Recitation 7: Partiality and Recursive Types

15-312: Foundations of Programming Languages

Charles Yuan

February 28, 2018

1 Partiality

So far, every system we have discussed in this course is *strongly normalizing*; that is, System **E**, System **T**, System **F**, and System **FE** may only express computations that provably terminate and evaluate to a value—they are **total**. It was impossible to express infinite loops or divergent programs in any of those systems since they do not support general recursion. Living in the garden of total languages has allowed us to disregard the possibility that a program does not terminate, so why should we leave?

Recall from your theory course that since the halting problem is undecidable, the set of Turing machine programs that terminate is uncomputable. Therefore *no total language can express every total program.* By contrast, a **partial** programming language, one that allows for the possibility of divergence, can possibly express all programs. We will introduce a partial programming language, System **PCF** (Plotkin, 1977), which is the first of this course to be Turing-complete. It does so by introducing fixed points, a means of **general recursion**.

PCF is a very simple system and writing programs in it is much easier than in System **T** or **F**. As we've mentioned before in the class, total programming languages also incur a blowup in the size of their programs, since they must embed the proof of the program's correctness. That issue will now disappear, and **PCF** programs look very much like ML programs.[1]

[1] These notes are partially derived from 15-312 Spring 2017 course notes by Jake Zimmerman.

2 System PCF

PCF is derived from **T** by adding the fixed-point recursion operator. It and the new zero test operator replace the System **T** primitive recursor.

Typ	τ	$::=$	$\texttt{nat}$	natural number
			$\tau_1 \to \tau_2$	function[2]
Exp	e	$::=$	x	variable
			$\lambda\,(x:\tau)\,e$	abstraction
			$e_1(e_2)$	application
			$\texttt{z}$	zero
			$\texttt{s}(e)$	successor
			$\texttt{ifz}\ e\ \{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \hookrightarrow e_1\}$	zero test
			$\texttt{fix}\ x:\tau\ \texttt{is}\ e$	recursion

The natural numbers, along with zero and successor, are exactly as we expect them to be. The zero test is analogous to a `case` on an expression, branching on whether it is zero or not. Unlike the System **T** recursor, it does not recursively compute on the predecessor.

Recursion is the job of the fixed-point operator `fix`, which defines a self-referential expression of some type. $\texttt{fix}\ x:t\ \texttt{is}\ e$ is roughly equivalent to the ML expression

```
val rec x : t = e
```

except that this fixed-point construct is much more powerful than recursive values in ML! In particular, e does not have to be a lambda expression.

2.1 Statics

There is only one interesting rule, that of the fixed-point operator. You should be able to infer how the typing of the zero test works; it's based on e_0 and e_1. Everything else is exactly as it was in System **T**.

$$\frac{\Gamma, x:\tau \vdash e:\tau}{\Gamma \vdash \texttt{fix}\ x:\tau\ \texttt{is}\ e:\tau}$$

Since x is a self-reference in e, we expect that they have the same type.

2.2 Dynamics

Again, the only interesting rule is that of the fixed-point operator. Most rules are as they were in lazy/eager System **T**, with the zero test either yielding e_0 or e_1 with e substituted for x depending on whether e is zero.

$$\frac{}{\texttt{fix}\ x:\tau\ \texttt{is}\ e \longmapsto [\texttt{fix}\ x:\tau\ \texttt{is}\ e/x]e}$$

That's it! To evaluate a fixed-point expression, we merely substitute an instance of the fixed-point for the self-reference in that expression. Intuitively, this satisfies exactly what recursion should be. Every time e refers to itself, we substitute in an instance of itself there.

[2]The "partial function" arrow $\rightharpoonup$ is often used for partial functions.

2.3 Examples in PCF

Certainly, we may write trivial programs in **PCF** using numbers and functions. They correspond to basic ML:

$$\begin{aligned}
\mathtt{z} &\cong \mathtt{0}\\
\mathtt{s}(\mathtt{s}(\mathtt{z})) &\cong \mathtt{2}\\
(\lambda\,(x:\mathtt{nat}\to\mathtt{nat})\,x)(\lambda\,(x:\mathtt{nat})\,x) &\cong \texttt{(fn (x : int -> int) => x)(fn (x : int) => x)}\\
\mathtt{ifz}\ \mathtt{s}(\mathtt{s}(\mathtt{z}))\ \{\mathtt{z}\hookrightarrow\mathtt{z}\mid\mathtt{s}(x)\hookrightarrow\mathtt{s}(x)\} &\cong \texttt{if 2 = 0 then 0 else (2 - 1) + 1}
\end{aligned}$$

What about a recursive ML function, like this?

