Notes on recursion

Introduction

These notes give an overview of *fixed points*, a core programming language features used to explain recursive definition.

Approximating recursive functions

One way we can attempt to understand what a recursive definition means is to consider a *series of approximations* to the definition. For example, consider the standard definition of the factorial function, as expressed in our λ-calculus:

```
fact = λ n. if n = 0 then 1 else n * fact (n - 1)
```

We want to construct a series of approximations to the `fact` function. Each approximation will be able to do one more recursive call than the approximation before it. We can start with the version that makes no recursive calls:

```
fact₀ = λ n. if n = 0 then 1 else error
```

This approximates the `fact` function when it makes no recursive calls; I have used `error` to indicate cases that we have not yet approximated. We could replace `error` with any integer value; the point is simply to indicate that, whatever value we choose, it is not a correct approximation of the result of `fact`.

The next version of `fact` that we can approximate is the version that makes one (or fewer) recursive calls. This version will rely on the previous approxmation to handle the recursion:

```
fact₁ = λ n. if n = 0 then 1 else n * fact₀ (n - 1)
```

Observe two things about this code. First: `fact`$_1$ handles both the cases with 0 and 1 recursive calls. Second: none of our definitions so far use recursion. Other than how to implement `error` (really, anything will do), there are no mysteries about how any of these functions work.

We can continue in the same vein, building deeper and deeper approximations of the `fact` function:

```
fact_{n+1} = λ n. if n = 0 then 1 else n * fact_n (n - 1)
```

Now, we can observe that, for any program we can write, there is a value of *n* large enough that $\texttt{fact}_n$ is indistinguishable from the recursive definition of `fact`. For example, if the largest argument we ever pass to `fact` is 256, then $\texttt{fact}_{256}$ is a good enough approximation. Of course, so is $\texttt{fact}_{257}$, or $\texttt{fact}_{1024}$, and so forth.

However, this is a little unsatisfactory—we would like our formal understanding of `fact` to match our intuitive understanding, and our intuitive understanding works for all inputs, not just inputs up to some pre-defined threshold. To get a more satisfying understanding, we have to introduce a little bit of mathematics.

Fixed points

The key idea we need to steal from mathematics is that of a *fixed point*. The short version is that, for some function $f(x)$, c is a fixed point of f iff $f(c) = c$. Not every function has a fixed point; for example, the there is no integer fixed point for the function $f(x) = x + 1$. On the other hand, many functions do have fixed points; for example, 0 is the fixed point of the sine function.

How can this help us? Well, we can start by describing our series of appoximations to `fact` themselves as a function; that is, we want a function `factStep` such that $\texttt{factStep fact}_n$ gives $\texttt{fact}_{n+1}$. This is not too hard to do.

```
factStep = λ f. λ n. if n = 0 then 1 else n * f (n - 1)
```

You should be able to satisfy yourself that $\texttt{factStep fact}_0$ is $\texttt{fact}_1$, `factStep` $(\texttt{factStep fact}_0)$ is $\texttt{fact}_2$, and so forth. The question we then have to ask is: does `factStep` have a fixed point? That is, is there a function $\texttt{fact}_\omega$ such that `factStep` $\texttt{fact}_\omega$ is $\texttt{fact}_\omega$? Thanks to the *Kleene fixed-point theorems*, one of the more significant results in the mathematics of computation, we know that, not only does this function have a fixed point, but it is exactly the one obtained by iterating the function. That is to say, $\texttt{fact}_\omega$ is exactly the result of `factStep (factStep (factStep ... (factStep` $\texttt{fact}_0$`)))`.

Now, suppose that `fact` is defined by iterating `factStep`. How would `fact 3` behave? Well, because `fact` is the fixed point of `factStep`, we know that `face` is equal to `factStep fact`. So `fact 3` should be equal to `factStep fact 3`. We can substitute into the definition of `factStep`, getting `if isz 3 then 1 else 3 * fact (3 - 1)`, and so forth. Evaluating, we get to `3 * fact 2`, and we have gotten to the next recursive call, as we expect.

