CS3110 Spring 2016 Lecture 20: Fixed Point Operators

Robert Constable

Only four more lectures on new material plus one guest and two course summary/exam preparation lectures.

Topics

1. Plans for remaining seven lectures 20-26.
2. Fixed point operators on *functionals*: fix, $efix$.
3. Fixed point operators on type constructing functions – lazy fixed points, fix.
4. Provably unsolvable OCaml tasks.

1 Plans for remaining lectures

Tues.	April 19	Lecture 20	Recursion with fix, $efix$, evaluators, halting.
Thurs.	April 21	Lecture 21	Asynchronous computation protocols.
Tues.	April 26	Lecture 22	Asynchronous protocols – consensus.
Thurs.	April 28	Lecture 23	Specifying protocols.
Tues.	May 3	Lecture 24	Guest lecture by Yaron Minskey from Jane Street.
Thurs.	May 5	Lecture 25	Course summary.
Tues.	May 10	Lecture 26 (Last)	Final exam prep.
Tues.	**May 17**	**Final exam 9:00-11:00 am. Uris Hall G01**	

2 Fixed point operators on functionals

In Lecture 19 we briefly discussed fixed point operators in relating OCaml to type theory. We mentioned the idea of *co-lists* and *co-trees*. These are examples of co-recursive types. They can be used to define recursive types, but OCaml does not allow co-inductive types.

The co-lists and co-trees were defined using a fixed point operator, fix. We will see that this operator provides an excellent account of recursion. This account does not require discussing how compilers implement recursion using "stacks." The account also reveals that there are at least two kinds of "fixed point" operators.

Let us look at a recursive function we studied carefully, the integer square root from Lecture 9 on March 1.

```
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
```

This computes the integer square root of a non-negative integer, e.g. a natural number n in the *mathematical type* $\mathbb{N} = \{0, 1, 2, ...\}$.

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:

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

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

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

`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."

3 Fix

(a.) Operational

Operationally, we define an operation called fix that captures the way we compute with recursive functions.

Given $\lambda(f.\lambda(x.\text{body}(f,x)))$, we compute $fix(\lambda(f.\lambda(x.\text{body}(f,x))))$ by one step to $\lambda(x.\text{body}(fix(\ ),x))$. We use $fix(\ )$ to abbreviate the whole expression $fix(\lambda(f.\lambda(x.body(f,x))))$.

We are now able to apply $\lambda(x.\_\_)$ to a value, say

0	to get	0
1	to get	$\textbf{if } fix(\ )\,0 * fix(\ )\,0 \leq n \textbf{ then } (fix(\ )\,0)+1 \textbf{ else } fix(\ )\,0$

(b.) Denotational

Another way to understand the fix operator is to look at properties of the computable functional `F : (nat → nat) → (nat → nat)`.

What if we apply F to a function that is everywhere undefined, say f_0 such that $f_0(n)$ is *undefined for every* n?

Can we say anything about $F(f_0)$? It reduces to

$$\lambda(n.\ \textbf{if } n \leq 0 \textbf{ then } 0 \textbf{ else if } f_0(n-1) \times f_0(n-1) \ldots.)\,0$$

This application of the value of the functional produces the value 0 when applied to 0.

Call this function f_1 and notice that it extends f_0.

$f_0(n)$	$=$	undefined for all n
$f_1(0)$	$=$	0
$f_1(1)$	$=$	undefined, as are $f_1(n),\ n > 0$

What happens to $fix(\ )\,f_1$?

In the standard account of the undefined applications, we say $f_0(0) = \perp,\ f_0(1) = \perp, \ldots,\ f_0(n) = \perp$ for all n. We think of $\perp$ as an "undefined value." We can consider it to be a "diverging value," something *less defined than any numerical output.*

As we apply F to these "partial functions," we created "slightly less undefined functions." In the limit, we "converge" to the integer square root function.

Consider the sequence of applications

$$\begin{array}{lcl} f_0(x) & = & \bot \text{ for all } x \\ f_1(x) & = & F(f_0)(x) \\ f_2(x) & = & F(f_1)(x) \\ \vdots & & \\ f_n(x) & = & F(f_{n-1})(x). \end{array}$$

In the "limit" the function defined, call it f_ω, is the integer square root function. So we have seen two ways to explain the meaning of recursively defined functions. We can prove that these lead to the same computable function.

