma a2 =             (* le contexte (let x = gamma in a2) *)
  function (a : expression) -> Let(x, gamma a, a2)
```

La fonction `décompose` prend une expression a et renvoie un contexte E et un terme r tels que $a = E(r)$ et r se réduit en tête si a se réduit. Il y a trois familles de cas :

- a est déjà en forme normale : on lève l'exception `Forme_normale`.
- a se réduit en tête : on renvoie la paire $(\texttt{trou}, a)$.
- Dans les autres cas : on détermine dans quelle sous-expression il faut "creuser" et on récurse sur cette sous-expression.

```
let rec décompose a =
  match a with
  (* Déjà en forme normale *)
  | Var _ | Const _ | Op _ | Fun(_, _) ->
      raise Forme_normale
  (* Ici on reconnaît les cas où l'on peut réduire en tête.
     On renvoie alors le contexte trivial [] et l'expression a elle-même. *)
  | App(Fun(x, a1), a2) when est_une_valeur a2 ->
      (trou, a)
  | Let(x, a1, a2) when est_une_valeur a1 ->
      (trou, a)
  | App(Op("+" | "-" | "*"), Paire(Const n1, Const n2)) ->
      (trou, a)
  | App(Op("fst" | "snd"), Paire(a1, a2))
    when est_une_valeur a1 && est_une_valeur a2 ->
      (trou, a)
  | App(Op "fix", Fun(x, a2)) ->
      (trou, a)
  | App(Op "ifzero", Paire(Const n, Paire(a1, a2))) ->
      (trou, a)
  (* Maintenant, on voit si l'on peut réduire dans les sous-expressions *)
  | App(a1, a2) ->
      if est_une_valeur a1 then begin
        (* a1 est déjà évaluée, il faut creuser dans a2 *)
        let (gamma, rd) = décompose a2 in
```

```
      (app_droite a1 gamma, rd)
    end else begin
      (* a1 n'est pas encore évalué, c'est là qu'il faut creuser *)
      let (gamma, rd) = décompose a1 in
      (app_gauche gamma a2, rd)
    end
| Paire(a1, a2) ->
    if est_une_valeur a1 then
      (* a1 est déjà évaluée, il faut creuser dans a2 *)
      let (gamma, rd) = décompose a2 in
      (paire_droite a1 gamma, rd)
    else begin
      (* a1 n'est pas encore évalué, c'est là qu'il faut creuser *)
      let (gamma, rd) = décompose a1 in
      (paire_gauche gamma a2, rd)
    end
| Let(x, a1, a2) ->
    (* On sait que a1 n'est pas une valeur, sinon le cas beta-let ci-dessus
       s'appliquerait.  On va donc creuser dans a1. *)
    let (gamma, rd) = décompose a1 in
    (let_gauche x gamma a2, rd)
```

Nous pouvons enfin définir la fonction de réduction : `réduction` a renvoie ou bien `None` si a est en forme normale, ou bien `Some` a' si $a \to a'$.

```
let réduction a =
  try
    let (gamma, rd) = décompose a in Some(gamma(réduction_tête rd))
  with Forme_normale ->
    None
```

On en déduit la fonction qui réduit de manière répétée jusqu'à obtention d'une forme normale :

```
let rec forme_normale a =
  match réduction a with None -> a | Some a' -> forme_normale a'
```

Il est temps de tester :

```
# forme_normale (App(Op "+", Paire(Const 1, Const 2)));;
- : expression = Const 3

# forme_normale (Paire(App(Op "+", Paire(Const 1, Const 2)),
                       App(Op "+", Paire(Const 3, Const 4))));;
 - : expression = Paire (Const 3, Const 7)

# forme_normale (App (Fun ("x", App(Op "+", Paire(Var "x", Const 1))), Const 2));;
- : expression = Const 3
```

```
# forme_normale (App (Const 1, Const 2)) ; ;
- : expression = App (Const 1, Const 2)

