CS3110 Spring 2017 Lecture 23: Fixed Point Operators

Robert Constable

1 Topics

1. Plans for remaining two lectures 24 and 25.
2. Message Sequence Diagrams provide examples of free choice sequences.
3. Fixed point operators on *functionals*: fix, $efix$.
4. Fixed point operators on type constructing functions – lazy fixed points, fix.
5. Provably unsolvable OCaml tasks.

2 Message Sequence Diagrams and Free Choice

One reason that the PRL research group at Cornell adopted Brouwer's notion of free choice sequences and is implementing them is that we learned from working with the distributed systems group, Ken Birman and Robbert van Renesse in particular, that the sequence of messages that we see in the message sequence diagrams is unpredictable. They arise as if they were generated by free choices of the participating processes. It is impossible in practice to predict of understand in detail why messages arrive in a particular order. They do not follow a "hidden random process," so it is not correct to think of them as random.

It took years of working with these processes to appreciate that they constitute a new element in computing theory. This is clear in our basic articles on event logic [1] and distributed processes [2]. It is remarkable in some ways that insights about distributed computing can have an impact

on constructive real analysis, yet it is an idea that can be grasped even in an undergraduate programming language course that treats both functional processes and the constructive real numbers. Brouwer's result that all computable functions on a closed interval of the real line into the real line are uniformly continuous shows this connection and illustrates the interconnectedness of concepts in computing theory.

3 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 session using the above definition.

```
# 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, $(fun\ x \to (fun\ y \to x + y))(2 + 3)$ reduces in one step to the value $(fun\ y \to (2 + 3) + y$.

The operator *efix* provides eager evaluation of the function's input, so the e is for *eager*. OCaml uses eager evaluation, thus $(fun\ x \to x + x)(2 + 3)$ 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:

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

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`.

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

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

4 Fix

(a.) Operational

Operationally, we define a simple explicit operation called fix that defines a fully transparent way to define recursive functions.

Given $\lambda(f.\lambda(x.\text{body}(f,x)))$, we reduce $fix(\lambda(f.\lambda(x.\text{body}(f,x))))$ by one step to $\lambda(x.\text{body}(fix(\,),x))$. This step substitutes the fix expression for the function variable f. 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 **if** $fix(\,)\,0 * fix(\,)\,0 \leq n$ **then** $(fix(\,)\,0)+1$ **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

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

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 $f_1(n),\ n > 0$

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*.

5 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 → nat) → nat → nat) → (nat → 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 → nat) → (nat → nat)) → (nat → nat)`

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

Abbreviate using

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

$$\begin{array}{lll} sqrt2 = efix(f\_rt)\ \ x & & \text{expand } efix \\ \quad = f\_rt(efix\ \ f\_rt)\ \ x & & \end{array}$$

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

$$\lambda(f.\lambda(n.body(n, f)))(efix(f\_rt)) \;\; x$$

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

$$\lambda(n.body(n, efix(f\_rt))) \;\; x$$

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.

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

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

$fix(\lambda(f.\lambda(n.\text{body}(n, f)))) \, exp$
$\qquad\quad \lambda(n.\text{body}(n, fix(\;))) \; exp \downarrow$
$\qquad\qquad \text{body}(exp, fix(\;))$
$\qquad\qquad$ **if** $exp = 0$ **then** 0
$\qquad\qquad$ **else** $r = f \; exp - 1$

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 : `unit` $\rightarrow$ `bool` such that $h(x) =$ 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

$$\begin{aligned} h(( )) &= \text{true} \\ h(count(-1)) &= \bot \\ h(count(0)) &= ( ). \end{aligned}$$

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

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.

```
# 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>
#
```

References

[1] Mark Bickford, Robert Constable, and David Guaspari. Constructing event structures over a general process model. Department of Computer Science TR1813-23562, Cornell University, Ithaca, NY, 2011.

[2] Robert L. Constable. Effectively nonblocking consensus procedures can execute forever: a constructive version of flp. Technical Report Tech

Report 11512, Cornell University, 2008.