# Lecture #4: Fixed Points

September 19, 2007

## 1 Today

What is a fixed point? How can we find a fixed point? Today's lecture covers almost all of Chapter 5. It is probably the most mathematically involved of the entire semester, but it's necessary to understand what we mean when we write 'recursive definitions,' which we'll do a lot of later in this course.

## 2 Recursive Definitions

We start with an example of a 'recursive definition'

$$x = \ldots x \ldots$$

or mutually-recursive definitions:

$$\begin{aligned} x_1 &= < \ldots x_1 \ldots x_n \ldots > \\ x_n &= < \ldots x_1 \ldots x_n \ldots > \end{aligned}$$

It turns out that we can re-interpret mutiply-recursive definitions as singly-recursive versions (see the text, for details).

Recursion may seem intuitive to you, but mathematically, it's not obvious how it's well-defined. What kinds of values can $x$ take on? What kind of domains do we want to consider as the domains of possible solutions? Our task today will be to answer these kinds of questions, and to bring equations like this into the realm of 'mathematical decency.'

But the domain of $x$ can influence the number and existence of solutions:

$$x = \frac{1}{16}x^3$$

has 1 solution if $x$ is in the integers, two if $x$ is rational, 4 among the complex numbers, etc.

### 2.1 Graphs of a function

Here is a recursive definition in the lambda calculus:

$$f = \lambda n.\text{if } (n = 0) \text{ then } 0 \text{ else } (2 + (f(n-1)))$$

where $f : \text{Nat} \to \text{Nat}$

We can write out the graph of this function, which we can define by starting with $f(0)$ and working upwards from there, using values we've already computed.

$$f = \{< 0, 0 >, < 1, 2 >, < 2, 4 >, < 3, 6 > \ldots\}$$

Obviously, this is the doubling function.

Now! We write down a 'generating function', $g$, which is a function that takes a function, and returns a function.

$$g = \lambda f.(\lambda n.\text{if } (n = 0) \text{ then } 0 \text{ else } (2 + (f(n-1))))$$

What is the domain of this 'generating function?' $g : (\text{Nat} \to \text{Nat}) \to (\text{Nat} \to \text{Nat})$. Now, let's apply $g$ to some other functions. For instance, what happens when we apply $g$ to the identity function?

$$(g\,(\lambda n.n)) = \lambda n.\text{if } (n = 0) \text{ then } 0 \text{ else } (n + 1)$$

What happens when we apply $g$ to $f$, the doubling function from above?

$$(g\,f) = \lambda n.\text{if } (n = 0) \text{ then } 0 \text{ else } (2 + (f(n-1)) = f$$

In other words, $(g\,f) = f$, which is to say that $f$ (the double function) is a 'fixed point' of $g$. A 'fixed point' of a function $f : \text{D} \to \text{D}$ is $x : \text{D}$ such that $(f\,x) = x$.

Our claim is going to be that we can rewrite any recursive definition in terms of a generating function. Then, when we find a fixed point of that generating function, we will have solved the original definition. Therefore, 'finding fixed points' is the basis of giving a sound mathematical basis to our recursive definitions.

Dave says: "It even gets better. There's something called the Fixed Point Operator, that when applied to a function g, will give the fixed point of $g$. We will write down, at the end of class today, the 'code' for the Fixed Point Operator which will compute the fixed point of any generating function we give it." In other words, to do recursion, we don't need recursive definitions – we just require the ability to write the fixed point operator, and that can lead to (among other things) non-termination.

How is this defined? Can we find the fixed point of any function, or are there constraints we must face?

## 3 Some Setup

First, we define a partial order $\sqsubseteq$that relates values based on 'information content.' The expression $a \sqsubseteq b$will mean that $a$ is 'weaker than' $b$. We will also

define a symbol, $\perp$ ('bottom'), that will signify something that contains no information.

$$\forall x. \perp \sqsubseteq x$$

A 'lifted' domain is one that has 'bottom' added to it, in such a way that bottom is less than everything in the original domain. Remember that our ordering is 'partial' – that is to say, there is not necessarily a relationship between every pair of elements in the domain.

## 3.1 Partial Orders on Compound Domains

Assume that we start with 'lifted' primitive domains. How do we define partial orders on the compound domains, built with our compound domain combinators?

$$\begin{array}{ll} A \times B & < a, b > \sqsubseteq < c, d > iff\, a \sqsubseteq c \, and \, b \sqsubseteq d \\ A + B & (A \rightarrow (A + B)\, a) \sqsubseteq (A \rightarrow (A + B)\, c)\, iff\, a \sqsubseteq c \\ A \rightarrow B & f_1 \sqsubseteq f_2\, iff\, \forall a \in A.(f_1\, a) \sqsubseteq (f_2\, a) \end{array}$$

So, now that we've added bottom to our primitive domains, we get a partial order on all our compound domains. We will do this because we're going to define 'least' fixed points, which will be (it turns out) what your computer computes.

## 3.2 Least Upper Bounds

When we're working with domains that have this partial ordering defined for them, we can define something called the 'least upper bound'. The Least Upper Bound (LUB) of $X \subseteq D$is $a \in D$ such that $\forall x \in X.x \sqsubseteq a$ and a is 'weaker' than all other upper bounds of $X$. It's completely possible that the LUB doesn't exist – take, for instance, the case where there are two upper bounds of $X$ that aren't, themselves, comparable.

More terminology: a 'chain' is a totally ordered subset of a partial order (that is, every pair of elements in the chain is related by the partial ordering). A 'complete partial order' is a partially ordered set that has a least-upper-bound for every chain, and a least element (which is weaker than every other element). (That is to say, a complete partial order is 'pointed', i.e., it contains bottom). The abbreviation "CPO" stands for a complete partial order.