# let b = Fun("x", App(Var "x", Var "x")) in forme_normale (App(b, b)) ; ;
Interrupted.
```

Le dernier exemple ne termine pas (réduction infinie).

Chapitre 3

Systèmes de types

3.1 Généralités sur le typage statique

Le but du typage statique est de détecter et de rejeter à la compilation un certain nombre de programmes absurdes, comme 1 2 ou $(\texttt{fun}\ x \to x) + 1$. Cela passe par l'attribution d'un *type* à chaque sous-expression du programme (p.ex. `int` pour une expression entière, $\texttt{int} \to \texttt{int}$ pour une fonction des entiers dans les entiers, etc.) et la vérification de la cohérence de ces types. Trois exigences pèsent sur les systèmes de types :

- Pour être effectif, un système de types doit être décidable : il ne s'agit pas d'exécuter entièrement le programme lors de la compilation.
- Pour être correct, un système de types doit rejeter les programmes absurdes comme 1 2, ou plus généralement les programmes dont l'évaluation bloque en atteignant une telle configuration absurde. On parle de propriété de *sûreté du typage* par-rapport à la sémantique du langage. (Voir section 3.5.)
- Enfin, pour être utilisable en pratique, un système de types doit être *expressif* et accepter une proportion suffisamment élevée de programmes non-absurdes pour ne pas trop gêner le programmeur. Bien sûr, déterminer si un programme est absurde ou non est indécidable (il n'y a pas d'autre moyen que de l'exécuter, risquant ainsi la non-terminaison), et donc un système de types statique va forcément rejeter certains programmes non-absurdes.

Ce qui est décrit dans le paragraphe précédent est le point de vue "langages de programmation" sur les systèmes de types. Il est développé dans le cours 2-4 (Langages de programmation). D'autres points de vue existent :

- Les systèmes de types possèdent également une interprétation en logique, appelée "théorie des types". Dans cette approche, les types sont vus comme des propositions logiques et les termes de type T comme des preuves de la proposition T (isomorphisme de Curry-Howard). Cette approche est explorée dans le cours 2-7 (Fondements des systèmes de preuves et Assistants de preuves).
- Les systèmes de types sont un cas particulier d'analyse statique de programme : l'établissement automatique de propriétés de sûreté sur des programmes sans les exécuter. Un cadre plus général pour l'analyse statique est l'interprétation abstraite (cours 2-6).
- Enfin, les systèmes de types ont également une grande importance pour la compilation efficace des langages de programmation. Les garanties établies par typage statique peuvent être exploitées de nombreuses manières pour produire du code machine plus efficace.

3.2 Typage monomorphe

Nous commençons par définir un système de type très simple pour mini-ML, caractérisé par le fait que chaque expression et chaque variable reçoit un type unique. On appelle cela le "typage *monomorphe*", ou encore le "λ-calcul simplement typé".

3.2.1 L'algèbre de types

Les expressions de types, ou plus simplement les types (notés τ), sont décrits par la syntaxe abstraite suivante :

Types :	$\tau ::= T$	type de base (`int`, `bool`, etc)
	$\mid \tau_1 \to \tau_2$	type des fonctions de τ_1 dans τ_2
	$\mid \tau_1 \times \tau_2$	type des paires de τ_1 et τ_2

3.2.2 Les règles de typage

La relation de typage porte sur des triplets (Γ, a, τ), où a est une expression, τ un type, et Γ un *environnement de typage* qui associe des types $\Gamma(x)$ aux identificateurs x libres dans a. La relation de typage est notée $\Gamma \vdash a : \tau$ et se lit "dans l'environnement Γ, l'expression a a le type τ", ou encore "sous les hypothèses de typage sur les variables exprimées dans Γ, l'expression a a le type τ".

$$\Gamma \vdash x : \Gamma(x) \ \text{(var)} \qquad \Gamma \vdash c : TC(c) \ \text{(const)} \qquad \Gamma \vdash op : TC(op) \ \text{(op)}$$

$$\frac{\Gamma + \{x : \tau_1\} \vdash a : \tau_2}{\Gamma \vdash (\texttt{fun}\ x \to a) : \tau_1 \to \tau_2}\ \text{(fun)} \qquad \frac{\Gamma \vdash a_1 : \tau' \to \tau \quad \Gamma \vdash a_2 : \tau'}{\Gamma \vdash a_1\ a_2 : \tau}\ \text{(app)}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash (a_1, a_2) : \tau_1 \times \tau_2}\ \text{(paire)} \qquad \frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma + \{x : \tau_1\} \vdash a_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2}\ \text{(let)}$$

Pour les règles (const) et (op), on se donne une fonction TC qui associe un type à chaque constructeur et opérateur ; par exemple, $TC(0) = TC(1) = \texttt{int}$ et $TC(+) = \texttt{int} \times \texttt{int} \to \texttt{int}$.

Dans la règle (var), $\Gamma(x)$ est le type qui est associé à x dans l'environnement Γ. Dans les règles (fun) et (let), $\Gamma + \{x : \tau\}$ est l'environnement qui associe τ à x et qui est identique à Γ sur toute variable autre que x.

3.2.3 Exemples de typages et de non-typages

Voici une dérivation de typage dans le système ci-dessus :

$$\dfrac{\dfrac{\dfrac{\dfrac{\{x : \texttt{int}\} \vdash x : \texttt{int} \quad \{x : \texttt{int}\} \vdash 1 : \texttt{int}}{\{x : \texttt{int}\} \vdash (x, 1) : \texttt{int} \times \texttt{int}} \quad \{x : \texttt{int}\} \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}}{\{x : \texttt{int}\} \vdash +(x, 1) : \texttt{int}}}{\emptyset \vdash \texttt{fun}\ x \to +(x, 1) : \texttt{int} \to \texttt{int}} \quad \dfrac{\dfrac{\{f : \texttt{int} \to \texttt{int}\} \vdash f : \texttt{int} \to \texttt{int}}{\{f : \texttt{int} \to \texttt{int}\} \vdash 2 : \texttt{int}}}{\{f : \texttt{int} \to \texttt{int}\} \vdash f\ 2 : \texttt{int}}}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to +(x, 1)\ \texttt{in}\ f\ 2 : \texttt{int}}$$

Autres exemples de typages que l'on peut dériver :

$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{int} \to \texttt{int}$$
$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{bool} \to \texttt{bool}$$

Exemples de typages que l'on ne peut pas dériver :

$$\emptyset \vdash \texttt{fun}\ x \to +(x, 1) : \texttt{int}$$
$$\emptyset \vdash \texttt{fun}\ x \to +(x, 1) : \texttt{bool} \to \texttt{int}$$

Exemples d'expressions que l'on ne peut pas typer (il n'existe pas de Γ et de τ tels que $\Gamma \vdash a : \tau$) :

$$1\ 2$$
$$\texttt{fun}\ x \to x\ x$$
$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$$

Détaillons pourquoi ces trois dernières expressions ne sont pas typables. Pour 1 2, une dérivation de typage serait nécessairement de la forme suivante :

$$\dfrac{\Gamma \vdash 1 : \tau' \to \tau \quad \Gamma \vdash 2 : \tau'}{\Gamma \vdash 1\ 2 : \tau}$$

mais le seul type que l'on peut attribuer aux constantes 1 et 2 est $TC(1) = TC(2) = \texttt{int}$, qui n'est pas de la forme $\tau' \to \tau$.

Pour $\texttt{fun}\ x \to x\ x$, la dérivation serait de la forme :

$$\dfrac{\dfrac{\Gamma + \{x : \tau_1\} \vdash x : \tau_3 \to \tau_2 \quad \Gamma + \{x : \tau_1\} \vdash x : \tau_3}{\Gamma + \{x : \tau_1\} \vdash x\ x : \tau_2}}{\Gamma \vdash (\texttt{fun}\ x \to x\ x) : \tau_1 \to \tau_2}$$

