Lecture Notes 7
Recursion

Carlo Angiuli

B522: PL Foundations
March 2, 2026

The STLC is a good starting point for understanding the syntax and semantics of typed functional programming languages, but it lacks many features of realistic languages, perhaps most notably *recursion*. In this lecture we will consider several core language features that enable recursive definitions: first at the term level, and then at the type level.

We will start by considering PCF (Programming Computable Functions), a famous core calculus first conceived of (but not published) in 1969 by Scott [Sco93], and further developed and popularized by Plotkin [Plo77]. Despite PCF's apparent simplicity, the development of a satisfactory denotational semantics for PCF—the so-called *full abstraction problem*—was one of the biggest open problems in programming language theory for over two decades until its resolution by Abramsky, Jagadeesan, and Malacaria [AJM00] and Hyland and Ong [HO00]. After that we will consider isorecursive types, an extension to PCF which brings us surprisingly close to typed functional programming languages such as OCaml and Haskell.

PCF and isorecursive types are respectively covered in Chapters 19 and 20 of Harper [Har16]; we will also touch on topics discussed in Chapters 9 (System T), 15 (inductive and coinductive types), and 47 (equational reasoning for PCF).

1 Recursion via fixed points

Realistic programming languages all come with mechanisms for *defining*—giving names to—constants and functions. (In the presence of λ, the latter can often be treated as a special case of the former.) Although definition mechanisms can certainly be interesting in their own right [Gra+25], it is typical to think of definitions as purely a convenience feature allowing us to abbreviate large terms.

For example, when we wrote:

$$\texttt{to} : (A \times (B \times C)) \to ((A \times B) \times C)$$
$$\texttt{to} \coloneqq \lambda x : A \times (B \times C).((\texttt{fst}(x), \texttt{fst}(\texttt{snd}(x))), \texttt{snd}(\texttt{snd}(x)))$$

we understood that `to` is just a shorthand for the large λ expression to the right of $\texttt{to} \coloneqq \ldots$, to be treated as completely interchangeable with `to`.

However, in real programming languages, function definitions are also a source of *self-reference*: that is, the body of a named function is allowed to mention the very same function that is currently being defined.

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

We can still think of this as a "defining equation" for `fact`,

`fact` = (λ (n) (if (zero? n) 1 (* n (fact (sub1 n)))))

but although though the left-hand side *can* be replaced by the right-hand side, the meaning of the equation is much less clear: `fact` appears on both the left- and the right-hand sides, so it is much more than just an *abbreviation*! Turning it into an abbreviation requires *solving for* `fact`, turning it into an equation of the form

`fact` = *[some expression not mentioning* `fact`*]*

But this is easier said than done. For one thing, we have not yet seen any mechanism for defining interesting functions such as `fact` on natural numbers. Besides, equations between terms may in general have multiple solutions:

$$\texttt{f} = \texttt{f}$$

or even no solutions:

$$\infty = \texttt{suc}(\infty)$$

We can recast the problem of solving equations—or indeed systems of equations, in the case of mutually-recursive functions—in terms of finding *fixed points of functionals*.[12] (Recall that a *fixed point* of a function $f : X \to X$ is some $x \in X$ for which $f(x) = x$.) In the case of `fact`, we define the (non-recursive!) functional $F : (\texttt{nat} \to \texttt{nat}) \to (\texttt{nat} \to \texttt{nat})$ as follows:

$F(f) \coloneqq$ (λ (n) (if (zero? n) 1 (* n (f (sub1 n)))))

[12] "Functionals" are what mathematicians call higher-order functions.

It is easy to see that any fixed point of F (any f for which $F(f) = f$) will be a solution to the defining equation of `fact`.

The language PCF, which we will define momentarily, provides a mechanism for computing fixed points of functionals (actually, of arbitrary functions $\tau \to \tau$) by successive approximation, which we will use in Section 3 to define recursive functions such as `fact`. Before introducing the syntax of PCF, let us quickly sketch how the definition of `fact` can be seen as the limit of a sequence of approximations.

