Introduction to functional programming with OCaml (part II)

P. Cousot

Structured types: pairs and tuples

Pairs

```ocaml
# 5, "abc";;
- : int * string = (5, "abc")
# fst (5, "abc");;
- : int = 5
# snd (5, "abc");;
- : string = "abc"
# snd 5, "abc";;
Error: This expression has type int but an expression was expected of type 'a * 'b
# snd (5, "abc"), true;;
- : string * bool = ("abc", true)
```

Tuples

```ocaml
# true, 1, 2.5, 'A', "abc";;
- : bool * int * float * char * string = (true, 1, 2.5, 'A', "abc")
# ((true, 1), 2.5, 'A'), "abc");
- : (bool * int) * float * char * string =
  ((true, 1), 2.5, 'A'), "abc")
# (true, 1, (2.5, ('A', "abc"))));
- : bool * (int * (float * (char * string))) =
  (true, 1, (2.5, ('A', "abc")));
# (true, 1, 2.5, 'A', "abc") = ((((true, 1), 2.5), 'A'), "abc");;
Error: This expression has type (((bool * int) * float) * char) * string
  but an expression was expected of type
  bool * int * float * char * string
# (true, 1, 2.5, 'A', "abc") = (true, (1, (2.5, ('A', "abc"))));
Error: This expression has type bool * (int * (float * (char * string)))
  but an expression was expected of type
  bool * int * float * char * string
```

# 5, "abc";;
- : int * string = (5, "abc")
# fst (5, "abc");;
- : int = 5
# snd (5, "abc");;
- : string = "abc"
# snd 5, "abc";;
Error: This expression has type int but an expression was expected of type 'a * 'b
# snd (5, "abc"), true;;
- : string * bool = ("abc", true)
# true, 1, 2.5, 'A', "abc";;
- : bool * int * float * char * string = (true, 1, 2.5, 'A', "abc")
# ((true, 1), 2.5, 'A'), "abc");
- : (bool * int) * float * char * string =
  ((true, 1), 2.5, 'A'), "abc")
# (true, 1, (2.5, ('A', "abc"))));
- : bool * (int * (float * (char * string))) =
  (true, 1, (2.5, ('A', "abc")));
# (true, 1, 2.5, 'A', "abc") = ((((true, 1), 2.5), 'A'), "abc");;
Error: This expression has type (((bool * int) * float) * char) * string
  but an expression was expected of type
  bool * int * float * char * string
# (true, 1, 2.5, 'A', "abc") = (true, (1, (2.5, ('A', "abc"))));
Error: This expression has type bool * (int * (float * (char * string)))
  but an expression was expected of type
  bool * int * float * char * string
Tuples as pairs

- Tuples of 2 elements are pairs (otherwise pairs of pairs of ...)

```ml
# fst (true, 1);;
- : bool = true
# snd (true, 1);;
- : int = 1

# fst (true, 1, 2.5);;
Error: This expression has type 'a * 'b * 'c
but an expression was expected of type 'd * 'e
# snd (true, 1, 2.5);;
Error: This expression has type 'a * 'b * 'c
but an expression was expected of type 'd * 'e
```

- No empty or 1-element tuple

```ml
# fst (true, 1, 1, 2.5);;
Error: This expression has type 'a * 'b * 'c
but an expression was expected of type 'd * 'e
# snd (true, 1, 1, 2.5);;
Error: This expression has type 'a * 'b * 'c
but an expression was expected of type 'd * 'e
```

Recursive higher-order first-class functions

```ml
# let x = 0;;
val x : int = 0
# (* function *)
let abs x = if x < 0 then -x else x;;
val abs : int -> int = <fun>
# abs (-17);;
- : int = 17
# abs 17;;
- : int = 17
# abs x;;
- : int = 0
```

Global function definitions

```
# (* variable *)
let x = 0;;
val x : int = 0
# (* function *)
let abs x = if x < 0 then -x else x;;
val abs : int -> int = <fun>
# abs (-17);;
- : int = 17
# abs 17;;
- : int = 17
# abs x;;
- : int = 0
```

Multiple parameters by currying

```
# let prod x y = x * y;;
val prod : int -> int -> int = <fun>
# prod 2 3;;
- : int = 6
# let prod' (x, y) = x * y;;
val prod' : int * int -> int = <fun>
# prod' (2, 3);;
- : int = 6
```

Functions have only one parameter but can return functions or take one tuple as parameter.

