OCaml Internals
Implementation of an ML descendant

Theophile Ranquet

Ecole Pour l'Informatique et les Techniques Avancées
SRS 2014

ranquet@lrde.epita.fr

November 14, 2013

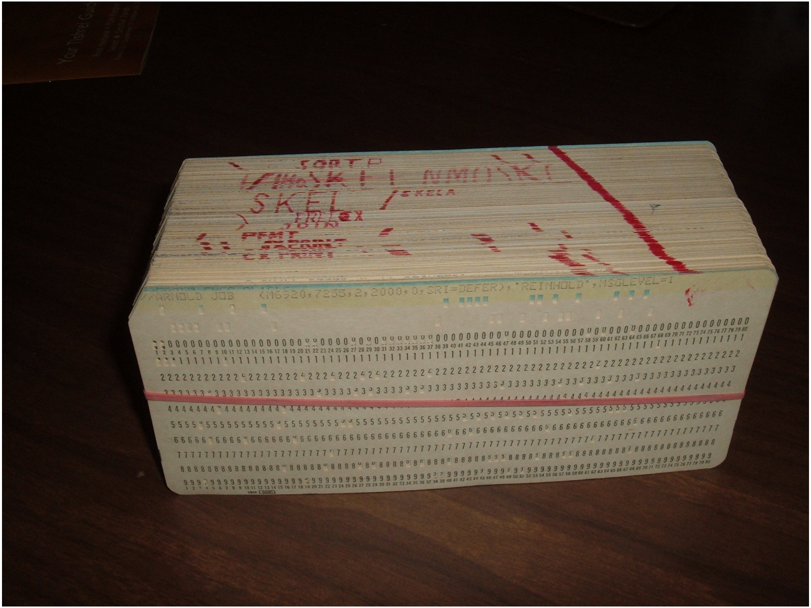
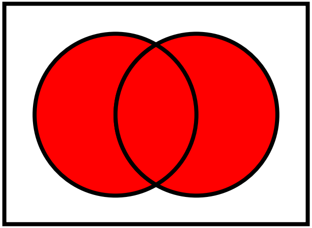

//ARNOLD JOB (M6920,7255,2,2000,0,SRI=DEFER),'REINHOLD',MSGLEVEL=1

Table of Contents

Variants

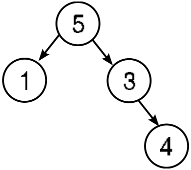
A *tagged union* (also called *variant*, *disjoint union*, *sum type*, or *algebraic data type*) holds a value which may be one of several types, but only one at a time.



This is very similar to the logical disjunction, in intuitionistic logic (by the Curry-Howard correspondance).

Variants are very convenient to represent data structures, and implement algorithms on these :

```
datatype tree = Leaf
              | Node of (int * tree * tree)

Node(5, Node(1,Leaf,Leaf), Node(3, Leaf, Node(4, Leaf, Leaf)))
```




```
fun countNodes(Leaf) = 0
  | countNodes(Node(int,left,right)) =
    1 + countNodes(left) + countNodes(right)
```

```
type basic_color =
  | Black | Red | Green | Yellow
  | Blue | Magenta | Cyan | White
type weight = Regular | Bold
type color =
  | Basic of basic_color * weight
  | RGB   of int * int * int
  | Gray  of int

```

```
let color_to_int = function
  | Basic (basic_color,weight) ->
    let base = match weight with Bold -> 8 | Regular -> 0 in
    base + basic_color_to_int basic_color
  | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
  | Gray i -> 232 + i

```

The limit of variants

Say we want to handle a color representation with an alpha channel, but just for `color_to_int` (this implies we do not want to redefine our `color` type, this would be a hassle elsewhere).

```
type extended_color =
  | Basic of basic_color * weight
  | RGB  of int * int * int
  | Gray of int
  | RGBA of int * int * int * int

let extended_color_to_int = function
  | RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | (Basic _ | RGB _ | Gray _) as color -> color_to_int color
(* Characters 154-159:
   Error: This expression has type extended_color
          but an expression was expected of type color *)
```

Table of Contents

Polymorphic variants

Polymorphic variants are more flexible and syntactically more lightweight than ordinary variants, but that extra power comes at a cost.

Syntactically, polymorphic variants are distinguished from ordinary variants by the leading backtick. And unlike ordinary variants, polymorphic variants can be used without an explicit type declaration :

```
let three = `Int 3
(* val three : [> `Int of int ] = `Int 3 *)

let li = [`On; `Off]
(* val li : [> `Off | `On ] list = [`On; `Off] *)
```

[< and [>

The > at the beginning of the variant types is critical because it marks the types as being open to combination with other variant types. We can read the type `[> ‘On | ‘Off ]` as describing a variant whose tags include `‘On` and `‘Off`, but may include more tags as well. In other words, you can roughly translate > to mean : "these tags or more."

```
‘Unknown :: li
(* -: [> ‘Off | ‘On | ‘Unknown ] list =[‘Unknown; ‘On; ‘Off]*)

let f = function ‘A | ‘B -> ()
(* val f : [< ‘A | ‘B ] -> unit = <fun> *)

let g = function ‘A | ‘B | _ -> ()
(* val f : [> ‘A | ‘B ] -> unit = <fun> *)
```

Extending types

```
let f = function `A -> `C | `B -> `D | x -> x
(* val f : ([> `A | `B | `C | `D ] as 'a) -> 'a = <fun> *)

f `E
(* - : [> `A | `B | `C | `D | `E ] = `E *)

f
(* val f : ([> `A | `B | `C | `D ] as 'a) -> 'a = <fun> *)
```

Abbreviations

Beware of the similarity :

```
type ab = A | B

type ab = [ `A | `B ]
```

```
let f (x:ab) = match x with v -> v
(* val f : ab -> ab = <fun> *)

f `A
(* - : ab = `A *)

f `C
(* Error: This expression has type [> `C ]
          but an expression was expected of type ab
          The second variant type does not allow tag(s) `C *)
```

Abbreviations

Useful shorthand :

```
let f = function `C -> 1 | #ab -> 0
(* val f : [< `A | `B | `C ] -> int = <fun> *)

f `A
(* - : int = 0 *)

f `C
(* - : int = 1 *)

f `D
(*   Error: This expression has type [> `D ]
     but an expression was expected of type [< `A | `B | `C ]
           The second variant type does not allow tag(s) `D *)

```

The solution to our color problem

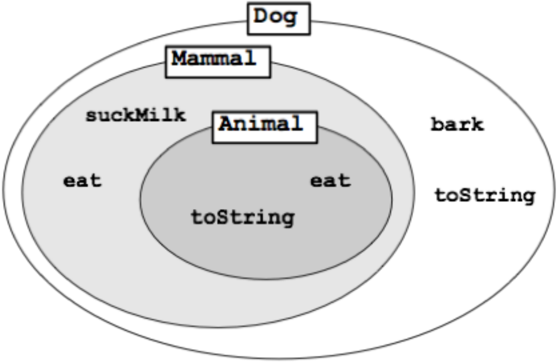
```
let extended_color_to_int = function
    | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color

(* val extended_color_to_int :
  [< `Basic of
       [< `Black
        | `Blue
        | `Cyan
        | `Green
        | `Magenta
        | `Red
        | `White
        | `Yellow ] *
       [< `Bold | `Regular ]
   | `Gray of int
   | `RGB of int * int * int
   | `RGBA of int * int * int * int ] ->
  int = <fun> *)
```

Table of Contents

Subtype polymorphism

In programming language theory, *subtyping* (also *subtype polymorphism* or *inclusion polymorphism*) is a form of type polymorphism in which a **subtype** is a datatype that is related to another datatype (the **supertype**) by some notion of *substitutability*, meaning that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype.