CPO Requirements:

- $A \times B$ is a CPO if both $A$ and $B$ are CPOs.
- $A + B$ is only a CPO if it's lifted explicitly (that is to say, $(A + B)_\perp$)
- $A \rightarrow B$ is a CPO if $B$ is a CPO.

And a monotonic function: "$f$ is monotonic" if $d_1 \sqsubseteq d_2 \rightarrow f(d_1) \sqsubseteq f(d_2)$.

### 3.3 Continuous Functions

Why are we talking about least upper bounds? Let's define a function, $halt : \mathrm{Nat}_\perp \to \mathrm{Nat}_\perp$.

$$halt = \lambda x.\mathrm{if}\,(x = \perp)\,then\,1\,else\,2$$

So if, $\perp \sqsubseteq 1$then $(halt\,\perp) \sqsubseteq (halt\,1)$. But we know this isn't the case. This is the reason we restrict our attention to CPOs and monotonic functions – it requires that our functions aren't 'making things up.'

We define a 'continuous' function: "$f : D \to E$ is continuous" if and only if $\sqcup_E\{(f\,c)|c \in C\} = f(\sqcup_D C)$, for $C \subseteq D$. A continuous function is always monotonic.

Let's work out an example.

$$even0 = \lambda n.if\,(n = 0)\,then\,0\,else\,(even0\,(n\,rem\,2))$$

And the generating function for this recursive definition:

$$g = \lambda f.\lambda n.if\,(n = 0)\,then\,0\,else\,(f\,(n\,rem\,2))$$

What is the fixed-point of $g$? Can we describe the graph of a function that is a fixed point of $g$?

$$fp_1 = \{< 0, 0 >, < 1, \perp >, < 2, 0 >, < 4, 0 >, \ldots\}$$

If we just omit those pairs that 'return' bottom, which is now implied, we can write this

$$fp_1 = \{< 2n, 0 >\}$$

Therefore,

$$(g\,fp_1) = fp_1$$

since $fp_1$ is a fixed point of $g$. But let's imagine that someone comes along, and shows you another fixed point:

$$fp_2 = \{< 2n, 0 >, < 2n + 1, 17 >\}$$

That is, it returns 17 where the other function didn't halt. Well, obviously, $(g\,fp_2) = fp_2$, since it is, of course, a fixed point as well. Mathematically, $fp_2$ is just as valid a fixed point as $fp_1$, although 'we don't like it', because we intuitively feel like $fp_2$ is "too creative." We're making stuff up, and it's not "what our computer would do," etc.

But it turns out, $fp_1 \sqsubseteq fp_2$– that is to say that our fixed point is weaker than the second fixed point. Our fixed point operator will be the 'least' fixed point operator – it will return the fixed point that is less than all other fixed points. This is what our computer will calculate, the least fixed-point.

# 4 Least Fixed Points

## 4.1 Mathematical Definition

We define something called the "Least Fixed Point Theorem".

If D is a pointed CPO, then for a continuous function $f : D \to D$,

$$(\text{fix}_D\, f) = \sqcup_D \{f^n \bot_D \,|\, n \geq 0\}$$

(This is a series of functions, the first that returns bottom for every input, the second that returns bottom if it tries to call itself recursively, the third that returns bottom if it tries to call itself recursively twice, etc.)

Proof:

$$\begin{aligned}
\bot_D &\sqsubseteq f(\bot_D) && \text{(Last element)} \\
f^n(\bot_D) &\sqsubseteq f^{n+1}(\bot_D) && \text{(Monotonicity)} \\
\{f^n(\bot_D) \,|\, n \geq 0\} & && \text{(Chain)} \\
f(\text{fix}_D\, f) &= && \\
& && f(\sqcup_D\{f^n(\bot_D)|n \geq 0\}) \\
& && \ldots \\
& && \text{fix}_\text{D}\, f
\end{aligned}$$

Finally, we prove that $fix_D\, f$ is the Least fixed point of all the fixed points of f. – that part of the proof I won't attempt to write out, although it should be in the book (I think?).

We can then use this definition on all sorts of recursive definitions, as long as (a) the domains are defined properly, and (b) the generating functions are recursive. In that case, we can use the $\text{fix}_D$ operator to find the Least Fixed Point, which always exists, and the recursive definition 'makes sense.'

## 4.2 Programmatic Definition

We'll write this bad boy out in Scheme. It's awesome. (Dave says, "Be prepared to be disturbed," but it's not as bad as all that.) The great thing about this is that it works on any function. And you should think about how to write this out in other programming languages.

```
(define fact-gen
   (lambda (f)
      (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))
```

Obviously, in this case, `((fact-gen fact) 5) = 120`. Is everyone alright with that, so far?

```
(define fixed-point
   (lambda (f)
      ((lambda (x) (lambda (z) ((f (x x)) z)))
       (lambda (x) (lambda (z) ((f (x x)) z))) )))
```

Hot! And of course, this works: `((fixed-point fact-gen) 5) = 120`. Go try it!

What's the intuition, about what's going on here? Well, to start, the fixed-point operator is your friend. It tests the expressive power of your language, for starters.

Ask yourself, what does 'z' represent? It'll be ranging over integers – it is the parameter of the factorial function (in this example). The (x x) part is going to be expanded, to give the same pair-of-lambdas that we start with. If you hit the base-case, then it uses it, otherwise it unwinds itself again.

It's the self-application that makes this possible. Self-application is necessary for recursion, (x x). And if you can do it, you can get recursion.

Go type it into Scheme.