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

We need to define multiplication in **PCF** first. In ML:

```
fun mult 0 _ = 0
  | mult m n = n + mult (m - 1) n
```

OK, we need addition too. In ML:

```
fun plus 0 n = n
  | plus m n = 1 + plus (m - 1) n
```

Now in **PCF**, we can translate directly.

$$\begin{aligned}
\mathtt{plus} \triangleq\ & \mathtt{fix}\ x:\mathtt{nat}\to\mathtt{nat}\to\mathtt{nat}\ \mathtt{is}\\
&\quad\lambda\,(m:\mathtt{nat})\ \lambda\,(n:\mathtt{nat})\\
&\qquad\mathtt{ifz}\ n\ \{\mathtt{z}\hookrightarrow m\mid\mathtt{s}(n')\hookrightarrow\mathtt{s}(x(n')(m))\}\\
\mathtt{mult} \triangleq\ & \mathtt{fix}\ x:\mathtt{nat}\to\mathtt{nat}\to\mathtt{nat}\ \mathtt{is}\\
&\quad\lambda\,(m:\mathtt{nat})\ \lambda\,(n:\mathtt{nat})\\
&\qquad\mathtt{ifz}\ m\ \{\mathtt{z}\hookrightarrow\mathtt{z}\mid\mathtt{s}(m')\hookrightarrow\mathtt{plus}(n)(x(m')(n))\}\\
\mathtt{fact} \triangleq\ & \mathtt{fix}\ x:\mathtt{nat}\to\mathtt{nat}\to\mathtt{nat}\ \mathtt{is}\\
&\quad\lambda\,(n:\mathtt{nat})\\
&\qquad\mathtt{ifz}\ n\ \{\mathtt{z}\hookrightarrow\mathtt{s}(\mathtt{z})\mid\mathtt{s}(n')\hookrightarrow\mathtt{mult}(n)(x(n'))\}
\end{aligned}$$

Note the structure of each expression: a fixed-point declaration with the type, then a body that refers back to x which is the fixed point. We could even have renamed x to `plus`, `mult`, `fact` respectively to make it even more like ML.

Now, what about our good friend, the Ackermann function? It was a mess to encode it in System **T**, while it's trivial in ML:

```
fun A 0 n = n + 1
  | A m 0 = A (m - 1) 1
  | A m n = A (m - 1) (A m (n - 1))
```

Guess what? It's trivial in **PCF** too!

$$\begin{aligned}
\mathtt{A} \triangleq\ & \mathtt{fix}\ x:\mathtt{nat}\to\mathtt{nat}\to\mathtt{nat}\ \mathtt{is}\\
&\quad\lambda\,(m:\mathtt{nat})\ \lambda\,(n:\mathtt{nat})\\
&\qquad\mathtt{ifz}\ m\ \{\mathtt{z}\hookrightarrow\mathtt{s}(n)\mid\mathtt{s}(m')\hookrightarrow\mathtt{ifz}\ n\ \{\mathtt{z}\hookrightarrow x(m')(\mathtt{s}(\mathtt{z}))\mid\mathtt{s}(n')\hookrightarrow x(m')(x(m)(n'))\}\}
\end{aligned}$$

Observe how easy **PCF** is to work with relative to **T**. Programs are shorter and more intuitive, and having full recursion is nice.

3 Partiality and Laziness

It turns out **PCF** can easily express things that even ML cannot, because the fixed point is not restricted to function types. We could easily declare a recursive natural number:

$$\omega \triangleq \texttt{fix}\ x : \texttt{nat}\ \texttt{is}\ \texttt{s}(x)$$

What is this thing? When we attempt to evaluate it, we get:

$$\begin{aligned} \omega &\longmapsto \texttt{fix}\ x : \texttt{nat}\ \texttt{is}\ \texttt{s}(x) \\ &\longmapsto \texttt{s}(\texttt{fix}\ x : \texttt{nat}\ \texttt{is}\ \texttt{s}(x)) \\ &\longmapsto \texttt{s}(\texttt{s}(\texttt{fix}\ x : \texttt{nat}\ \texttt{is}\ \texttt{s}(x))) \\ &\longmapsto \ldots \end{aligned}$$

This is an infinite stack of successors, and it never terminates. Could we call this an "infinite ordinal", perhaps? In an eager dynamics, we could never work with such a thing, but in a lazy dynamics where successor is lazy, it would actually be a value.