mais la seule règle permettant de typer x est la règle (var), qui impose $\tau_3 \to \tau_2 = \tau_1$ et $\tau_3 = \tau_1$. Autrement dit, τ_1 devrait être de la forme $\tau_1 \to \ldots$, et ce n'est pas possible (les types sont des termes finis).

Enfin, pour $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$, le typage de la partie gauche du `let` donne un type $\tau \to \tau$ pour $\texttt{fun}\ x \to x$ (pour un certain τ choisi arbitrairement). La partie droite du `let` est donc typée sous l'hypothèse $\texttt{f} : \tau \to \tau$. Or, pour que $f\ 1$ soit bien typé, il faudrait que $\tau = \texttt{int}$; et pour que $f\ \texttt{true}$ soit bien typé, il faudrait $\tau = \texttt{bool}$. Il est impossible de satisfaire ces deux contraintes simultanément.

3.3 Typage polymorphe : système F

La faiblesse du typage monomorphe est qu'un identificateur ne peut avoir qu'un seul type, même s'il est lié à une fonction qui possède naturellement plusieurs types. Par exemple, la fonction identité $\texttt{fun}\ x \to x$ possède les types $\texttt{int} \to \texttt{int}$, $\texttt{bool} \to \texttt{bool}$, et plus généralement $\tau \to \tau$ pour tout type τ. On dit que cette fonction est *polymorphe*. Cependant, comme le montre le dernier exemple, dès lors qu'on la lie à une variable `f` via un `let`, on est forcé de choisir un seul de ces types.

Un problème similaire se pose pour certains opérateurs comme `fst` et `snd` qui possèdent naturellement plusieurs types, par exemple $\texttt{fst} : \texttt{int} \times \texttt{int} \to \texttt{int}$ et $\texttt{fst} : \texttt{bool} \times \texttt{bool} \to \texttt{bool}$. Le système de types monomorphe nous oblige à choisir l'un de ces types comme $TC(\texttt{fst})$.

Pour dépasser cette limitation, il faut refléter dans le système de types le caractère polymorphe de fonctions comme l'identité et d'opérateurs comme `fst`. Dans le reste de ce chapitre, nous allons introduire une forme de typage polymorphe appelée *polymorphisme paramétrique*. (D'autres formes de polymorphisme sont étudiées dans le cours 2-4 : polymorphisme par sous-typage, polymorphisme par surcharge.)

Le polymorphisme paramétrique introduit dans l'algèbre de types des types de la forme

$$\forall\alpha.\ \alpha \to \alpha$$

où α est une *variable de type* qui représente un type quelconque, et la quantification $\forall\alpha$ indique que tous les choix de types τ à mettre à la place de α sont valides. Le type $\forall\alpha.\ \alpha \to \alpha$ représente donc l'ensemble infini des types $\{\tau \to \tau \mid \tau \text{ type}\}$ que l'on peut attribuer à la fonction $\texttt{fun}\ x \to x$.

De même, le polymorphisme nous permet d'attribuer les types suivants aux opérateurs polymorphes :

$$TC(+) = \texttt{int} \times \texttt{int} \to \texttt{int} \qquad TC(\texttt{fst}) = \forall\alpha,\beta.\,\alpha \times \beta \to \alpha \qquad TC(\texttt{snd}) = \forall\alpha,\beta.\,\alpha \times \beta \to \beta$$

$$TC(\texttt{ifzero}) = \forall\alpha.\,\texttt{int} \times \alpha \times \alpha \to \alpha \qquad TC(\texttt{fix}) = \forall\alpha.\,(\alpha \to \alpha) \to \alpha$$

Le premier système de type offrant du polymorphisme paramétrique s'appelle "système F" ou "λ-calcul typé du second ordre" et a été inventé indépendamment par J.-Y. Girard et par J. C. Reynolds. L'algèbre des types est la suivante :

Types :	$\tau ::= T$		type de base
	$\mid \tau_1 \to \tau_2$		type des fonctions de τ_1 dans τ_2
	$\mid \tau_1 \times \tau_2$		type des paires de τ_1 et τ_2
	$\mid \alpha$		variable de type
	$\mid \forall\alpha.\tau$		type polymorphe

Les règles de typage sont celles du système de type monomorphe (section 3.2) plus les règles suivantes d'introduction et d'élimination des types polymorphes :

$$\frac{\Gamma \vdash a : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash a : \forall\alpha.\tau}\ (\text{gen}) \qquad\qquad \frac{\Gamma \vdash a : \forall\alpha.\tau}{\Gamma \vdash a : \tau[\alpha \leftarrow \tau']}\ (\text{inst})$$

La règle (inst) permet d'utiliser un terme ayant un type polymorphe $\forall\alpha.\tau$ avec n'importe quel type qui soit *instance* de ce type polymorphe, c'est-à-dire obtenu par spécialisation dans τ de la variable α en un type τ' arbitraire.

La règle (gen) est l'opération duale de généralisation : toute variable α qui n'est pas mentionnée dans les hypothèses de typage Γ peut être *généralisée*, c'est-à-dire quantifiée universellement. La notation $\mathcal{L}(\Gamma)$ représente l'ensemble des variables de types apparaissant libres dans Γ. Elle est formellement définie comme suit :

$$\begin{array}{rclcrcl} \mathcal{L}(T) & = & \emptyset & & \mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\ \mathcal{L}(\alpha) & = & \{\alpha\} & & \mathcal{L}(\forall\alpha.\,\tau) & = & \mathcal{L}(\tau) \setminus \{\alpha\} \\ \mathcal{L}(\tau_1 \to \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) & & \mathcal{L}(\Gamma) & = & \bigcup_{x \in \mathrm{Dom}(\Gamma)} \mathcal{L}(\Gamma(x)) \end{array}$$

Voici une dérivation de typage pour le dernier exemple qui ne "passait" pas avec le typage monomorphe :

$$\dfrac{\dfrac{\dfrac{\Gamma + \{x : \alpha\} \vdash x : \alpha}{\Gamma \vdash \texttt{fun}\ x \to x : \alpha \to \alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \forall\alpha.\ \alpha \to \alpha} \quad \dfrac{\dfrac{\dfrac{\Gamma' \vdash f : \forall\alpha.\ \alpha \to \alpha}{\Gamma' \vdash f : \texttt{int} \to \texttt{int}} \quad \dfrac{\Gamma' \vdash f : \forall\alpha.\ \alpha \to \alpha}{\Gamma' \vdash f : \texttt{bool} \to \texttt{bool}}}{\Gamma' \vdash f\ 1 : \texttt{int}} \quad \dfrac{\Gamma' \vdash \texttt{true} : \texttt{bool}}{\Gamma' \vdash f\ \texttt{true} : \texttt{bool}}}{\Gamma' \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}}{\Gamma \vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

avec $\Gamma' = \Gamma + \{f : \forall\alpha.\ \alpha \to \alpha\}$ et α choisie non libre dans Γ.

Exercice 3.1 *Construire une dérivation de typage de*

$$\Gamma \vdash \texttt{fun}\ x \to x\ x : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$$

La restriction $\alpha \notin \mathcal{L}(\Gamma)$ dans la règle (inst) est cruciale car sinon on pourrait dériver des typages faux menant à accepter des programmes absurdes. Par exemple, si on enlève la restriction dans (inst), on aurait :

$$\dfrac{\dfrac{\Gamma + \{x : \alpha\} \vdash x : \alpha}{\Gamma + \{x : \alpha\} \vdash x : \forall\alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \alpha \to \forall\alpha.\alpha}$$

ce qui n'est pas un type correct pour la fonction identité.

3.4 Typage polymorphe : le système de Hindley-Milner