Subtypes



Subtype polymorphism (ctd.)

In object-oriented programming the term ‘polymorphism’ is commonly used to refer solely to this subtype polymorphism, while the techniques of *parametric polymorphism* would be considered **generic programming**.

In the branch of mathematical logic known as type theory, **System $\mathbf{F}_{<:}$**, pronounced "F-sub", is an extension of system F with subtyping. System $F_{<:}$ has been of central importance to programming language theory since the 1980s because the core of functional programming languages, like those in the ML family, support both parametric polymorphism and record subtyping, which can be expressed in System $F_{<:}$.

Example : in Object Oriented Programming

```
type farm = quadrupede list

let li : farm = new cat :: new dog :: []
(* Error: This expression has type cat but an expression was
    expected of type quadrupede *)
```

Unlike other languages, cat and dog being both derived from the quadrupede class isn't enough to treat them both as quadrupedes. One must use an explicit **coercion** :

```
let x :> quadrupede = new cat
(* val x : quadrupede = <obj> *)

let li : farm = (new cat :> quadrupede) :: (new dog :>
    quadrupede) :: [] in
  List.iter (fun c -> print_endline (c#species())) li
```

Coercion on variants

```
let f x = (x : [ `A ] :>  [ `A | `B ])
(* val f : [ `A ] -> [ `A | `B ] = <fun> *)

fun x -> (x :> [`A|`B|`C])
(* - : [< `A | `B | `C ] -> [ `A | `B | `C ] = <fun> *)
```

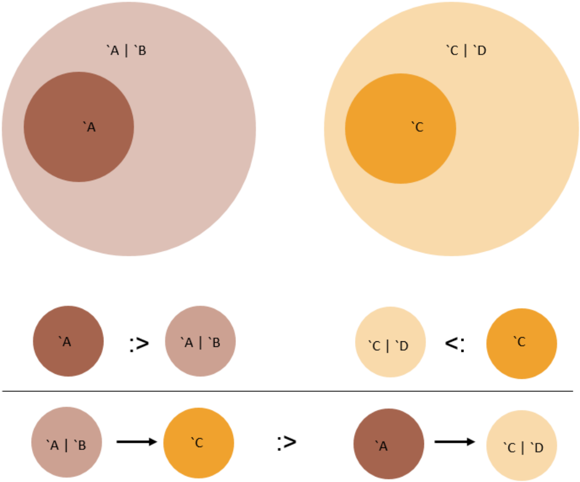
On tuples :

```
let f x = (x : [ `A ] * [ `B ] :> [ `A | `C ] * [ `B | `D ])
(* val f : [ `A ] * [ `B ] -> [ `A | `C ] * [ `B | `D ] = <fun
    > *)
```

On arrow types :

```
let f x = (x : [ `A | `B ] -> [ `C ] :> [ `A ] -> [ `C | `D ])
(* val f : ([ `A | `B ] -> [ `C ]) -> [ `A ] -> [ `C | `D ] =
    <fun> *)
```

Variance : covariance and contravariance



Variance annotations : +a and -a

A somewhat obscure corner of the language.
For types in OCaml like tuple and arrow, one can use + and -, called "variance annotations", to state the essence of the subtyping rule for the type – namely the direction of subtyping needed on the component types in order to deduce subtyping on the compound type.

```
type (+'a, +'b) t = 'a * 'b
(* type ('a, 'b) t = 'a * 'b *)

type (+'a, +'b) t = 'a -> 'b
(* Error: In this definition, expected parameter variances are
          not satisfied. The 1st type parameter was expected
          to be covariant, but it is injective contravariant.
    *)

type (-'a, +'b) = 'a -> 'b
(* type ('a, 'b) t = 'a -> 'b *)
```

Variance annotations : +a and -a (ctd.)

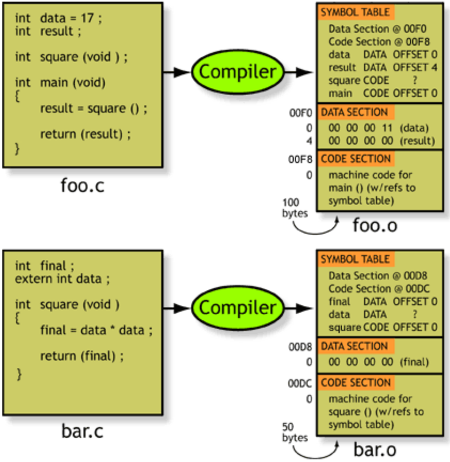
```
module M : sig
  type ('a, 'b) t
end = struct
  type ('a, 'b) t = 'a * 'b
end

let f x = (x : ([ `A ], [ `B ]) M.t
          :> ([ `A | `C ], [ `B | `D ]) M.t)
(* Error: Type ([ `A ], [ `B ]) M.t is not a subtype of
           ([ `A | `C ], [ `B | `D ]) M.t
         The first variant type does not allow tag(s) `C *)
```

Variance annotations : +a and -a (ctd.)