Functions in PCF are actually *partial functions*, which are like functions that may not be defined on all inputs. We can imagine that the zeroth-order approximation to `fact` is a function that is nowhere defined:

$$\texttt{fact}_0 \coloneqq (\lambda\ (\texttt{n})\ (\texttt{error "undefined"}))$$

This function bears little resemblance to the actual factorial function, but it has type `nat` $\to$ `nat` so it is a valid input to the functional F.

$$\texttt{fact}_1 \coloneqq F(\texttt{fact}_0) \coloneqq (\lambda\ (\texttt{n})\ (\texttt{if}\ (\texttt{zero?}\ \texttt{n})\ 1\ (\texttt{*}\ \texttt{n}\ (\texttt{fact}_0\ (\texttt{sub1}\ \texttt{n})))))$$

We see that $\texttt{fact}_1$ returns the correct answer on `0` but will throw an `error` on $n \geq 1$. Going one more round,

$$\texttt{fact}_2 \coloneqq F(\texttt{fact}_1) \coloneqq (\lambda\ (\texttt{n})\ (\texttt{if}\ (\texttt{zero?}\ \texttt{n})\ 1\ (\texttt{*}\ \texttt{n}\ (\texttt{fact}_1\ (\texttt{sub1}\ \texttt{n})))))$$

is even closer to `fact`, working for `0` and (because $\texttt{fact}_1$ works on `0`) also `1`.

There are two important things to note. First, for any finite n, $\texttt{fact}_n$ is emphatically *not* a fixed point of F and thus *not* a solution to the defining equation of `fact`; rather, $\texttt{fact}_n = F(\texttt{fact}_{n-1})$. But it is a kind of *approximate* solution, in the sense that $\texttt{fact}_n$ behaves just like `fact` up to recursion depth n.

In fact, one can make precise the idea that `fact`—or more precisely, the smallest solution to the defining equation of `fact`—is the *limit* of these approximations:

$$\lim_{n\to\infty} \texttt{fact}_n$$

or the result of infinitely iterating the functional F on $\texttt{fact}_0$: $F(F(\cdots(F(\texttt{fact}_0))))$.

2 PCF

In what is now a familiar pattern, we will quickly run through the syntax, type system, and operational semantics of PCF, and prove type safety.

2.1 Syntax

Different authors present PCF's syntax differently, and as with the STLC, one can and typically does extend PCF with various type formers. The canonical starting point, which we present below, is to include natural numbers and functions in addition to PCF's signature feature, a fixed point operator for arbitrary types.

Types	$\tau ::=$	$\mathtt{nat}$	$\mathtt{nat}$	natural numbers
		$\mathtt{arr}(\tau_1, \tau_2)$	$\tau_1 \to \tau_2$	(partial) functions
Terms	$e ::=$	$\mathtt{zero}$	$\mathtt{zero}$	zero
		$\mathtt{suc}(e)$	$\mathtt{suc}(e)$	successor
		$\mathtt{case}_{\mathtt{nat}}(e, e_z, x.e_s)$	$\mathtt{case}_{\mathtt{nat}}\ e\ [\mathtt{zero} \to e_z]\ [\mathtt{suc}(x) \to e_s]$	case analysis for nat
		$\mathtt{lambda}(\tau, x.e)$	$\lambda x : \tau.e$	λ-abstraction
		$\mathtt{app}(e_1, e_2)$	$e_1\ e_2$	application
		$\mathtt{fix}(\tau, x.e)$	$\mathtt{fix}\ x : \tau\ \mathtt{is}\ e$	fixed point operator

The syntax, typing rules, and operational semantics of function types remain unchanged from the STLC, but we reproduce them in these notes for completeness. Product and sum types can also be included with no changes.

Remark 7.1. The $\mathtt{case}_{\mathtt{nat}}$ term is slightly nonstandard; a more common (but equivalent) version is to include a predecessor function and an "$\mathtt{if}\ \mathtt{zero}$" term $\mathtt{ifz}(e, e_z, e_s)$ that steps to e_z if e evaluates to $\mathtt{zero}$ and e_s otherwise. Note that Harper [Har16] uses $\mathtt{case}_{\mathtt{nat}}$ but calls it $\mathtt{ifz}$.