La plupart des langages fonctionnels typés (Caml, SML, Haskell, ...) basent leurs systèmes de types sur une restriction du système F inventée par Milner. Dans cette restriction, les quantificateurs $\forall$ ne peuvent apparaître qu'en tête des types (on parle de quantification prénexe) mais pas dans les arguments des constructeurs de types $\to$, $\times$, ... Ainsi, $\forall\alpha.\forall\beta.\alpha \to \beta$ est autorisé, mais $(\forall\alpha.\alpha) \to (\forall\alpha.\alpha)$ est interdit. La raison de cette restriction est de rendre l'inférence de types décidable (voir le chapitre 3), alors qu'elle est indécidable pour le système F.

Cette contrainte sur la forme des types se reflète comme suit dans l'algèbre des types :

Types (ou types simples) :	$\tau ::= T$	type de base
	$\mid \tau_1 \to \tau_2$	type des fonctions de τ_1 dans τ_2
	$\mid \tau_1 \times \tau_2$	type des paires de τ_1 et τ_2
	$\mid \alpha$	variable de type
Schémas de types (ou poly-types) :	$\sigma ::= \tau \mid \forall\alpha.\sigma$	

Les environnements de typage Γ ainsi que la fonction de typage des constantes et des opérateurs TC associent des schémas de types σ aux identificateurs, constantes et opérateurs.

3.4.1 Règles de types non dirigées par la syntaxe

La relation de typage est $\Gamma \vdash a : \sigma$, associant un schéma de type à un environnement Γ et une expression a. Elle est définie par les règles suivantes, qui sont une variante de celles de système F modifiées afin de garantir que les types sont toujours bien formés (pas de $\forall$ sauf en position prénexe).

$$\Gamma \vdash x : \Gamma(x) \ \text{(var)} \qquad \Gamma \vdash c : TC(c) \ \text{(const)} \qquad \Gamma \vdash op : TC(op) \ \text{(op)}$$

$$\frac{\Gamma + \{x : \tau_1\} \vdash a : \tau_2}{\Gamma \vdash (\texttt{fun}\ x \to a) : \tau_1 \to \tau_2}\ \text{(fun)} \qquad \frac{\Gamma \vdash a_1 : \tau' \to \tau \quad \Gamma \vdash a_2 : \tau'}{\Gamma \vdash a_1\ a_2 : \tau}\ \text{(app)}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash (a_1, a_2) : \tau_1 \times \tau_2}\ \text{(paire)} \qquad \frac{\Gamma \vdash a_1 : \sigma_1 \quad \Gamma + \{x : \sigma_1\} \vdash a_2 : \sigma_2}{\Gamma \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \sigma_2}\ \text{(let)}$$

$$\frac{\Gamma \vdash a : \sigma \quad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash a : \forall\alpha.\sigma}\ \text{(gen)} \qquad \frac{\Gamma \vdash a : \forall\alpha.\sigma}{\Gamma \vdash a : \sigma[\alpha \leftarrow \tau]}\ \text{(inst)}$$

Remarquons que dans ce système, seuls les identificateurs liés par `let` peuvent recevoir un type polymorphe (un schéma σ). Les identificateurs liés par `fun` sont forcément typés de manière monomorphe, car sinon le type de la fonction serait mal formé.

Par exemple, on a encore $\Gamma \vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}$, car on peut encore typer le membre droit du `let` sous l'hypothèse $f : \forall\alpha.\ \alpha \to \alpha$.

En revanche, l'expression $\texttt{fun}\ x \to x\ x$ est maintenant non typable car on ne peut plus attribuer à x un type polymorphe. Seuls les types simples τ sont acceptés pour x, et comme dans le cas du typage monomorphe, aucun choix de τ ne permet de typer $x\ x$.

3.4.2 Règles de types dirigées par la syntaxe

On dit qu'un système de types est dirigé par la syntaxe (*syntax-directed* dans la littérature en anglais) si, pour chaque expression a, il y a au plus une règle de typage qui conclut $\Gamma \vdash a : \tau$. Autrement dit, la forme de l'expression a détermine quelle règle de typage appliquer en dernier.

Par exemple. le système de types monomorphes de la section 3.2 est dirigé par la syntaxe. Au contraire, les systèmes polymorphes des sections 3.3 et 3.4.1 ne sont pas dirigés par la syntaxe : pour chaque expression a, trois règles de typage peuvent s'appliquer (la règle (inst), la règle (gen), et l'une des règles du système monomorphe).

Étant donné un système de types, on veut construire des algorithmes de vérification ou d'inférence de types, et les programmer dans les compilateurs. Cette tâche est grandement facilitée si le système de types est dirigé par la syntaxe : la forme de l'expression à typer détermine de manière unique la forme de la dérivation de typage ; il ne reste plus au typeur qu'à essayer de calculer les types qui rendent cette dérivation valide. Avec un système non dirigé par les types, le typeur peut devoir considérer plusieurs formes de dérivations, avec du retour en arrière (*backtracking*) en cas de problèmes. C'est plus compliqué et potentiellement moins efficace.

Nous allons maintenant donner des règles dirigée par la syntaxe pour le système de types de Hindley-Milner. Ces règles ont la même puissance d'expression que celles de la section 3.4.1 : tout terme clos a est typable dans le jeu de règles dirigé par la syntaxe si et seulement si a est typable dans le jeu de règles de la section 3.4.1.

$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau}\ \text{(var-inst)} \qquad \frac{\tau \leq TC(c)}{\Gamma \vdash c : \tau}\ \text{(const-inst)} \qquad \frac{\tau \leq TC(op)}{\Gamma \vdash op : \tau}\ \text{(op-inst)}$$

$$\frac{\Gamma + \{x : \tau_1\} \vdash a : \tau_2}{\Gamma \vdash (\texttt{fun}\ x \to a) : \tau_1 \to \tau_2}\ \text{(fun)} \qquad \frac{\Gamma \vdash a_1 : \tau' \to \tau \quad \Gamma \vdash a_2 : \tau'}{\Gamma \vdash a_1\ a_2 : \tau}\ \text{(app)}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash (a_1, a_2) : \tau_1 \times \tau_2}\ \text{(paire)} \qquad \frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma + \{x : \mathit{Gen}(\tau_1, \Gamma)\} \vdash a_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2}\ \text{(let-gen)}$$

La relation $\tau \leq \sigma$ (lire : "le type τ est une instance du schéma σ") est définie comme suit :

$\tau \leq \forall \alpha_1 \ldots \alpha_n . \tau'$ si et seulement si il existe $\tau_1, \ldots, \tau_n$ tels que $\tau = \tau'[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]$

Symétriquement, l'opérateur $\mathit{Gen}(\tau, \Gamma)$ calcule le schéma obtenu en généralisant dans τ toutes les variables de types α qui sont généralisables, c'est-à-dire non libres dans Γ :

$$\mathit{Gen}(\tau_1, \Gamma) = \forall \alpha_1 \ldots \alpha_n . \tau_1 \quad \text{où} \quad \{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(\Gamma)$$

Par-rapport aux règles de la section 3.4.1, on voit que les deux règles non dirigées par la syntaxe, (inst) et (gen), ont été combinées avec d'autres règles : l'instantiation est effectuée "au vol" dans les règles pour les variables, constantes et opérateurs (var-inst, const-inst et op-inst) ; la généralisation est effectuée uniquement sur le type de la partie gauche d'un `let` (règle let-gen).

En conséquence de ces regroupements de règles, le type attribué à une expression est toujours un type simple τ et non plus un schéma σ. La relation de typage est donc de la forme $\Gamma \vdash a : \tau$ au lieu de $\Gamma \vdash a : \sigma$.

Exemple : on peut construire la dérivation suivante pour $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$.