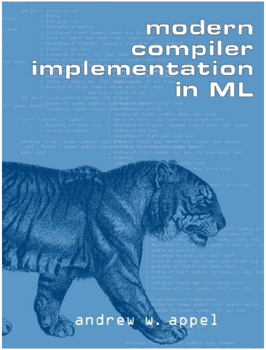
```
module M : sig
  type (+'a, +'b) t
end = struct
  type ('a, 'b) t = 'a * 'b
end

let f x = (x : ([ `A ], [ `B ]) M.t
            :> ([ `A | `C ], [ `B | `D ]) M.t)
(* Ok *)
```

Table of Contents

Compilers



int data = 17 ;
int result ;
int square (void) ;
int main (void)
{
result = square () ;
return (result) ;
}
foo.c
Compiler
SYMBOL TABLE
Data Section @ 00F0
Code Section @ 00F8
data DATA OFFSET 0
result DATA OFFSET 4
square CODE ?
main CODE OFFSET 0
00F0
DATA SECTION
0
4
00 00 00 11 (data)
00 00 00 00 (result)
00F8
CODE SECTION
0
machine code for main () (w/refs to symbol table)
100 bytes
foo.o
int final ;
extern int data ;
int square (void)
{
final = data * data ;
return (final) ;
}
bar.c
Compiler
SYMBOL TABLE
Data Section @ 00D8
Code Section @ 00DC
final DATA OFFSET 0
data DATA ?
square CODE OFFSET 0
00D8
DATA SECTION
0
00 00 00 00 (final)
00DC
CODE SECTION
0
machine code for square () (w/refs to symbol table)
50 bytes
bar.o



modern compiler implementation in ML
andrew w. appel

The OCaml toolchain

```
Source code                              Lambda
     |                                    /   \
     | parsing and preprocessing         /     \ closure conversion,
     |                                  /       \  inlining, uncurrying,
     | camlp4 syntax extensions        v         \  data representation
     |                               Bytecode     \  strategy
     v                                  |          +----+
 Parsetree (untyped AST)                |              Cmm
     |                                  |ocamlrun       |
     | type inference and checking      |               | code generation
     v                                  |               | assembly and
 Typedtree (type-annotated AST)         |               | linking
     |                                  v               v
     | pattern-matching compilation  Interpreted    Compiled
     | elimination of modules and classes
     v
  Lambda
```

Table of Contents

Value representation

In a running OCaml program, a value is either an "integer-like thing" or a pointer to a block.

Stored as integers (1 word) :

- integers
- characters
- (), []
- true, false
- variants with no parameters

Stored as blocks :

- lists
- tuples
- arrays, strings
- structs
- variants with parameters

```
let (%) x y = y x
(* val ( % ) : 'a -> ('a -> 'b) -> 'b = <fun> *)
Obj.repr 42
(* - : Obj.t = <abstr> *)
```

```
Obj.repr 42 % Obj.is_int
(* - : bool = true *)
Obj.repr 42 % Obj.is_block
(* - : bool = false *)

Obj.repr [] % Obj.is_int
(* - : bool = true *)
Obj.repr [42] % Obj.is_int
(* - : bool = false *)
```

```
type t = Foo
  | Bar of int
  | Baz
  | Qux

Obj.repr Foo % Obj.is_int
(* - : bool = true *)
Obj.repr (Bar 42) % Obj.is_int
(* - : bool = false *)

Printf.printf "%d %d %d"
  (Obj.magic Foo)
  (Obj.magic Baz)
  (Obj.magic Qux)
(* 0 1 2- : unit = () *)
```

The **Obj** module : operations on internal representations of values.

The LSB (least significant byte)

Pointers are always aligned on 1 word, i.e. 4 bytes (32 bits) or 8 bytes (64 bits). Thus, their 2 (32 bits) or 3 (64 bits) least significant bits are always null.

```
+-----------------------------------------+---+---+
| pointer                                 | 0 | 0 |
+-----------------------------------------+---+---+

+---------------------------------------------+---+
| integer (31 or 63 bits)                     | 1 |
+---------------------------------------------+---+
```

How do integer arithmetics work with this LSB ?

```
+-------+---+     +-------+---+     +-------+---+
|   a   | 1 |  +  |   b   | 1 |  =  | a + b | 1 |
+-------+---+     +-------+---+     +-------+---+
```

To perform an addition :

```
2 * a + 1
+ 2 * b + 1
= 2 * (a + b) + 2
```

So just add the two, then subtract 1.

```
; addition
lea     -1(%eax, %ebx), %eax

; subtraction
subl    %ebx, %eax
incl    %eax

; multiplication
sarl    $1, %ebx
decl    %eax
imull   %ebx, %eax
incl    %eax
```

Block values

```
+---------------+---------------+---------------+- - - - -
| header        | word[0]       | word[1]       | ....
+---------------+---------------+---------------+- - - - -
                  ^
                  |
               pointer
```

In a block (array, list, etc.) of (integers | block values), each word is (an integer | a pointer to a block value).

Block values header

```
+---------------+---------------+----------+--+--+---------------+
|       size of the block in words         | col |    tag byte    |
+---------------+---------------+----------+--+--+---------------+
<-- 22 bits (i686) or 54 bits (x86_64) ---><- 2b-><--- 8 bits --->
```

- Size : 22 bits $\Rightarrow$ maximum size of 2^{22} words (16 MBytes).
- Color : Used by the garbage collector (GC).
- Tag :
  - $\in [0; 250]$: Block contains values which the GC should scan :
    - Arrays ;
    - Objects ;
  - $\in [251; 255]$: Block contains values which the GC should **not** scan :
    - Strings ;
    - Doubles.

The tag byte

```
Obj.tag (Obj.repr "foo")
(* - : int = 252 *)

Obj.tag (Obj.repr 1.0)
(* - : int = 253 *)

Obj.double_tag
(* - : int = 253 *)

Obj.is_block (Obj.repr 1.0)
(* - : bool = true *)
```

```
Obj.tag
  (Obj.repr [| 1.0; 2.0 |])
(* - : int = 254 *)

Obj.double_field
  (Obj.repr [| 1.1; 2.2 |]) 1
(* - : float = 2.2 *)

Obj.double_field
  (Obj.repr 1.234) 0
(* - : float = 1.234 *)
```

Variants with parameters

Stored as blocks, with the **value tags** ascending from 0. Due to this encoding, there is a limit around 240 variants (with parameters).

```
type t = Apple | Orange of int | Pear of string | Kiwi

Obj.tag (Obj.repr (Orange 1234))
(* - : int = 0 *)

Obj.tag (Obj.repr (Pear "xyz"))
(* - : int = 1*)

(Obj.magic (Obj.field (Obj.repr (Orange 1234)) 0) : int)
(* - : int = 1234 *)

(Obj.magic (Obj.field (Obj.repr (Pear "xyz")) 0) : string)
(* - : string = "xyz" *)
```

C string handling

```
Obj.size (Obj.repr "1234567") (* 7 chars *)
(* - : int = 2 *)
Obj.size (Obj.repr "123456789abc") (* 12 chars *)
(* - : int = 4 *)
```

```
strlen = number_of_words_in_block * sizeof(word)
  + last_byte_of_block - 1

+-------------+-------------+
| 31 32 33 34 | 35 36 37 00 | (i686)
+-------------+-------------+


+----------------+-------------------------+
| 31 32 33 34 ... | 39 61 62 63 00 00 00 03 | (X86_64)
+----------------+-------------------------+
```

C string handling (ctd.)

String length mod 4	Padding
0	00 00 00 03
1	00 00 02
2	00 01
3	00

Note : on 64bits, the padding goes down from 00 00 ... 07.