Generalizing fixed point construction

Hopefully, you found this brief journey into the mathematics of fixed points interesting, even if perhaps not entirely understandable. What is important, however, is that it has given us a mechanical way of explaining recursion. That is, rather than considering

recursive definitions, such as the one given for `fact` earlier, we can define recursion using fixed points of step functions, such as `factStep`. That is to say, we want to introduce an explicit fixed point construct `fix` to our λ-calculus. Then, we can factorial by

```
let factStep = λ f. λ n. if n = 0 then 1 else n * f (n - 1) in
let fact = fix factStep in
  (fact 3, fact 5)
```

And, the same construct can be used to define arbitrary other recursive functions. For example, the Fibonacci numbers

```
let fibStep = λ f. λ n.
  if isz n then 0
  else if isz (n - 1) then 1
  else f (n - 1) + f (n - 2) in
let fib = fix fibStep in
  (fib 3, fib 5)
```

In fact, you could even define the type checking and evaluation functions we write in class using fixed points (albeit, in a slightly more complicated λ-calculus that we have built so far). Evaluation might begin as follows:

```
let evalStep = λ eval. λ t.
  case t of
    Const x -> VInt x
  | Plus t1 t2 ->
      let VInt v1 = eval e1 in
      let VInt v2 = eval e2 in
        VInt (v1 + v2)
  ... in
let eval = fix evalStep in
  ...
```

`evalStep` is more complicated than `factStep` or `fibStep`, but the basic idea is the same: `evalStep` computes the next iteration of the evaluation function from the previous one. So, the fixed point of `evalStep` will be the recursive evaluation function. (You may recognize `evalStep` as being similar to the generic evaluation function we built in class recently. This should not be surprising; there, as here, our concern was the structure of recursion.)

Formalizing `fix`

To finish our discussion of fixed points, we want to introduce typing and evaluation rules for the `fix` term. We can make a first attempt based on translating the intuitive idea of a fixed point; that is, that if expression `t` is a stepping function (of type `T -> T`), then `fix t` should be the result of iterating that function (of type `T`):

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \texttt{fix}\, t : T} \qquad \frac{t\,(\texttt{fix}\, t) \Downarrow v}{\texttt{fix}\, t \Downarrow v}$$

If you check the definitions earlier in this file, you should see that they are accepted by this typing rule. The evaluation rule seems to capture the intuition of fixed points. And, if our λ-calculus were a *call-by-name* calculus, like Haskell, we would be done. (In fact, you can observe that this is all you need in Haskell; look at the definitions in Fix.hs).

However, our λ-calculus is *call-by-value* instead. And that means to compute the result of an application (like `t (fix t)`), we need to evaluate both the function (`t`) *and the argument* (`fix t`). However, this seems to leave us back where we started: to determine the value of `fix t`, we need to already have the value of `fix t`.

We can find our way out of this nest by making another observation. All of the values for which we have wanted fixed points are themselves functions. How does this help? It means that we can assume that the result of `fix t` will be applied to at least one more argument, and so we can delay the evaluation of the `fix t` until we see that argument. Concretely, instead of the equation

$$\texttt{fix}\, t = t\,(\texttt{fix}\, t)$$

we will have the equation

$$\texttt{fix}\, t\, x = t\,(\texttt{fix}\, t)\, x$$

or, equivalently,

$$\texttt{fix}\, t = \lambda x.\, t\,(\texttt{fix}\, e)\, x$$

Hopefully you can see how this fixes the problem: because functions are themselves values, the evalutation of `fix t` stops immediately. On the other hand, when applied to its next argument, it will evaluate as before.

We can update our typing and evaluation rules to reflect the restriction of fixed points to functions, as follows.

$$\frac{\Gamma \vdash t : (T_1 \to T_2) \to (T_1 \to T_2)}{\Gamma \vdash \texttt{fix}\, t : T_1 \to T_2} \qquad \frac{}{\texttt{fix}\, t \Downarrow \lambda x.\, t\,(\texttt{fix}\, t)\, x}$$

And now we have the final (for now) account of recursion in our λ-calculus. Because we have limited recursion to functions, it is not quite as expressive as it is in Haskell. For example, we cannot write the infinite streams of names used in homework 2. But, for most practical purposes, this is not a significant restriction.

Conclusion

The study of recursion is one of the more intricate parts of programming language theory. These notes have given you an introduction to one way of describing and reasoning about recursion. There is still plenty to come. For example, while we have a way to talk about recursively defined *values*, we have equally come to rely on recursively defined *types*. We will return to these topics later in the course.