The interaction of partiality and laziness is now much more interesting; prior to this point, whether we used an eager or lazy dynamics had little bearing on program evaluation, in that all values in an eager setting are also values in the lazy setting, and all values in the lazy setting can be evaluated to values in the eager setting (obviously, since the language was total). But now, it is possible for a lazy semantics to yield a value, and an eager semantics to *diverge.*

How do we reason about this? Morally, we could consider the lazy semantics more versatile, as it allows a larger class of programs to terminate with some useful result while preserving every computation that could be done eagerly. But there is a critical vulnerability to the lazy semantics. In a lazy partial language, *non-termination is a value.* In the past, we could reason about values by induction, such as structural induction on ML datatypes. But induction requires that our structure be finite, which now *it may not be.* This is why, from a certain point of view, it is invalid to prove properties about structures in lazy partial languages by structural induction.[3] If an "inductive type" is required to be finite, as in our prior studies, then such a language cannot have true inductive types at all, as fixed points introduce arbitrary non-terminating expressions of any type.

Because of the issues discussed above, fixed points on non-function types are not as useful in eager settings as in lazy ones. In a lazy setting, fixed points may be used to easily define infinite datatypes, such as infinite numbers, streams, etc.

Of course, in a lazy setting, it's also trivial to define absurd expressions that always diverge, such as:

$$\bot \triangleq \texttt{fix}\ x : \tau\ \texttt{is}\ x$$

which has type τ, but diverges immediately in either eager or lazy dynamics. Fixed points irreversibly introduce the possibility of non-termination into the language.

3.1 Verification

In System **T**, the master proof that every program written in the language must terminate suffices to show the termination of any particular program. This is not the case with **PCF** programs, however. We can write all sorts of programs in **PCF** which may or may not terminate,

[3] Transfinite induction might come into play, but that is hardly a pleasant solution.

and it is up to the programmer to prove that their programs terminate. Suppose you wanted to implement the Euclidean algorithm for greatest common divisor in **T**:

$$\mathrm{gcd}(m,n) = \begin{cases} n & m = 0 \\ m & n = 0 \\ \mathrm{gcd}(m-n,n) & m > n \\ \mathrm{gcd}(n,n-m) & m \leq n \end{cases}$$

It would take a while; you'd define arithmetic and then every inductive step. [4]

Once you've implemented GCD in **T**, you've also completed a proof that the Euclidean algorithm terminates. But if you wrote it in **PCF** instead (exercise!), termination is no longer immediately apparent. You've saved time implementing the algorithm, but will now have to prove its termination. Of course, once you've done that, you have a **T** program...

3.2 Fixed Points

Why do we refer to `fix` as a fixed point, and why do we refer to it as such when its semantics are actually really simple? It's due to the mathematical interpretation of recursion. Consider the definition of factorial we previously discussed:

$$f(0) \triangleq 1$$
$$f(n) \triangleq n \times f(n-1)$$

At first glance, we could call this an inductive definition of f, and indeed it satisfies the properties of primitive recursion. But in general, the usage of f in its own definition is not always in a primitive recursive fashion (e.g. in Ackermann), so we should instead think about this as *a set of equations in the unknown f*.

Since f is unknown, we have to solve for it. One way is to think about what properties the solution must satisfy. If somebody gave us a function f' and claimed that it satisfies the definition of factorial, we can write this function F to see whether it really is a factorial:

$$F \triangleq \lambda\,(f : \mathtt{nat} \to \mathtt{nat})\,\lambda\,(n : \mathtt{nat})\,\mathtt{ifz}\ n\ \{\mathtt{z} \hookrightarrow \mathtt{s}(\mathtt{z}) \mid \mathtt{s}(n') \hookrightarrow \mathtt{mult}(n)(f(n'))\}$$

Given f', we can confidently claim that it really is the factorial function if $F(f') = f'$ in the sense of extensional equivalence. That should make sense, because F just checks each of the equations in the definition.

Now, what are the functions f that satisfy the equation $F(f) = f$? Any such function is known as a **fixed point** of F—an input to F that is the same as the output.

For any recursive specification of f, we can create such a tester function F. The `fix` operator in **PCF** computes a fixed point of F, which satisfies the specification of f. Proving that the dynamics of `fix` actually compute this value requires a bit more work, but conceptually this is the connection between recursion and fixed points in mathematics.

Keep the concept of fixed points in mind, as we'll revisit them later in the course when we talk about the untyped lambda calculus.

[4] An implementation is online at https://gist.github.com/joom/b5d6f69f32e1e8d9026c8a27f7d8de96.