Polymorphic variants

A polymorphic variant without any parameters is stored as an **unboxed integer** and so only takes up one word of memory, just like a normal variant. This integer value is determined by applying a hash function to the *name* of the variant.

```
Pa_type_conv.hash_variant "Foo"
(* - : int = 3505894 *)

(Obj.magic (Obj.repr `Foo) : int)
(* - : int = 3505894 *)

```

Table of Contents

Allocations : obvious and hidden

```
let f x =
  let tmp = 42 in (* stuff *)
```

Ocamlopt creates a **new** tuple :

```
let f (a, b) =
  (a, b)
```

To avoid this, tell ocamlopt to use the same value :

```
let f (x : ('a * 'b)) =
  x
```

The two heaps

A typical functional programming style means that young blocks tend to die young and old blocks tend to stay around for longer than young ones. This is often referred to as the *generational hypothesis*.

OCaml's memory model is optimized for this usage.

Two **heaps** :

- the **minor (young)** heap ;
- the **major** heap.

The minor heap

Allocation is done in the **minor** heap, which holds 32 K-words (128KBytes on 32 bits, 256KBytes on 64 bits).

```
<----- allocation proceeds in this direction
+----------------------------------------+
| unallocated           |///allocated part///|
+----------------------------------------+
 ^                       ^
 |                       |
caml_young_limit     caml_young_ptr
```

The minor collection

When the minor heap runs out, it triggers a **minor collection**.

- All local roots (i.e. pointers in variables of the current environment) have their target, in the minor heap, moved over to the **major** heap ;
- Everything left in the minor heap is data which is now unreachable, so the minor heap is once again considered empty ;
- This is a **Stop&Copy** garbage collection.

The major heap

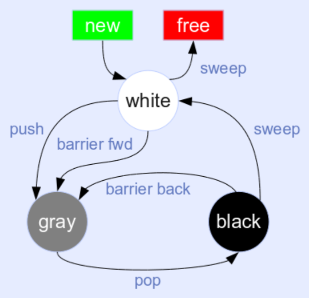
The **major** heap is a a large chunk of memory :

- Allocated by malloc(2) ;
- It does not run out ;
- It does not expire.

Because the garbage collector must not meddle with ressources allocated outside it's own heap (e.g. allocated by C code), it keeps a **page table** up to date. Any pointer which points outside those pages is considered **opaque**, and ignored. Any **block** whose **tag byte** is above 250 is also considered opaque.

The major collection

It's a simple tri-color marking, for 'on-the-fly' operation. This is known as **Mark&Sweep** garbage collection.



Garbage collection : mutables

```
type t = { tt : int }
type a = { mutable x : t }

let m = { x = { tt = 42 }};
(* minor collection happens : 'm' and its child moves to major
   heap *)

let n = { tt = 43 } in m.x <- n
(* minor collection happens : 'n' should be collected , because
   there is no local root in the minor heap; but it musn't
   because it is refered by the major heap *)
```

Because OCaml is not a purely functional language, it allows mutable contents :

- An old struct can contain a pointer to a **newer** struct ;
- The **refs list** (*remembered set*) keeps track of these.

The Gc module

Memory management control and statistics.

```
Gc.minor()
(* - : unit = () *)
Gc.compact()
(* - : unit = () *)
```

```
Gc.stat()
(* - : Gc.stat =
{Gc.minor_words=555758; Gc.promoted_words=61651; Gc.
    major_words=205646;
 Gc.minor_collections=18; Gc.major_collections=4; Gc.
    heap_words=190464;
 Gc.heap_chunks=3; Gc.live_words=130637; Gc.live_blocks=30770;
 Gc.free_words=59827; Gc.free_blocks=1; Gc.largest_free=59827;
 Gc.fragments=0; Gc.compactions=1} *)
```

The Gc module (2)

```
let c = Gc.get ()
(* val c : Gc.control =
  {Gc.minor_heap_size = 262144; Gc.major_heap_increment = 126976;
   Gc.space_overhead = 80; Gc.verbose = 0; Gc.max_overhead = 500;
   Gc.stack_limit = 1048576; Gc.allocation_policy = 0} *)

c.Gc.verbose <- 255;
Gc.set c;
Gc.compact
(* - : unit = () *)

(* <>Starting new major GC cycle
    allocated_words = 329
    extra_heap_memory = 0u
    amount of work to do = 3285u
    Marking 1274 words
    !Starting new major GC cycle
    Compacting heap...
    done. *)
```

Table of Contents

Syntax trees : a simple example

```
type t = Foo | Bar
let v = Foo
```

The untyped syntax tree

```
$ ocamlc -dparsetree typedef.ml 2>&1
[
  structure_item (typedef.ml[1,0+0]..[1,0+18])
    Pstr_type
    [
      "t" (typedef.ml[1,0+5]..[1,0+6])
        type_declaration (typedef.ml[1,0+5]..[1,0+18])
          ptype_params =
            []
          ptype_cstrs =
            []
          ptype_kind =
            Ptype_variant
              [
                (typedef.ml[1,0+9]..[1,0+12])
```

The typed syntax tree

```
$ ocamlc -dtypedtree typedef.ml 2>&1
[
  structure_item (typedef.ml[1,0+0]..typedef.ml[1,0+18])
    Pstr_type
    [
      t/1008
        type_declaration (typedef.ml[1,0+5]..typedef.ml[1,0+1
          ptype_params =
            []
          ptype_cstrs =
            []
          ptype_kind =
            Ptype_variant
              [
                "Foo/1009"
```

The untyped lambda form

The first code generation phase eliminates all the static type information into a simpler intermediate *lambda form*. The lambda form discards higher-level constructs such as modules and objects and replaces them with simpler values such as records and function pointers. Pattern matches are also analyzed and compiled into highly optimized automata.

The lambda form is the key stage that discards the OCaml type information and maps the source code to the runtime memory model. This stage also performs some optimizations, most notably converting pattern-match statements into more optimized but low-level statements.

monomorphic_large.ml :

```
type t = | Alice | Bob | Charlie | David

let test v =
  match v with
  | Alice   -> 100
  | Bob     -> 101
  | Charlie -> 102
  | David   -> 103
```

monomorphic_small.ml :

```
type t = | Alice | Bob

let test v =
  match v with
  | Alice   -> 100
  | Bob     -> 101
```

```
$ ocamlc -dlambda -c pattern_monomorphic_large.ml 2>&1
(setglobal Pattern_monomorphic_large!
  (let
    (test/1013
       (function v/1014
         (switch* v/1014
          case int 0: 100
          case int 1: 101
          case int 2: 102
          case int 3: 103)))
    (makeblock 0 test/1013)))
```

```
$ ocamlc -dlambda -c pattern_monomorphic_small.ml 2>&1
(setglobal Pattern_monomorphic_small!
  (let (test/1011 (function v/1012 (if (!= v/1012 0) 101 100)
    (makeblock 0 test/1011)))
```

```
$ ocamlc -dlambda -c pattern_polymorphic.ml 2>&1
(setglobal Pattern_polymorphic!
  (let
    (test/1008
       (function v/1009
         (if (!= v/1009 3306965)
           (if (>= v/1009 482771474) (if (>= v/1009 884917024
             (if (>= v/1009 3457716) 104 103))
           101)))
    (makeblock 0 test/1008)))
```