A function of many arguments can be written as a function that accepts one argument and returns a function to consume the remaining arguments. This transformation is known as currying. A curried function may then be partially applied to obtain a more specialised function.
Partial parameter binding

```ocaml
# let prod x y z = x * y * z;;
val prod : int -> int -> int -> int = <fun>
# let prod2 = prod 2;;
val prod2 : int -> int -> int = <fun>
# let prod2x4 = prod2 4;;
val prod2x4 : int -> int = <fun>
# prod2x4 5;;
- : int = 40
# let prod2x4x5 = prod2x4 5;;
val prod2x4x5 : int = 40
# prod2x4x5;;
- : int = 40
#
```

Anonymous function (lambda expression)

```ocaml
# function x -> x + 1;;
- : int -> int = <fun>
# (function x -> x + 1) 2;;
- : int = 3
# let x = function x -> x + 1;;
val x : int -> int = <fun>
# x 1;;
- : int = 2
# x x 1;;
Error: This function is applied to too many arguments;
maybe you forgot a `;`
#
```

Simultaneous function definitions

```ocaml
# let sqr x = x * x
     and cube x = x * x * x;;
val sqr : int -> int = <fun>
val cube : int -> int = <fun>
# sqr 5;;
- : int = 25
# cube 5;;
- : int = 125
```

Left to right parameter binding

```ocaml
f x y = ((f x) y)
```

Error: This function is applied to too many arguments;
maybe you forgot a `;`

Simultaneous function definitions (cont’d)

# let sqr' x = x * x
and cube' x = x * (sqr' x);;
Error: Unbound value sqr'
Did you mean sqrt?
# let rec sqr' x = x * x
and cube' x = x * (sqr' x);;
val sqr' : int -> int = <fun>
val cube' : int -> int = <fun>
# cube' 3;;
- : int = 27

With let, the function sqr’ is defined only at the end of the simultaneous definition. let rec allows recursion. The two cases have to be distinguished because of type inference.

Lexical scoping

% ocaml
OCaml version 4.01.0

# x
;;
Error: Unbound value x
# let x = 0;;
val x : int = 0
# let x = x + 1 in x;;
- : int = 1
# x;;
- : int = 0
#

Local functions

# let cube' x = let sqr' x = x * x in x * (sqr' x);;
val cube' : int -> int = <fun>
# cube' 5;;
- : int = 125
# sqr' 5;;
Error: Unbound value sqr'
#

sqr' is visible only within the let definition, after in:

Recursion

# let prod' x y = if x = 0 then 0
  else (prod' (x - 1) y) + y;;  undefined
Error: Unbound value prod'
# let rec prod' x y = if x = 0 then 0
  else (prod' (x - 1) y) + y;;
val prod' : int -> int -> int = <fun>
# prod' 3 5;;
- : int = 15
# let prod' x y = if x = 0 then 0
  else (prod' (x - 1) y) + y;;
val prod' : int -> int -> int = <fun>
# prod' 3 5;;
- : int = 15
#
Tracing execution of functions

```
# let rec fib x = if x <= 1 then 1 else fib (x - 1) + fib (x - 2);;
val fib : int -> int = <fun>
# trace fib;
fib is now traced.
# fib 3;;
fib <-- 3
fib --> 1
fib <-- 1
fib --> 2
fib <-- 0
fib --> 1
fib <-- 1
fib --> 2
fib --> 3
- : int = 3
#
```

Using libraries with lexical scoping and block structure

```
# open Int32;;
# mul;;
- : int32 -> int32 -> int32 = <fun>
# (* non-recursive definition using the Int32 library *)
let mul x y = if x = zero then zero
  else (add (mul (pred x) y) y);
val mul : int32 -> int32 -> int32 = <fun>
# let x = of_int 3;;
val x : int32 = 3l
# mul x x;;
- : int32 = 9l
# (* recursive definition *)
let rec mul x y = if x = zero then zero
  else (add (mul (pred x) y) y);
val mul : int32 -> int32 -> int32 = <fun>
```

Lexical scoping with block structure

```
# let f x = 0;;
val f : 'a -> int = <fun>
# let f x = f x;;
val f : 'a -> int = <fun>
# f 17;;
- : int = 0
# let rec f x = f x;;
val f : 'a -> 'b = <fun>
# f 17;;
^CInterrupted.
# (* looping, interrupted by ctrl+C *)
```