$$\dfrac{\dfrac{\dfrac{}{\alpha \leq \alpha}}{\Gamma + \{x : \alpha\} \vdash x : \alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\texttt{int} \to \texttt{int} \leq \forall\alpha.\ \alpha \to \alpha}{\Gamma' \vdash f : \texttt{int} \to \texttt{int}} \quad \Gamma' \vdash 1 : \texttt{int}}{\Gamma' \vdash f\ 1 : \texttt{int}} \quad \dfrac{\dfrac{\texttt{bool} \to \texttt{bool} \leq \forall\alpha.\ \alpha \to \alpha}{\Gamma' \vdash f : \texttt{bool} \to \texttt{bool}} \quad \Gamma' \vdash \texttt{true} : \texttt{bool}}{\Gamma' \vdash f\ \texttt{true} : \texttt{bool}}}{\Gamma' \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

$$\Gamma \vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}$$

avec $\Gamma' = \Gamma + \{f : \mathit{Gen}(\alpha \to \alpha, \Gamma)\} = \Gamma + \{f : \forall\alpha.\ \alpha \to \alpha\}$ si α est choisie non libre dans Γ.

3.5 Sûreté d'un système de type

Nous avons affirmé que le but du typage statique est de rejeter les programmes absurdes. Le slogan que l'on trouve souvent dans la littérature est *well-typed programs do not go wrong*. Il est temps de donner de la substance à cette affirmation, d'une part en formalisant ce qu'on entend par "programmes absurdes" ou "*to go wrong*", ensuite en démontrant que tous les programmes bien typés ne sont pas absurdes. Ce dernier résultat s'appelle le théorème de sûreté (*soundness theorem*) pour le système de types considéré.

Caractérisation des programmes absurdes Dans la section 2.4, nous avons vu qu'il y a trois formes possibles d'exécutions pour un programme a donné :

1. $a \to a_1 \to a_2 \to \ldots \to v$
   (L'évaluation termine normalement en renvoyant une valeur.)
2. $a \to a_1 \to a_2 \to \ldots \to a_n$ et $a_n \not\to$ et a_n n'est pas une valeur
   (L'évaluation bloque sur un terme irréductible qui n'est pas une valeur.)
3. $a \to a_1 \to a_2 \to a_3 \ldots$
   (L'évaluation se poursuit indéfiniment, sans jamais terminer ni bloquer.)

C'est le cas (2) que nous allons prendre comme caractérisation des programmes absurdes. En effet, dans notre vision de haut niveau, les termes irréductibles qui ne sont pas des valeurs correspondent, dans une vision "machine" de plus bas niveau, à des configurations d'exécution qui peuvent faire "planter" le code machine. Par exemple, une machine qui tenterait d'exécuter 1 2 ou `fst`(1) va utiliser l'entier 1 comme un pointeur vers une fonction ou vers une paire, et faire "segmentation fault" lors du déréférencement de ce pointeur.

Notons que les programmes qui ne terminent pas (cas 3) ne sont pas considérés comme absurdes, et nous n'attendons pas de nos systèmes de types qu'ils rejettent les programmes qui ne terminent pas. En d'autres termes, le but du typage statique en programmation n'est pas de garantir la terminaison des programmes. C'est une différence majeure par-rapport aux systèmes de types utilisés comme formalismes logiques (théorie des types), où les termes qui ne terminent pas sont des preuves mal formées et doivent être rejetés par les règles de typage.

Autre remarque : on voit sur cette caractérisation des programmes absurdes que la sémantique à grands pas de la section 2.3 ne nous permet pas de caractériser les programmes absurdes. En effet, cette sémantique ne distingue pas les programmes a qui bloquent (cas 2) des programmes qui

bouclent (cas 3) : dans les deux cas, il n'existe pas de valeur v telle que $a \xrightarrow{v} v$. C'est pour cette raison qu'on utilise généralement des sémantiques à petits pas pour énoncer et prouver la sûreté d'un système de types.

Le théorème de sûreté du typage Nous pouvons maintenant énoncer le théorème qui relie typage et sémantique, et montre qu'un système de types remplit bien son office :

Théorème 3.1 (Sûreté du typage) *Si $\emptyset \vdash a : \tau$, alors toutes les suites de réductions commençant par a ou bien sont infinies ($a \to a_1 \to a_2 \to a_3 \to \ldots$), ou bien se terminent sur une valeur ($a \xrightarrow{*} v$).*

Dans l'hypothèse, on exige que a soit bien typé dans l'environnement vide de typage $\emptyset$ afin de garantir que a est clos et que donc son évaluation ne va pas bloquer sur la réduction d'une variable libre.

Un énoncé équivalent mais mathématiquement plus clair est le suivant :

Théorème 3.2 (Sûreté du typage) *Si $\emptyset \vdash a : \tau$ et $a \xrightarrow{*} a'$ et a' est irréductible, alors a' est une valeur.*

La preuve de ce théorème (ou plus exactement de ces théorèmes – un par système de type considéré) sera détaillée dans le cours 2-4. Nous donnons ici juste l'idée de la preuve, qui s'appuie sur les deux lemmes fondamentaux suivants :

Théorème 3.3 (Préservation du typage par réduction) *Si $\emptyset \vdash a : \tau$ et $a \to a'$, alors $\emptyset \vdash a' : \tau$*

Théorème 3.4 (Progression) *Si $\emptyset \vdash a : \tau$, alors de deux choses l'une : ou bien a est une valeur, ou bien il existe a' tel que $a \to a'$.*

Une fois ces deux lemmes montrés (ce qui est l'essentiel de la difficulté), la preuve du théorème 3.2 est simple :

Démonstration: (du théorème 3.2). Considérons a et a' tels que $\emptyset \vdash a : \tau$ et $a \xrightarrow{*} a'$ et a' est irréductible.

Nous avons $\emptyset \vdash a' : \tau$. En effet, écrivons la réduction de a vers a' sous la forme $a \to a_1 \to \ldots \to a_n \to a'$. Par le lemme de préservation (3.3) appliqué à a, nous obtenons $\emptyset \vdash a_1 : \tau$. Appliquant ce lemme successivement à $a_1, \ldots, a_n$, nous obtenons $\emptyset \vdash a_i : \tau$ pour $i = 1, \ldots, n$, et enfin $\emptyset \vdash a' : \tau$.

Appliquant le théorème de progression (3.4) à a', nous obtenons que ou bien a' est une valeur, ou bien a' se réduit. Comme a' est irréductible par hypothèse, c'est donc que a' est une valeur. CQFD. □

Chapitre 4

Inférence de types

4.1 Problèmes algorithmiques de typage

Pour un langage statiquement typé, un *typeur* est un algorithme qui prend un programme en entrée et détermine si ce programme est typable ou non. En général, il va aussi déterminer un type τ pour ce programme.

Si plusieurs types τ sont possibles, on essaye de renvoyer comme résultat du typeur un *type principal*, c'est-à-dire un type plus précis que tous les autres types possibles (pour une notion de "plus précis" qui dépend du système de types considéré). Par exemple, en mini-ML, `fun` $x \to x$ a les types $\tau \to \tau$ pour n'importe quel type τ, mais le type $\alpha \to \alpha$ est principal, puisque tous les autres types possibles s'en déduisent par substitution.

La quantité de travail fournie par le typeur est inversement proportionnelle à la quantité de déclarations de types présentes dans le langage source, et donc à la quantité d'information de typage écrite par le programmeur :

Vérification pure : Dans le source, toutes les sous-expressions du programme, ainsi que tous les identificateurs, sont annotés par leur type.