Benchmarking pattern matching

```
$ corebuild -pkg core_bench bench_patterns.native
$ ./bench_patterns.native -ascii
Estimated testing time 30s (change using -quota SECS).

  Name                           Time/Run   % of max
 ------------------------------ ---------- ----------
  Polymorphic pattern                 104     100.00
  Monomorphic larger pattern        95.28      91.29
  Monomorphic small pattern         53.56      51.32
```

The bytecode

```
$ ocamlc -dinstr pattern_monomorphic_small.ml 2>&1
  branch L2
L1:     acc 0
  push
  const 0
  neqint
  branchifnot L3
  const 101
  return 1
L3:     const 100
  return 1
L2:     closure L1, 0
  push
  acc 0
  makeblock 1, 0
  pop 1
  setglobal Pattern_monomorphic_small!
```

Native code : monomorphic comparision

```
let cmp (a:int) (b:int) =
  if a > b then a else b
```

```
$ ocamlopt -inline 20 -nodynlink -S compare_mono.ml
```

```
_camlCompare_mono__cmp_1008:
        .cfi_startproc
.L101:
        cmpq    %rbx, %rax
        jle     .L100
        ret
        .align  2
.L100:
        movq    %rbx, %rax
        ret
        .cfi_endproc
```

Native code : polymorphic comparision

```
let cmp a b =
  if a > b then a else b
```

```
_camlCompare_poly__cmp_1008:
        .cfi_startproc
        subq    $24, %rsp
        .cfi_adjust_cfa_offset 24
.L101:
        movq    %rax, 8(%rsp)
        movq    %rbx, 0(%rsp)
        movq    %rax, %rdi
        movq    %rbx, %rsi
        leaq    _caml_greaterthan(%rip), %rax
        call    _caml_c_call
.L102:
        leaq    _caml_young_ptr(%rip), %r11
        movq    (%r11), %r15
        cmpq    $1, %rax
        je      .L100
        movq    8(%rsp), %rax
        addq    $24, %rsp
        .cfi_adjust_cfa_offset -24
        ret
        .cfi_adjust_cfa_offset 24
        .align 2
.L100:
        movq    0(%rsp), %rax
        addq    $24, %rsp
        .cfi_adjust_cfa_offset -24
        ret
        .cfi_adjust_cfa_offset 24
        .cfi_endproc
```

Native code : comparisions benchmark

```
$ corebuild -pkg core_bench bench_poly_and_mono.native
$ ./bench_poly_and_mono.native -ascii
Estimated testing time 20s (change using -quota SECS).

  Name                        Time/Run   % of max
 ------------------------ ---------- ----------
  Polymorphic comparison      40_882     100.00
  Monomorphic comparison       2_837       6.94
```

Table of Contents

Hindley–Milner type system

A classical type system for the lambda calculus with parametric polymorphism. Among the properties making HM so outstanding is completeness and its ability to deduce the most general type of a given program without the need of any type annotations.

$$\texttt{Let} : \frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let } x = e_0 \texttt{ in } e_1 : \tau}$$

Algorithm W is a fast algorithm, performing type inference in almost linear time with respect to the size of the source, making it practically usable to type large programs.

Typing rules

Generalization : $\vdash$ **let** *id* $= \lambda x.x$ **in** *id* $: \forall\alpha.\alpha \rightarrow \alpha$

1 :	$x : \alpha \vdash x : \alpha$	[Var]	$(x : \alpha \in \{x : \alpha\})$
2 :	$\vdash \lambda x.x : \alpha \rightarrow \alpha$	[Abs]	(1)
3 :	$\vdash \lambda x.x : \forall\alpha.\alpha \rightarrow \alpha$	[Gen]	(2), $(\alpha \notin free(\epsilon))$
4 :	$id : \forall\alpha.\alpha \rightarrow \alpha \vdash id : \forall\alpha.\alpha \rightarrow \alpha$	[Var]	$(id : \forall\alpha.\alpha \rightarrow \alpha \in \{id : \forall\alpha.\alpha$
5 :	$\vdash$ **let** *id* $= \lambda x.x$ **in** *id* $: \forall\alpha.\alpha \rightarrow \alpha$	[Let]	(3), (4)

Typing rules

Generalization : $\vdash$ **let** *id* $= \lambda x.x$ **in** *id* $: \forall \alpha.\alpha \rightarrow \alpha$

1 :	$x : \alpha \vdash x : \alpha$	[Var]	$(x : \alpha \in \{x : \alpha\})$
2 :	$\vdash \lambda x.x : \alpha \rightarrow \alpha$	[Abs]	(1)
3 :	$\vdash \lambda x.x : \forall \alpha.\alpha \rightarrow \alpha$	[Gen]	(2), $(\alpha \notin free(\epsilon))$
4 :	$id : \forall \alpha.\alpha \rightarrow \alpha \vdash id : \forall \alpha.\alpha \rightarrow \alpha$	[Var]	$(id : \forall \alpha.\alpha \rightarrow \alpha \in \{id : \forall \alpha.\alpha$
5 :	$\vdash$ **let** *id* $= \lambda x.x$ **in** *id* $: \forall \alpha.\alpha \rightarrow \alpha$	[Let]	(3), (4)

Are you still there ?

Traditional Algorithm W

Here is a trivial example of generalization :

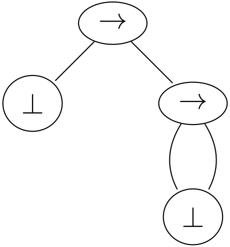
```
fun x ->
  let y = fun z -> z in y
(* 'a -> ('b -> 'b) *)
```

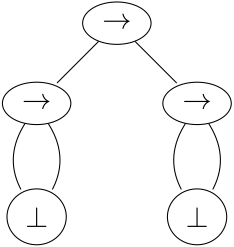
The type checker infers for `fun z ->z` the type $\beta \to \beta$ with the fresh, and hence unique, type variable β. The expression `fun z -> z` is syntactically a value, generalization proceeds, and y gets the type $\forall\beta.\beta \to \beta$.