Lexical scoping

- Scope of a variable: set of program points where it is visible (can be used)
- Scope of x in
  
    let x = e;;

  is the following instructions
- Scope of x in
  
    let x = e in e’;;

  is e and e’
Higher-order functions

# epsilon_float;;
- : float = 2.22044604925031308e-16
# (* derivative *)
let d f x =
  let dx = sqrt epsilon_float in
  (f(x +. dx) -. f(x -. dx)) /. (2. *. dx);;
val d : (float -> float) -> float -> float = <fun>
# (* function *)
let g x = x ** 3. -. x -. 1.;;
val g : float -> float = <fun>
# g 3.0;;
- : float = 23.
# (* derivatives of function *)
let g' = d g;;
val g' : float -> float = <fun>
# g' 3.0;;
# let g'' = d g';;
val g'' : float -> float = <fun>
# g'' 2.0;;
- : float = 12.

Functionals (taking functions as parameters)

# (* iter f a n = (f n (f (n-1) (... (f 1 a)...))) *)
let rec iter f a n = if n = 0 then a else
  (f n (iter f a (n - 1)));
val iter : (int -> 'a -> 'a) -> 'a -> int -> 'a = <fun>
# let sum = iter (+) 0;;
val sum : int -> int = <fun>
# sum 4;;
- : int = 10
# let fact = iter ( * ) 1;;
val fact : int -> int = <fun>
# fact 4;;
- : int = 24
# let invfact = iter (function x -> (function y -> y /. (float_of_int x))) 1.0;;
val invfact : int -> float = <fun>
# invfact 2;;
- : float = 0.5
# invfact 3 (* 1/n! *);;
- : float = 0.166666666666666657
#

Polymorphic functions

# let id x = x;;
val id : 'a -> 'a = <fun>
# id 3;;
- : int = 3
# id true;;
- : bool = true
# id "abc";;
- : string = "abc"
# id;;
- : 'a -> 'a = <fun>
#

Correctness proof by recurrence

iter f a 0
  = if 0 = 0 then a else ...
  = a

iter f a 1
  = if 1 = 0 then a else (f 1 (iter f a 0))
  = (f 1 a)

iter f a (n+1) assuming n > 0
  = if (n+1) = 0 then a else (f (n+1) (iter f a ((n+1) - 1)))
  = (f (n+1) (iter f a n))
  = (f (n+1) (f n (f (n-1) (... (f 1 a)...))))  ind. hyp.

proving by recurrence on n >= 0 that
iter f a n = (f n (f (n-1) (... (f 1 a)...)))
Local procedures as parameters

C pointer functions versus OCaml higher-order function

- In C you can pass and return pointers to functions
- However, in C these functions cannot refer to local computations as is the case in OCaml, e.g.

```ocaml
# let f x = let x2 = x * x in fun y -> x2 + y * y;;
val f : int -> int -> int = <fun>
# f 1 2;;
- : int = 5
# f 2 3;;
- : int = 13
# let g1 = f 1;;
val g1 : int -> int = <fun>
# g1 2;;
- : int = 5
# let g2 = f 2;;
val g2 : int -> int = <fun>
# g2 3;;
- : int = 13
```

And the result is

```
9
89
789
6789
56789
456789
3456789
23456789
123456789
- : unit = ()
```

Passing local procedures as parameters

What is the result of executing the following recursive procedure?

```ocaml
# let rec p x q =  
  let r () = print_int x; q () in 
  if x > 0 then
    (r (); p (x-1) r)
  else
    ()
  in
  p 9 print_newline;;
```

A new local procedure is created at each call of p printing the current value of x and then recursively calling all previously declared procedures.
Undecidability results

- Theorem: It is possible to simulate a Turing machine using only Booleans [1]
- Corollary I: The invariance proof in Hoare logic is undecidable for languages with static scoping, recursion and passing of local procedures as parameters even if the program uses only Boolean (or finite data types) [2]
- Corollary II: Model-checking is undecidable for OCaml, Pascal, etc. even for purely Boolean programs


Intuition for undecidability

- Intuition for these results:
  1. An infinite tape can be simulated on the (recursion) stack
  2. Each cell on the tape is a local variable inside the stack which can be globally read or written by calling local procedures passed as parameters

The End,
Thank You