4 System FPC

PCF gave us recursive expressions, which bring us much closer to ML. But what about the types themselves? We've seen inductive and coinductive types, but ML supports a large class of types that allow recursion in many ways:

```
datatype weird = Weird | Wonky of weird -> weird
```

Such a type is certainly impossible to express inductively or coinductively, meaning that we would like recursion at type level as well. Introducing **recursive types** gives us System **FPC**:

$$\begin{array}{llll} \mathsf{Typ} & \tau & ::= & t & \text{self-reference} \\ & & & \texttt{rec}(t.\tau) & \text{recursive type} \\ \\ \mathsf{Exp} & e & ::= & \texttt{fold}\{t.\tau\}(e) & \text{fold} \\ & & & \texttt{unfold}(e) & \text{unfold} \end{array}$$

Be warned—the `fold` and `unfold` operators are named the name way but are not exactly the same as `fold` and `unfold` from inductive and coinductive types. They are very conceptually similar, though.

4.1 Statics

In a sense, **FPC** combines the statics rules for `fold` and `unfold` from our previous formulation of inductive and coinductive types.

$$\frac{\Gamma \vdash e : [\texttt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \texttt{fold}\{t.\tau\}(e) : \texttt{rec}(t.\tau)} \qquad \frac{\Gamma \vdash e : \texttt{rec}(t.\tau)}{\Gamma \vdash \texttt{unfold}(e) : [\texttt{rec}(t.\tau)/t]\tau}$$

Now we have the ability to arbitrarily fold and unfold a recursive type, rather than being limited to just one or the other. But that isn't the most powerful aspect—now we are no longer limited to polynomial, or even positive type operators. We can use arbitrary well-formed type operators that include self-references, even in negative positions.

4.2 Dynamics

The eager dynamics are familiar from inductive and coinductive types.

$$\frac{e\ \mathsf{val}}{\texttt{fold}\{t.\tau\}(e)\ \mathsf{val}} \qquad \frac{e \longmapsto e'}{\texttt{fold}\{t.\tau\}(e) \longmapsto \texttt{fold}\{t.\tau\}(e')}$$

$$\frac{e \longmapsto e'}{\texttt{unfold}(e) \longmapsto \texttt{unfold}(e')} \qquad \frac{\texttt{fold}\{t.\tau\}(e)\ \mathsf{val}}{\texttt{unfold}(\texttt{fold}\{t.\tau\}(e)) \longmapsto e}$$

4.3 Induction and Coinduction

As you might expect, it is possible to encode all the familiar inductive types in eager **FPC**, using `fold` as the introduction form and `unfold` as elimination. For example, we may encode the natural numbers in **FPC** with products:

$$\begin{aligned} \texttt{z} &\triangleq \texttt{fold}\{t.[\texttt{z} \hookrightarrow \langle\rangle, \texttt{s} \hookrightarrow t]\}(\texttt{z} \cdot \langle\rangle) \\ \texttt{s}(e) &\triangleq \texttt{fold}\{t.[\texttt{z} \hookrightarrow \langle\rangle, \texttt{s} \hookrightarrow t]\}(\texttt{s} \cdot e) \end{aligned}$$

The introduction is relatively clear, but how do we perform recursion? We can use `unfold` to unpack an inductive type, perhaps implementing the `ifz` operator. But it turns out we don't need to bake `rec` into the language, as **FPC** includes much richer recursion.

How would we encode coinductive types? It's not quite clear under the eager dynamics, as `fold`s naturally will produce finite data structures which are inductive in nature. But what about a lazy interpretation, where a `fold` is a value even if its argument is not a value? In the example of natural numbers, we would be able to produce successors of non-values which we call values.

Thinking back to our earlier conundrum in **PCF**, we realize that the lazy dynamics encodes infinite natural numbers that resemble infinite coinductive types! So it turns out that eager numbers, lists, etc. transform to their coinductive counterparts in lazy **FPC**. Laziness and coinduction are connected!

4.4 Recursive Expressions

Despite **FPC** not having an explicit fixed-point operator like **PCF**, surprisingly it also has the capability of general recursion and is therefore Turing-complete. To see how this is possible, we will do the following:

1. Encode the notion of self-reference into the type system;
2. Show how self-reference can be translated into basic recursive types;
3. Encode the fixed-point operator in terms of self-references.

4.4.1 Self-Reference

In **PCF**, we explored the idea of computations being provided with self-references. The `fix` operator provides an explicit self-reference to a recursive expression, but did so purely at the expression level. What if we leveraged the new **FPC** type system to *reify* self-references at the type level? We would need a new type for self-references, and corresponding introduction and elimination forms:

Typ	τ	::=	...	
			$\texttt{self}(\tau)$	self-referential type
Exp	e	::=	...	
			$\texttt{self}\ x : \tau\ \texttt{is}\ e$	self-referential expression
			$\texttt{unroll}(e)$	unroll self-reference

The idea here is that any expression which wishes to reference itself must take on a self-referential type and be wrapped in a self-referential expression. No other expression shall perform self-reference, and it shall be possible to unroll a self-reference by substituting an instance of the overall expression for all self-referential locations.