2.2 Type system

There are only two interesting typing rules, those for $\mathtt{case}_{\mathtt{nat}}$ and $\mathtt{fix}$.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{var}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}}\ \mathsf{nat}\text{-}\textsc{intro}_1 \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{suc}(e) : \mathsf{nat}}\ \mathsf{nat}\text{-}\textsc{intro}_2$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma, x : \mathsf{nat} \vdash e_s : \tau}{\Gamma \vdash \mathsf{case}_{\mathsf{nat}}\ e\ [\mathsf{zero} \to e_z]\ [\mathsf{suc}(x) \to e_s] : \tau}\ \mathsf{nat}\text{-}\textsc{case}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e_2 : \tau_1 \to \tau_2} \to\text{-}\textsc{intro} \qquad \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash f\ e_1 : \tau_2} \to\text{-}\textsc{elim}$$

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

The nat-case rule is very similar to the +-elim rule, and for good reason: terms of type nat should intuitively be either zero or suc(e) for e : nat, so given a term of type nat, we can "case" (or "match") on whether it is the former or the latter. In the latter case, there is data inside the constructor to be passed to the suc branch.

What's unusual about nat and $\mathsf{case}_{\mathsf{nat}}$ in comparison to $\tau_1 + \tau_2$ and case is that case "simplifies" the situation of having a term of type $\tau_1 + \tau_2$ into the situations of having a term of type τ_1 or a term of type τ_2, whereas the suc branch of $\mathsf{case}_{\mathsf{nat}}$ "simplifies" having a term of type nat to having a term of type nat. (Of course, this is to be expected, because suc creates a nat out of a nat.)

Remark 7.2. In fact, there is a precise sense in which we can think of nat as being the sum type unit + nat. More on this later.

Exercise 7.3. Define the predecessor function in terms of $\mathsf{case}_{\mathsf{nat}}$.

The fix operator is unusual in that its typing rule does not mention any particular type constructor in either the premise or conclusion. Recalling our discussion of fixed points, we can think of fix as a function $(\tau \to \tau) \to \tau$ which returns a fixed point of its input function. (Or rather its input *functional*, given that one often instantiates τ with a function type.)

Another less abstract way of understanding fix is to think of it as a mechanism for defining *anonymous recursive functions*. Whereas λs are a natural way to decouple the *definition* of functions from the *naming* of functions—hence why λs are also known as anonymous functions—it is less clear how to decouple the definition of *recursive* functions from their naming. (How exactly do we call the

function within its own body?) To solve this problem, ($\texttt{fix}\ x : \tau\ \texttt{is}\ e$) binds a variable x which refers to the entire `fix` term itself within the expression e. This will hopefully become clearer once we consider some examples in Section 3.

The typing judgment of PCF satisfies the usual lemmas.

Lemma 7.4 (Uniqueness of types). *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau'$ *then* $\tau = \tau'$.

Lemma 7.5 (Weakening). *If* $\Gamma \vdash e : \tau$ *and* τ' ty *then* $\Gamma, x : \tau' \vdash e : \tau$.

Lemma 7.6 (Substitution). *If* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash e[e'/x] : \tau$.

2.3 Operational semantics

By this point you should be able to guess the operational semantics of $\mathsf{case}_{\mathsf{nat}}$, so the only particularly interesting rule here is the one for `fix`.

$$\frac{}{\mathsf{zero}\ \mathsf{val}} \qquad \frac{v\ \mathsf{val}}{\mathsf{suc}(v)\ \mathsf{val}} \qquad \frac{}{\lambda x : \tau.e\ \mathsf{val}}$$

$$\frac{e \longmapsto e'}{\mathsf{suc}(e) \longmapsto \mathsf{suc}(e')} \qquad \frac{f \longmapsto f'}{f\ e_1 \longmapsto f'\ e_1} \qquad \frac{e_1 \longmapsto e_1'}{(\lambda x : \tau_1.e_2)\ e_1 \longmapsto (\lambda x : \tau_1.e_2)\ e_1'}$$

$$\frac{v_1\ \mathsf{val}}{(\lambda x : \tau_1.e_2)\ v_1 \longmapsto e_2[v_1/x]} \qquad \frac{e \longmapsto e'}{\begin{array}{c}\mathsf{case}_{\mathsf{nat}}\ e\ [\mathsf{zero} \to e_z]\ [\mathsf{suc}(x) \to e_s] \longmapsto \\ \mathsf{case}_{\mathsf{nat}}\ e'\ [\mathsf{zero} \to e_z]\ [\mathsf{suc}(x) \to e_s]\end{array}}$$

$$\frac{}{\mathsf{case}_{\mathsf{nat}}\ \mathsf{zero}\ [\mathsf{zero} \to e_z]\ [\mathsf{suc}(x) \to e_s] \longmapsto e_z}$$

$$\frac{v\ \mathsf{val}}{\mathsf{case}_{\mathsf{nat}}\ \mathsf{suc}(v)\ [\mathsf{zero} \to e_z]\ [\mathsf{suc}(x) \to e_s] \longmapsto e_s[v/x]}$$

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

The operational semantics of PCF satisfy the usual lemmas.