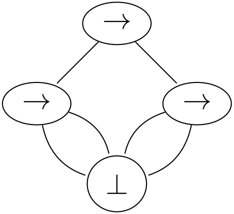
Because of the polymorphic type, y may occur in differently typed contexts (may be applied to arguments of different types), as in :

```
fun x ->
  let y = fun z -> z in
  (y 1, y true, y "caml")
(* 'a -> int * bool * string)
```

ML graphic types







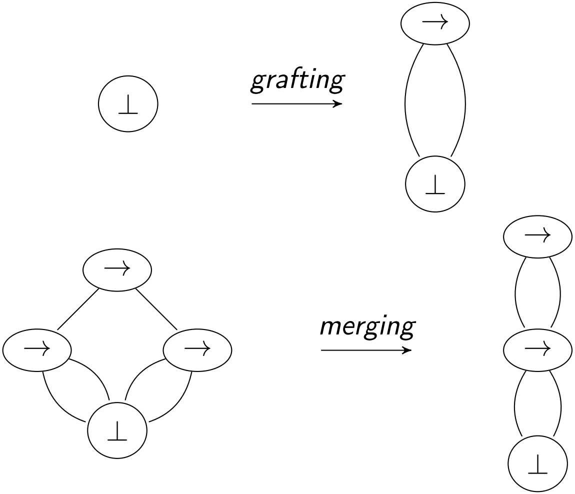
$\tau_1 : \alpha \to (\beta \to \beta)$ $\quad$ $\tau_2 : (\alpha \to \alpha) \to (\beta \to \beta)$ $\quad$ $\tau_3 : (\alpha \to \alpha) \to (\alpha \to \alpha)$

$$\tau_1 \leq \tau_2 \leq \tau_3$$

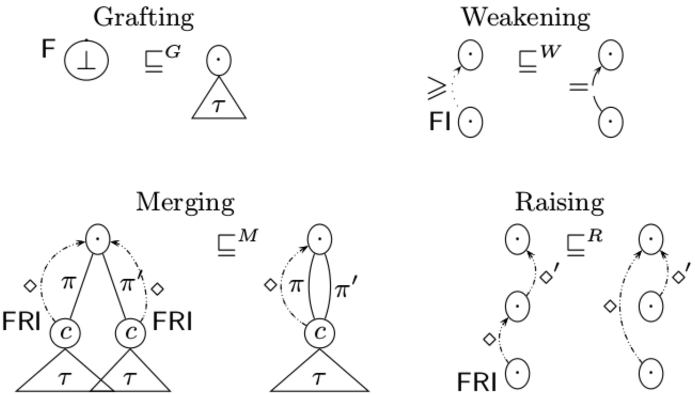
since

$$\alpha \leq (\gamma \to \gamma) \Rightarrow \langle 1 \rangle_{\tau_1} \leq \langle 1 \rangle_{\tau_2} \ (\textit{grafting})$$

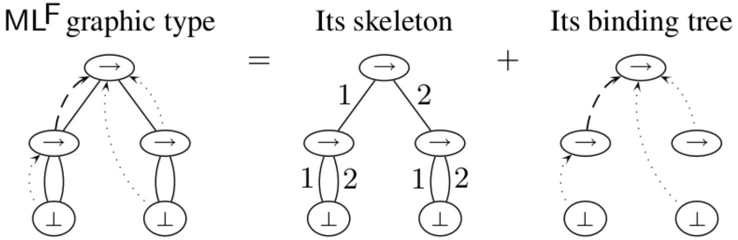
ML graphic types (2)



The instance relation



MLf



$$\sigma = \forall(\alpha)\,\forall(\beta = \forall(\gamma)\,\gamma \to \gamma)\,\forall(\delta \geqslant \alpha \to \alpha)\,\beta \to \delta$$

Figure 6. An example of MLF graphic type

MLf

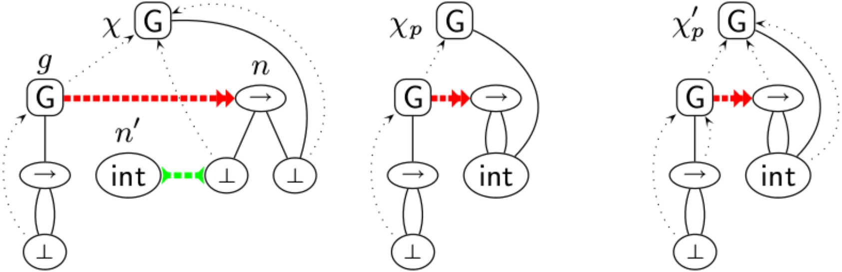


Figure 3. Typing id 1

MLf

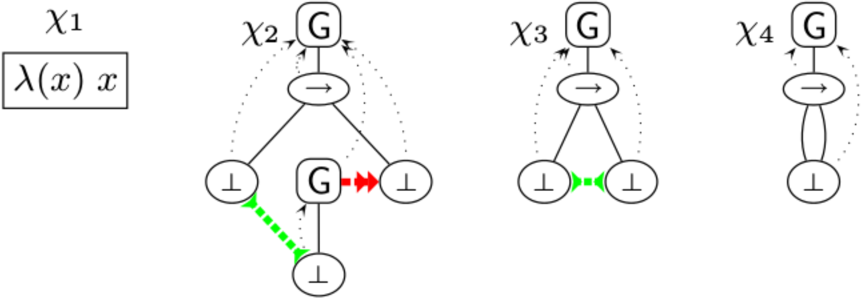


Figure 16. Typing $\lambda(x)\ x$

MLf

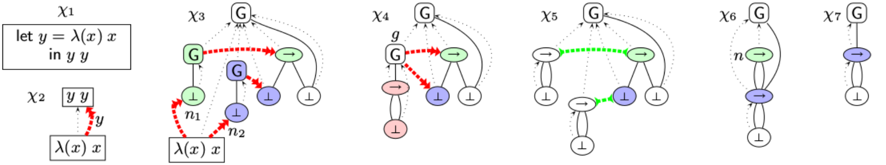


Figure 15. Typing let $y = \lambda(x)\ x$ in $y\ y$

Table of Contents

Recursive types

OCaml accepts them in variants and structures :

```
type foo = Ctor of foo
(* type foo = | Ctor of foo *)

type bar_t = { field : bar_t }
(* bar_t = { field : bar_t } *)
```

```
let rec fooval = Ctor fooval
(* val fooval : foo =
  Ctor
     (Ctor
          (Ctor
          ... *)

let rec bar = { field = baz } and baz = { field = bar }
(* val bar : bar_t = {field={field={field=...}}} *)
(* val baz : bar_t = {field={field={field=...}}} *)
```

Recursive types

But there are limitations :

```
type 'a tree = 'a * 'a tree list
(* Error: The type abbreviation tree is cyclic *)

let rec f = function
    _, [] -> ()
  | _, xs -> f xs
(* Error: This expression has type 'a list but is here used
   with type ('b * 'a list) list *)
```

Even though the object extensions do work recursively :

```
let rec f o = match o#xs with
    [] -> ()
  | xs -> f xs
  (* val height : (< xs : 'a list; .. > as 'a) -> unit = <fun>
    *)
```

Black magic

Use OCaml compiler option `-rectypes`. The previous function becomes :

```
(* f : ('b * 'a list as 'a) -> unit = <fun> *)
```

You can also do some magic with the as keyword :

```
type 'a tree = ('a * 'vertex list) as 'vertex
(* type 'a tree = 'a * 'a tree list *)
```

Table of Contents

Weak types

Type defaulting :

```
let x = ref []
(* val x : '_a list ref = {contents = []} *)

x := 42 :: !x
(* - : unit = () *)

x
(* - : int list ref = {contents = [42]} *)

x := 'a' :: !x
(* Error: This expression has type char but
   an expression was expected of type int *)
```

This is called the **value restriction**, or monomorphism restriction. It prevents breaking *type safety* with references.