```
fun (x:int) →
  (let (y:int) = (+:int×int→int)((x:int),(1:int):int×int) in (y:int) : int)
```

Le typeur est alors très simple, puisque le programme source contient déjà autant d'informations de typage que la dérivation de typage correspondante. La patience du programmeur est mise à rude épreuve par la quantité d'annotations de types à fournir. Aucun langage réaliste ne suit cette approche.

Déclaration des types des identificateurs et propagation des types : Le programmeur déclare les types des paramètres de fonctions et des variables locales. Le typeur infère les types des sous-expressions en "propageant" les types des feuilles de l'expression vers la racine. Par exemple, sachant que `x` est de type `int`, le typeur peut non seulement vérifier que l'expression `x+1` est bien typée, mais aussi inférer qu'elle a le type `int`. L'exemple ci-dessus devient :

```
fun (x:int) → let (y:int) = +(x,1) in y
```

Le typeur infère le type `int` → `int` pour cette expression. Cette approche est suivie par la plupart des langages impératifs : Pascal, C, Java, ...(En fait, ces langages exigent un peu plus d'annotations de types ; par exemple, il faut aussi déclarer le type du résultat des fonctions.)

Déclaration des types des paramètres de fonction et propagation des types : La seule différence par-rapport à l'approche précédente est que les variables locales (p.ex. les identificateurs liés par `let`) ne sont pas annotées par leur type, ce dernier étant déterminé par le type de l'expression liée à la variable. Exemple :

