CS3110 Spring 2017 Lecture 24: Computability Theory in OCaml

Robert Constable

1 Topics

(1) Review of fixed point operators fix, $efix$.

(2) A hierarchy of computable functions.

(3) Fixed point operators on type constructing functions – lazy fixed points, fix and eager ones $efix$.

(4) Provably unsolvable OCaml tasks – detecting non-termination in partial types.

2 Fixed point operators on functionals

Below is a typical OCaml recursive definition of a function, showing its typing as well. We can also write it without the types as shown just below the typed definition. In this case the integer square root function that we derived from a constructive proof of the following theorem from Lecture 9.

$$\forall n : nat.\exists r : nat.(r^2 \leq n < (r+1)^2).$$

We defined the following OCaml recursive function to compute this value on the natural numbers. In OCaml we need to use the type *int* since *nat* is not a primitive OCaml type.

```
let rec sqrt (n : int) : int =
     if n <= 0 then 0
        else let r = sqrt(n-1) in
        if (r+1)*(r+1) <= n then r+1 else r
```

Here is a definition without the explicit types since these can be inferred by the type checker.

```
let rec sqrt n =
     if n <= 0 then 0
        else let r = sqrt(n-1) in
        if (r+1)*(r+1) <= n then r+1 else r
```

Here is an actual OCaml session using the above definition. It is followed by definitions of addition, multiplication, exponentiation and hyper exponentiation in OCaml.

```
# let rec sqrt n = if n<= 0 then 0 else let r = sqrt (n-1) in
  if (r+1)*(r+1) <= n then r+1 else r ;;
val sqrt : int -> int = <fun>
# sqrt 9 ;;
```

```
- : int = 3
# sqrt 17 ;;
- : int = 4
```

This function computes the integer square root of a non-negative integer, e.g. a natural number n in the *mathematical type* $\mathbb{N} = \{0, 1, 2, ...\}$. To fully understand how this standard OCaml definition "works" it is necessary to know about the OCaml implementation of *let rec*. In this lecture we show another way to understand recursion by defining two simple operations on functions. The operations are called *fix* and *efix*. The first operation works when the computation system uses *lazy evaluation* of functions, e.g. $f\ a$ for f a function, the argument is not evaluated before the function f is applied. So in this example, $(fun\ x \rightarrow (fun\ y \rightarrow x + y))(2 + 3)$ reduces in one step to the value $(fun\ y \rightarrow (2 + 3) + y$.

The operator *efix* provides eager evaluation of the function's input, so the *e* is for *eager*. OCaml uses eager evaluation, thus $(fun\ x \rightarrow x + x)(2 + 3)$ evaluates to 10 by first evaluating $2 + 3$ to 5 and then adding $5 + 5$.

We can understand this function in a particularly interesting way if we generalize it to a functional. Consider this function:

```
fun f -> fun n -> if n <= 0 then 0
     else   let r = f(n-1) in
     if (r+1)*(r+1) <= n then r+1 else r
```

This function has two inputs, the first is a function, f, the second a number, n. This looks nicer with λ-notation:

$$\begin{aligned}\lambda(f.\lambda(n.\ &\textbf{if}\ n \leq 0\ \textbf{then}\ 0\\ &\textbf{else if}\ f(n-1) * f(n-1) \leq n\ \textbf{then}\ f(n-1) + 1\\ &\textbf{else}\ f(n-1)))\end{aligned}$$

The type of function is `(nat → nat) → nat → nat`[1], which is the same as `(nat → nat) → (nat → nat)`. This kind of function is called a *functional*. We'll use a capital F to denote it:

`F : (nat → nat) → (nat → nat)`

so `F(f) ∈ nat → nat`, if `f ∈ nat → nat`.

These functionals are a natural way to compute and to understand recursive functions. Below we explain them "operationally" and "denotationally."

[1] We use the type `nat` here although OCaml does not have this type.