Value restriction

A rule that governs when type inference is allowed to polymorphically generalize a value declaration : only if the right-hand side of an expression is syntactically a value.

```
val f = fn x => x
val _ = (f "foo"; f 13)
```

The expression `fn x => x` is syntactically a value, so `f` has polymorphic type `'a -> 'a` and both calls to `f` type check.

```
val f = let in fn x => x end
val _ = (f "foo"; f 13)
```

The expression `let in fn x => end end` is not syntactically a value and so the program fails to type check.

Value restriction (ctd.)

The Definition of Standard ML spells out precisely which expressions are syntactic values (it refers to such expressions as *non-expansive)*. An expression is a value if it is of one of the following forms :

- a constant (13, "foo", 13.0, . . .)
- a variable (x, y, . . .)
- a function (fn x => e)
- the application of a constructor other than ref to a value (Foo v)
- a type constrained value (v : t)
- a tuple in which each field is a value (v1, v2, . . .)
- a record in which each field is a value {l1 = v1, l2 = v2, . . . }
- a list in which each element is a value [v1, v2, . . .]

λ calculus

The meaning of lambda expressions is defined by how expressions can be reduced.

There are three kinds of reduction :

- α-**conversion** : changing bound variables (alpha) ;
- β-**reduction** : applying functions to their arguments (beta) ;
- η-**conversion** : which captures a notion of extensionality (eta).

We also speak of the resulting equivalences : two expressions are β-equivalent, if they can be β-converted into the same expression, and α/β-equivalence are defined similarly.

Weak polymorphism and η conversions

η reduction :

```
let f x = g x
let f = g
```

η expansion :

```
let map_id = List.map (function x -> x)
(* val map_id : '_a list -> '_a list = <fun> *)
map_id [1;2]
map_id
(* - : int list -> int list = <fun> *)

let map_id eta = List.map (function x -> x) eta
(* val map_id : 'a list -> 'a list = <fun> *)
map_id [1;2]
map_id
(* - : 'a list -> 'a list = <fun> *)
```

Variance and the relaxed value restriction

```
module M : sig
  type 'a t
  val embed : 'a -> 'a t
end = struct
  type 'a t = 'a
  let embed x = x
end

M.embed []
(* - : '_a list M.t = <abstr> *)
```

Variance and the relaxed value restriction (ctd.)

With a +'a in the type signature, the embedded empty list is generalized :

```
module M : sig
  type +'a t
  val embed : 'a -> 'a t
end = struct
  type 'a t = 'a
  let embed x = x
end

M.embed []
(* - : 'a list M.t = <abstr> *)
```

Table of Contents

Question : How does one build a function with type $\alpha \to \beta$?

Question : How does one build a function with type $\alpha \rightarrow \beta$?

Answer 1 :

```
let rec f x = f x
(* val f : 'a -> 'b = <fun> *)
```

Question : How does one build a function with type $\alpha \to \beta$?

Answer 1 :

```
let rec f x = f x
(* val f : 'a -> 'b = <fun> )*
```

Answer 2 :

```
fun _ -> failwith ""
(* - : 'a -> 'b = <fun> *)
```

Question : How does one build a function with type $\alpha \to \beta$?

Answer 1 :

```
let rec f x = f x
(* val f : 'a -> 'b = <fun> )*
```

Answer 2 :

```
fun _ -> failwith ""
(* - : 'a -> 'b = <fun> *)
```

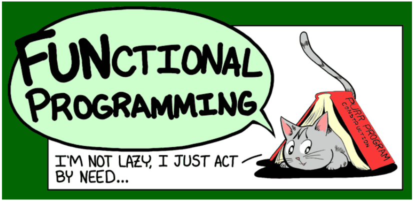
Answer 3 : Answers 1 and 2 are fallacious, this type is aberrant. We will now see why.

Table of Contents

Declarative programming

It can be described as :

- A program that describes what computation should be performed and not how to compute it ;
- Any programming language that lacks side effects ;
- A language with a clear correspondence to mathematical logic.



Why is it cool ?

- strict (eager) versus non-sritct (lazy) evaluation
  - OCaml module Lazy.
- Typed lambda-calculus is **safe** ;
- High-level programming is abstract, simple ;
- Purely functional is easy to handle for the compiler ;
  - variable life ;
  - optimization ;
  - thread-safety, etc.

1D Haar Discrete Wavelet Transform (DWT 1D)

Definition : $y[n] = (x * g)[n] = \sum_{k=-\infty}^{\infty} x[k]g[n-k]$.

Implementation :

- 28.540 lines of C ;
- or

```
let haar l =
  let rec aux l s d = match l, s, d with
    [s], [], d -> s :: d
  | [], s, d -> aux s [] d
  | h1 :: h2 :: t, s, d -> aux t (h1 + h2 :: s) (h1 - h2 :: d)
  | _ -> invalid_arg "haar" in aux l [] []
(* val haar : int list -> int list = <fun> *)
```

Table of Contents

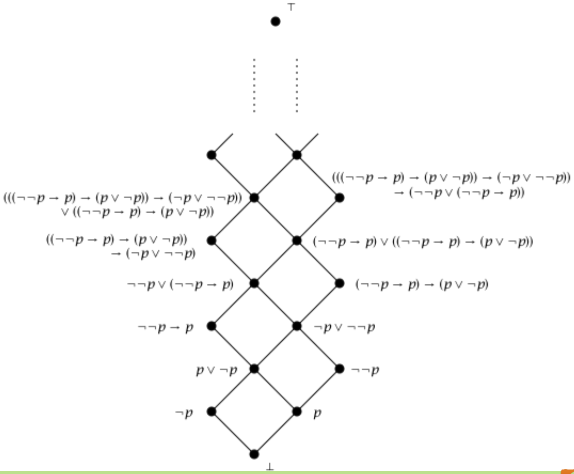
Combinators

Introduced by Moses Schönfinkel and Haskell Curry, a combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.