```
fun (x:int) → let y = +(x,1) in y
```

Ayant déterminé que `+(x,1)` est de type `int`, le typeur déduit que `y` est de type `int` dans le reste de la fonction.

Inférence complète de types : Le programme source ne contient aucune déclaration de type sur les paramètres de fonctions ni sur les variables locales. Le typeur détermine le type des paramètres de fonctions d'après l'utilisation qui en est faite dans le reste du programme. Exemple :

```
fun x → let y = +(x,1) in y
```

Puisque l'addition `+` n'opère que sur des paires d'entiers, `x` est forcément de type `int`. C'est l'approche suivie dans les langages fonctionnels tels que Caml, SML, et Haskell.

4.2 Propriétés attendues d'un algorithme de typage

Quelle que soit l'approche choisie pour le typeur (vérification, propagation, inférence), on s'attend à ce que l'algorithme du typeur soit conforme aux règles du système de types, en ce sens :

- *Correction* : si l'algorithme répond "oui" pour un programme donné, ce programme doit être typable d'après le système de types. Si l'algorithme répond "oui, et de plus le type est τ", le programme doit posséder le type τ d'après le système de types.
- *Complétude* : si le programme est typable d'après le système de types, l'algorithme doit répondre "oui".

Une troisième propriété, moins essentielle, est souvent nécessaire pour prouver la complétude :

- *Principalité* : si l'algorithme répond "oui, ce programme est typable, et de plus son type est τ", alors τ doit être *plus général* (au moins aussi précis) que tous les types τ' attribuables au programme d'après le système de types.

Des exemples de preuves de correction, complétude et principalité d'un algorithme d'inférence de types seront étudiées dans le cours 2-4.

4.3 Inférence de types pour mini-ML : l'algorithme W

Nous allons maintenant présenter l'algorithme standard d'inférence de types pour le système de types de Hindley-Milner. Cet algorithme est connu sous le nom de "W" et est dû à Milner, Damas et Tofte. Les idées de base de cet algorithme sont les suivantes :

- *Propagation des types inférés des sous-expressions vers les expressions englobantes.* Par exemple, l'expression 1 a forcément le type `int`. Pour des expressions comme (a_1, a_2) ou `let` $x = a_1$ `in` a_2, l'algorithme détermine récursivement les types de a_1 et a_2 et en déduit le type de l'expression.
- *Utilisation de nouvelles variables de types pour représenter les types inconnus.* Lorsque l'algorithme type une fonction `fun` $x \to a$, il ne sait pas quel type attribuer à x pour typer a. L'idée est de repousser la détermination du type de x à plus tard, et de typer a sous l'hypothèse $x : \alpha$ où α est une nouvelle variable de type.
  De même, lorsque l'algorithme type une utilisation d'un identificateur polymorphe comme `id` : $\forall\alpha.\ \alpha \to \alpha$, il ne sait pas quelle instance de ce schéma de type renvoyer. Il va donc renvoyer une *instance triviale* du schéma obtenue en remplaçant les variables universellement quantifiées par de nouvelles variables de types. Ainsi, le type renvoyé pour une référence à `id` est $\beta \to \beta$ pour une nouvelle variable β, c'est-à-dire une instance triviale de $\forall\alpha.\ \alpha \to \alpha$.
- *Déterminer les types inconnus par unification entre types.* Lorsque l'algorithme type une application $a_1\ a_2$, il infère un type, disons, $\tau \to \tau'$ pour a_1 et un type τ_2 pour a_2. La règle de typage de l'application exige $\tau = \tau_2$. Cependant, cette égalité n'est pas forcément déjà vraie, car τ et τ_2 peuvent contenir des variables de types introduites par l'algorithme pour représenter des types encore non déterminés. L'algorithme va donc *unifier* τ et τ_2, en replaçant des variables de types par des types plus précis jusqu'à ce que $\tau = \tau_2$.

Voici le pseudo-code (très informel) de l'algorithme d'unification entre deux types :

$$
\begin{array}{rcl}
\texttt{unifier}(T, T) & = & \text{succès} \\
\texttt{unifier}(\alpha, \tau) & = & \text{si } \alpha \notin \mathcal{L}(\tau), \text{ remplacer } \alpha \text{ par } \tau \text{ partout ;} \\
 & & \text{sinon, échec} \\
\texttt{unifier}(\tau, \alpha) & = & \texttt{unifier}(\alpha, \tau) \\
\texttt{unifier}(\tau_1 \to \tau_1', \tau_2 \to \tau_2') & = & \texttt{unifier}(\tau_1, \tau_2); \texttt{unifier}(\tau_1', \tau_2') \\
\texttt{unifier}(\tau_1 \times \tau_1', \tau_2 \to \tau_2') & = & \texttt{unifier}(\tau_1, \tau_2); \texttt{unifier}(\tau_1', \tau_2') \\
\texttt{unifier}(\tau_1, \tau_2) & = & \text{échec, dans tous les autres cas}
\end{array}
$$

Et voici le pseudocode de l'algorithme W. Il prend un environnement de typage Γ et un terme a, et renvoie le type τ inféré.