The textbook gives a formal semantics for this new self-referential language, but we may work more at an intuitive level. A self-referential expression is an expression that is provided with a reference to itself. We could implement a function like factorial, and it might look something like this:

$$\texttt{fact} \triangleq \texttt{self}\ f : \texttt{nat} \to \texttt{nat}\ \texttt{is}\ \lambda\,(n : \texttt{nat})\,\texttt{ifz}\ n\ \{\texttt{z} \hookrightarrow \texttt{s}(\texttt{z}) \mid \texttt{s}(n') \hookrightarrow \texttt{mult}(n)(\texttt{unroll}(f)(n'))\}$$

As you can see, we define it as a self-referential expression of type $\texttt{nat} \to \texttt{nat}$, with an explicit self-reference f that we unroll later. This is quite like fixed points in **PCF**, with a bit more syntactic baggage and also the innovation of having a self-reference type, $\texttt{self}(\texttt{nat} \to \texttt{nat})$.

This works fine, but we can actually interpret a self-reference as an explicit argument. If we want to produce a self-referential expression of type $\texttt{self}(\tau)$, we must first give it *itself*, also of type $\texttt{self}(\tau)$.

This bears repeating: a self-referential expression that computes type τ can be thought of as first taking an argument of *its own type*, then yielding τ. In short, we have the following isomorphism:

$$\texttt{self}(\tau) \cong \texttt{self}(\tau) \to \tau$$

4.4.2 Translation into FPC

But wait! That type equation can be rewritten as a type operator that we can try to solve:

$$t.t \to \tau$$

This is a new variety of type operator, one where t is in the negative position (left of the arrow). In the past, we couldn't deal with it in terms of inductive and coinductive types. We can now, with recursive types. The solution is:

$$\texttt{self}(\tau) \triangleq \texttt{rec}(t.t \to \tau)$$

Given this, we may write the introduction and elimination forms of the self-referential type. They are:

$$\begin{aligned} \texttt{self}\ x : \tau\ \texttt{is}\ e &\triangleq \texttt{fold}\{t.t \to \tau\}(\lambda\,(x : \texttt{rec}(t.t \to \tau))\,e) \\ \texttt{unroll}(e) &\triangleq \texttt{unfold}(e)(e) \end{aligned}$$

That double application of e is not a typo—try yourself to derive the proper type correspondence.

By the way, the formulation of $\texttt{self}(\tau)$ as a type isomorphism is highly accurate—the two forms we just derived are the forward and backwards mappings of a type isomorphism!

4.4.3 Fixed Points

Now that we have shown that self-references can be encoded in vanilla **FPC**, we can use them as macros to define a fixed-point operator. This is the definition we will use:

$$\texttt{fix}\ x : \tau\ \texttt{is}\ e \triangleq \texttt{unroll}(\texttt{self}\ y : \tau\ \texttt{is}\ [\texttt{unroll}(y)/x]e)$$

This definition may not be completely obvious, but it can be evaluated to see that it has the same semantics as the System **PCF** fixed-point construct.

Through our hard work, we have shown that System **FPC** is capable of encoding the same fixed points as **PCF**, and is therefore equally powerful, thanks to its recursive types!

5 Origin of Partiality

At this point in the course, we have introduced several methods of expanding our languages' computational power. By the Church-Turing thesis, the limit to a language's expressive power is the power of a Turing machine, or equivalently Church's lambda calculus, which we shall investigate as part of dynamic programming languages. We have given an explanation of why partiality is necessary to achieve this computational power, and introduced two systems, **PCF** and **FPC**, which are both Turing-complete.

As the course progresses, we will see even more interesting ways of maximizing computational power. While System **T** and **F** were provably total, it is remarkably easy to go beyond them with language features, and expand a language to become Turing-complete. When we do so, we inevitably enable ourselves to make new interesting computations, at the expense of losing the guarantees of a total language. It's a brave new world of undecidability.

With great power comes great responsibility, so from here on out, constructs like these will shift the burden of proof from the language to the programmer. They are all equally powerful additions to a programming language, giving us the expressive power to describe any computation:

- General recursion (abstractly, what you see in any typical language like C or ML)
- Fixed points (**PCF**)
- Recursive types (**FPC**)
- Mutable state (not seen yet, but it will be very exciting!)