- $Sxyz = xz(yz)$
- $Kxy = x$
- $Ix = x$

The B, C, K, W system forms 4 axioms of sentential logic :

- $B = S(KS)K$ *functional composition*
- $C = S(S(K(S(KS)K))S)(KK)$ *swap two arguments*
- $W = SS(SK)$ *self-duplication*

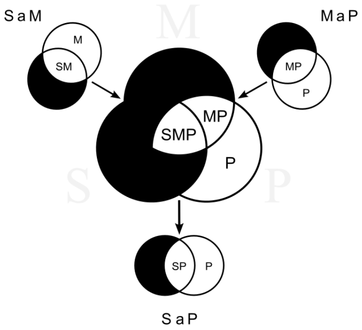


The B combinator

It's name is a reference to the **Barbara Syllogism**

AAA-1 Modus Barbara

$\overline{\exists x: Mx \wedge \overline{P}x}$	M a P	All M are P,
$\wedge\ \overline{\exists x: Sx \wedge \overline{M}x}$	S a M	and all S are M;
$\Rightarrow\ \overline{\exists x: Sx \wedge \overline{P}x}$	S a P	thus all S are P.



The X combinator

There are **one-point bases** from which every combinator can be composed extensionally equal to any lambda term. The simplest example of such a basis is {**X**} where :

$$X = \lambda x.((xS)K)$$

It is not difficult to verify that :

$$X(X(XX)) =^{\eta\beta} K \text{ and } X(X(X(XX))) =^{\eta\beta} S$$

Table of Contents

The Curry-Howard correspondence

In programming language theory and proof theory, the Curry–Howard correspondence is the direct relationship between computer programs and mathematical proofs. It is a generalization of a syntactic analogy between systems of formal logic and computational calculi that was first discovered by the American mathematician **Haskell Curry** and logician **William Alvin Howard**.

The Curry-Howard correspondence

In other words, the Curry–Howard correspondence is the observation that two families of formalisms which had seemed unrelated—namely, the proof systems on one hand, and the models of computation on the other—were, in the two examples considered by Curry and Howard, in fact structurally the same kind of objects.
A proof is a program, the formula it proves is a type for the program.

The Curry-Howard correspondence

More informally, this can be seen as an analogy that states that the return type of a function (i.e., the type of values returned by a function) is analogous to a logical theorem, subject to hypotheses corresponding to the types of the argument values passed to the function ; and that the program to compute that function is analogous to a proof of that theorem. This sets a form of logic programming on a rigorous foundation : *proofs can be represented as programs, and especially as lambda terms*, or *proofs can be run*.
Such typed lambda calculi derived from the Curry–Howard paradigm led to software like **Coq** in which proofs seen as programs can be formalized, checked, and run.

The Curry-Howard correspondence

Logic side	Programming side
implication	function type
conjuction	product type
disjunction	sum type
true formula	unit type
false formula	bottom type
assumption	variable
axioms	combinators
modus ponens	application

The Curry-Howard correspondence

If one restricts to the implicational intuitionistic fragment, a simple way to formalize logic in Hilbert's style is as follows. Let Γ be a finite collection of formulas, considered as hypotheses. We say that δ is derivable from Γ, and we write $\Gamma \vdash \delta$, in the following cases :

- δ is an hypothesis, i.e. it is a formula of Γ,
- δ is an instance of an axiom scheme ; i.e., under the most common axiom system :
- δ has the form $\alpha \to (\beta \to \alpha)$, or
- δ has the form $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$,
- δ follows by deduction, i.e., for some α, both $\alpha \to \delta$ and α are already derivable from Γ (this is the rule of modus ponens)

The Curry-Howard correspondence

The fact that the combinator **X** constitutes a **one-point basis** of (extensional) combinatory logic implies that the single axiom scheme

$$(((\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))) \to ((\delta \to (\epsilon \to \delta)) \to \zeta)) \to \zeta$$

which is the principal type of **X**, is an adequate replacement to the combination of the axiom schemes

$$\alpha \to (\beta \to \alpha)$$

and

$$(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma)),$$

Table of Contents

The Y combinator

The fixpoint combinator is a higher-order function that computes a fixed point of other functions : this is how **recursion** is implemented.

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$Y = SSK(S(K(SS(S(SSK))))K)$$

For call-by-value languages, an η expansion is necessary, resulting in the **Z combinator** :

$$Z = \lambda f.((\lambda x.f(\lambda y.(xx)y))(\lambda x.f(\lambda y.(xx)y)))$$

Recursion in OCaml

```
let facto f = function
    0 -> 1
  | x -> x * f (x - 1)
(* val facto : (int -> int) -> int -> int = <fun> *)

let rec fix f x = f (fix f) x
(* val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun> *)

(fix facto) 5
(* - : int = 120 *)
```

```
let fix f =
  (fun x -> f (x x))
  (fun x -> f (x x))
(* val fix : ('a -> 'a) -> 'a = <fun> *)

fix facto
(* Stack overflow during evaluation (looping recursion?). *)
```

Recursion in OCaml (2)

An η expansion is necessary to evaluate it :

```
let fix f =
  (fun x e -> f (x x) e)
  (fun x e -> f (x x) e)
(* val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun> *)
```

Type `'a` is an isomorphic type, supported by typing System F_ω.
OCaml needs option `-rectypes` to compile this.

Recursion in OCaml (3)

Using polymorphic variants instead of rectypes :

```
let fix f =
  (fun (`X x) -> f(x (`X x)))
  (`X(fun (`X x) y -> f(x (`X x))
      y))
```

The fixpoint combinator can of course be used as a λ :

```
(fun f ->
  (fun x e -> f (x x) e)
  (fun x e -> f (x x) e))
(fun f ->
  (function 0 -> 1
    | x -> x * f (x - 1)))
(* - : int -> int = <fun> *)
```

I fucking love C++

```cpp
typedef void(*f0)();
typedef void(*f)(f0);

int main()
{
  [](f x)
    {
      x((f0)x);
    } ([](f0 x)
      {
        ((f)x)(x);
      });
}

std::remove_if(std::find_if(list.begin(), list.end(),
                                [](t_element e) -> bool
                                  { return (EMPTY == e.pred; )}),
                    list.end(),
                    [](t_element e) -> bool
                      { return (0 > e.value; )});
```

Quines

A quine is a computer program which takes no input and produces a copy of its own source code as its only output.

Doesn't the OCaml fixpoint look very much like a quine ?

```
(fun s -> Printf.printf "%s%S", s s) "(fun s -> Printf.printf
    \"%s%S\", s s)"
```

```
(* Mc Carthy's angelic nondeterministic choice operator: *)
if (amb [(fun _ -> false); (fun _ -> true)]) then
  7
else failwith "failure"
(* equals 7 *)
```

```
let numbers =
  List.map (fun n -> (fun () -> n))
    [1;2;3;4;5]

let pyth () =
  let (a, b, c) =
    let i = amb numbers
      and j = amb numbers
      and k = amb numbers in
    if i * i + j * j = k * k then
      (i, j, k)
    else failwith "too bad"
  in Printf.printf "%d %d %d\n" a b c

let _ = toplevel pyth
(* 3 4 5 *)
```

That's it !