```
W(Γ, a) =

  si a est une variable x:
    renvoyer une instance triviale du schéma Γ(x)

  si a est une constante c:
    renvoyer une instance triviale du schéma TC(c)

  si a est un opérateur op:
    renvoyer une instance triviale du schéma TC(op)

  si a est une fonction fun x → b:
    soit α une nouvelle variable
    calculer τ = W(Γ + {x : α}, b)
    renvoyer α → τ

  si a est une application b c:
    calculer τ1 = W(Γ, b)
    calculer τ2 = W(Γ, c)
    soit α une nouvelle variable
    unifier(τ1, τ2 → α)
    renvoyer α

  si a est une paire (b, c):
    calculer τ1 = W(Γ, b)
    calculer τ2 = W(Γ, c)
    renvoyer τ1 × τ2

  si a est  let x=b in c:
    calculer τ = W(Γ, b)
    renvoyer W(Γ + {x : Gen(τ, Γ)}, c)
```

Exemple : voici le déroulement de l'algorithme W sur l'expression `fun` $x \to$ `fst`(x).

`W(0, fun x -> fst(x) + 1)`

`W(x:` α`, fst(x) + 1)`

`W(x:` α`, +) = int`$\times$`int` $\to$ `int`

`W(x:` α`, (fst(x), 1))`

`W(x:` α`, fst(x))`

`W(x:` α`, fst)` $= \beta \times \gamma \to \beta$

`W(x:` α`, x)` $= \alpha$

`unifier(`$\beta \times \gamma \to \beta$`,` $\alpha \to \delta$`)`

α `est remplacé par` $\beta \times \gamma$
δ `est remplacé par` β

`W(x:` $\beta \times \gamma$`, fst(x))` $= \beta$

`W(x:` $\beta \times \gamma$`, 1) = int`

`W(x:` $\beta \times \gamma$`, (fst(x), 1))` $= \beta \times$ `int`

`unifier(int`$\times$`int` $\to$ `int,` $\beta \times$ `int` $\to \varepsilon$`)`

β `est remplacé par int`
ε `est remplacé par int`

`W(x: int`$\times\gamma$`, fst(x) + 1) = int`

`W(0, fun x -> fst(x) + 1) = int`$\times\gamma$ $\to$ `int`

Le cours 2-4 donne une présentation beaucoup plus rigoureuse et plus moderne de l'inférence de types pour ML, en termes de construction et de résolution de contraintes entre types.