3 A hierarchy of primitive recursive functions

$a_0(x,y) = x+1$ $\quad$ $s(x) = x+1$ $\quad$ so $\quad$ $a_0 = s(x)$

$a_1(x,y) = add(x,y)$

$$\boxed{\begin{array}{l} add(0,y) = y \\ add(s(x),y) = s(add(x,y)) \end{array}}$$

note $\quad s(add(x,y) = a_0(a_1(x,y),y)$

$a_2(x,y) = mult(x,y)$

$$\boxed{\begin{array}{l} mult(0,y) = 0 \\ mult(s(x),y) = add(mult(x,y),y) \end{array}}$$

i.e. $(x+1)\cdot y = x\cdot y + y$
note $\quad add(mult(x,y),y) = a_1(a_2(x,y),y)$

$a_3(x,y) = exp(x,y)$

$$\boxed{\begin{array}{l} exp(0,y) = 1 \qquad \text{i.e. } y^0 = 1 \\ exp(s(x),y) = mult(exp(x,y),y) \end{array}}$$

i.e. $y^{x+1} = y^x \cdot y$
note $\quad mult(exp(x,y),y) = a_2(a_3(x,y),y)$

$a_4(x,y) = hypexp(x,y)$

$$\boxed{\begin{array}{l} hypexp(0,y) = y \\ hypexp(s(x),y) = exp(hypexp(x,y),y) \end{array}}$$

note $\quad exp(hypexp(x,y),y) = a_3(a_4(x,y),y)$

$a_{n+1}(x,y)$

$$\boxed{\begin{array}{l} a_{n+1}(0,y) = y \\ a_{n+1}(s(x),y) = a_n(a_{n+1}(x,y),y) \end{array}}$$

$a_n(x,y)$ can be thought of as a function of n,x,y. This is *Ackerman's function* in one form. It is not primitive recursive.

3.1 Example of primitive recursion

The typical recursive definition of addition on the natural numbers using successor is the following.

```
# let rec sqrt n = if n<= 0 then 0 else let r = sqrt (n-1) in
  if (r+1)*(r+1) <= n then r+1 else r ;;
val sqrt : int -> int = <fun>
# sqrt 9 ;;
- : int = 3
# sqrt 17 ;;
- : int = 4

# let rec add x y = if x = 0 then y else succ (add (x-1) y) ;;
val add : int -> int -> int = <fun>
# add 5 8 ;;
- : int = 13
```

```
# let rec mult x y = if x = 0 then 0 else add (mult (x-1) y) y ;;
val mult : int -> int -> int = <fun>
# mult 5 7 ;;
- : int = 35

# let rec exp x y = if x = 0 then 1 else mult (exp (x-1) y) y ;;
val exp : int -> int -> int = <fun>
# exp 5 3 ;;
- : int = 243
# exp 2 3 ;;
- : int = 9

# let rec hypexp x y = if x = 0 then y else exp (hypexp (x-1) y) y ;;
val hypexp : int -> int -> int = <fun>
# hypexp 2 3
Stack overflow during evaluation (looping recursion?).
```

We can write all of these functions using efix. Here are two examples of this process.

```
# efix (fun f p -> if fst p = 0 then snd p else succ (f ((fst(p)-1), snd p)) ) (0,1) ;;
- : int = 1

# efix (fun f p -> if fst p = 0 then snd p else succ (f ((fst(p)-1), snd p)) ) (4,6) ;;
- : int = 10
#

# efix (fun f p -> if fst p = 0 then snd p else succ (f ((fst(p)-1), snd p)) ) (0,1) ;;
- : int = 1

# efix (fun f p -> if fst p = 0 then snd p else succ (f ((fst(p)-1), snd p)) ) (4,6) ;;
- : int = 10

# let addf = efix (fun f p -> if fst p = 0 then snd p else succ (f ((fst(p)-1), snd p)) ) ;;
val addf : int * int -> int = <fun>

# efix (fun mult p -> if fst p = 0 then 0 else addf (mult ((fst(p)-1), snd p), snd p) ) ;;
- : int * int -> int = <fun>

# let multf = efix (fun mult p -> if fst p = 0 then 0 else addf (mult ((fst(p)-1), snd p), snd p) ) ;;
val multf : int * int -> int = <fun>
# multf (4,6) ;;
- : int = 24
#
```