We will adopt a simple computation rule for fix to explain recursion. For this to work, we need that function application is *lazy.* So this approach does not explain OCaml recursive functions. To do that we need an *eager fixed point operator.*

4 An eager fixed point operator – *efix*

```
let rec efix f x = f (efix f) x
```

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

$(($`nat`$\to$ `nat`$) \to$ `nat` $\to$ `nat`$) \to ($`nat` $\to$ `nat`$)$

$$(\underbrace{(\alpha \to \beta)}_{f} \to \underbrace{\alpha}_{x} \to \underbrace{\beta}_{x}) \to (\texttt{nat} \to \texttt{nat})$$

vs

```
let rec fix f = f (fix f)
```

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

$(($`nat` $\to$ `nat`$) \to ($`nat` $\to$ `nat`$)) \to ($`nat` $\to$ `nat`$)$

$$\begin{array}{l} f\_rt = \lambda(f.\lambda(n.\ \textbf{if}\ n = 0\ \textbf{then}\ 0 \\ \qquad\qquad \textbf{else let}\ r = f(n-1)\ \textbf{in} \\ \qquad\qquad \textbf{if}\ (r+1)*(r+1) \le n\ \textbf{then}\ r+1\ \textbf{else}\ r)) \end{array}$$

Abbreviate using

$$\begin{array}{l} body(n,f) == \textbf{if}\ n = 0\ \textbf{then}\ 0 \\ \qquad\qquad \textbf{else let}\ r = f(n-1)\ \textbf{in} \\ \qquad\qquad \textbf{if}\ (r+1)^2 \le n\ \textbf{then}\ r+1\ \textbf{else}\ r \end{array}$$

$$\begin{aligned} sqrt2 &= efix(f\_rt)\ \ x \qquad\qquad \text{expand } \textit{efix} \\ &= f\_rt(efix\ \ f\_rt)\ \ x \end{aligned}$$

Substitute definition of f_rt using $\textit{body}(n, f)$ abbreviation to get $f\_rt(efix\ \ f\_rt)$.

$$\lambda(f.\lambda(n.\textit{body}(n, f)))(efix(f\_rt))\ \ x$$

Substitute $efix(f\_rt)$ for f in function application.

$$\lambda(n.\textit{body}(n, efix(f\_rt)))\ \ x$$

Expand *body* and substitute x for n.

```
**if** $x = 0$ **then** $0$
**else let** $r = efix(f\_rt)\ \ x - 1$ **in**
    **if** $(r+1)^2 \leq n$ **then** $r + 1$ **else** $r$
```

We now see the computational pattern as we reduce $x - 1$ to $x - 2$ to $x - 3$ ...until we reach 0.

$$efix(\lambda(f.\lambda(n.\text{body}(n, f))))\, x = (\lambda(f.\lambda(n.\text{body}(n, f))(efix(f\_rt)))\, x$$

$$\begin{aligned} & efix(\lambda(f.\lambda(n.\text{body}(n, f))))\, exp \\ & \quad = (\lambda(f.\lambda(n.\text{body}(n, f))(efix(f\_rt)))\, exp \qquad \text{e.g. } exp \downarrow 17. \\ & \qquad\qquad \lambda(n.\text{body}(n, efix(f\_rt)))\, 17 \end{aligned}$$

$$\begin{aligned} & fix(\lambda(f.\lambda(n.\text{body}(n, f))))\, exp \\ & \qquad\qquad \lambda(n.\text{body}(n, fix(\ )))\ exp \downarrow \\ & \qquad\qquad\quad \text{body}(exp, fix(\ )) \\ & \qquad\qquad\quad \textbf{if } exp = 0 \textbf{ then } 0 \\ & \qquad\qquad\quad \textbf{else } r = f\ exp - 1 \end{aligned}$$

The implementation of this in OCaml can be found on the last page.

The halting problem using fix

Suppose there were a halting detector, say $h : \texttt{unit} \rightarrow \texttt{bool}$ such that $h(x) = \text{true}$ iff $x \downarrow$, i.e. $h(x)$ has value true if x converges to ( ), the canonical element of unit.

For example:

```
let rec count(n) =
   if n = 0 then ( )
   else count(n-1)
```

where

$$\begin{array}{rcl} h((\,)) & = & \text{true} \\ h(count(-1)) & = & \perp \\ h(count(0)) & = & (\,). \end{array}$$

Define $d = fix(\lambda(x.\ \textbf{if}\ h(x)\ \textbf{then}\ \perp\ \textbf{else}\ (\,)))$.

What is the value of $h(d)$?

If $h(d) = t$, then d converges, but by definition if $h(d)$ =true, then d diverges! If $h(d) =$ false, then by its definition, d converges.

We conclude that there is no halting function.

Logic

Is this expression programmable?
If so, how many different programs can be written for it?

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

With a small adjustment we can write an expression that can produce an unbounded number of programs:

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

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

# let frt = 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 ;;
val frt : (int -> int) -> int -> int = <fun>
#

# let sqrt x = efix frt x ;;
val sqrt : int -> int = <fun>
# sqrt 17 ;;
- : int = 4
# sqrt 101 ;;
- : int = 10

# let rec fix f = f (fix f) ;;
val fix : ('a -> 'a) -> 'a = <fun>
# let div = fix (fun x ->x) ;;

 let rec fix f = f (fix f) ;;
val fix : ('a -> 'a) -> 'a = <fun>
# let div = fix (fun x -> x) ;;
Stack overflow during evaluation (looping recursion?).
#

# let h = (fun x -> (if x = () then true else false) ) ;;
val h : unit -> bool = <fun>
#
```