Lemma 7.7 (Finality of values). *If* v val *then there is no* e' *such that* $v \longmapsto e'$.

Lemma 7.8 (Determinacy). *If* $e \longmapsto e'$ *and* $e \longmapsto e''$ *then* $e' = e''$.

2.4 Type safety

This isn't the interesting part of PCF either. We can prove type safety via canonical forms (Lemma 7.10), progress (Lemma 7.11), and preservation (Lemma 7.13).

Theorem 7.9 (Type safety)**.**

1. *If* $\cdot \vdash e : \tau$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' : \tau$.
2. *If* $\cdot \vdash e : \tau$ *then either* e val *or* $e \longmapsto e'$.

Lemma 7.10 (Canonical forms)**.** *Suppose* $\cdot \vdash v : \tau$ *and* v val. *Then:*

1. *If* $\tau = \mathtt{nat}$, *then* $v = \mathtt{zero}$ *or* $v = \mathsf{suc}(v')$ *where* v' val *and* $\cdot \vdash v' : \mathtt{nat}$.
2. *If* $\tau = \tau_1 \to \tau_2$, *then* $v = \lambda x : \tau_1 . e_2$ *where* $x : \tau_1 \vdash e_2 : \tau_2$.

Lemma 7.11 (Progress)**.** *If* $\cdot \vdash e : \tau$ *then either* e val *or* $e \longmapsto e'$ *for some* e'.

Proof. By rule induction on $\cdot \vdash e : \tau$. We consider only the `fix` rule.

- Case $\dfrac{x : \tau \vdash e : \tau}{\cdot \vdash \mathtt{fix}\ x : \tau\ \mathtt{is}\ e : \tau}\ \mathtt{fix}$:

  This always takes a step to $e[(\mathtt{fix}\ x : \tau\ \mathtt{is}\ e)/x]$. □

Remark 7.12. Note that if $e = x$, then `fix` steps to itself:

$$\mathtt{fix}\ x : \tau\ \mathtt{is}\ x \longmapsto x[(\mathtt{fix}\ x : \tau\ \mathtt{is}\ x)/x] = \mathtt{fix}\ x : \tau\ \mathtt{is}\ x$$

A term that steps to itself could perhaps be considered "stuck" in some sense, but it doesn't break type safety. For the purposes of type safety, recall that a "stuck" term is one whose behavior is unspecified by the operational semantics: a term which is neither a value nor takes a step.

Lemma 7.13 (Preservation)**.** *If* $\cdot \vdash e : \tau$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof. By rule induction on $e \longmapsto e'$. We consider only the `fix` rule.

- Case $\dfrac{}{\mathtt{fix}\ x : \tau\ \mathtt{is}\ e \longmapsto e[(\mathtt{fix}\ x : \tau\ \mathtt{is}\ e)/x]}$:

  We must show $\cdot \vdash e[(\mathtt{fix}\ x : \tau\ \mathtt{is}\ e)/x] : \tau$. By inversion on the typing judgment $\cdot \vdash \mathtt{fix}\ x : \tau\ \mathtt{is}\ e : \tau$, we have $x : \tau \vdash e : \tau$. The result follows by Lemma 7.6. □

What we do **not** have, for the first time, is termination. Indeed, by Remark 7.12, at every type τ there is at least one well-typed closed term, $\mathtt{fix}\ x : \tau\ \mathtt{is}\ x$, which steps to itself forever.

3 Programming in PCF

Let us return to the `fact` example from earlier. Rewriting the defining equation of `fact` and the associated functional F using $\mathsf{case}_{\mathsf{nat}}$, we have:

$$\mathtt{fact} = \lambda n : \mathtt{nat}.\mathsf{case}_{\mathsf{nat}}\ n\ [\mathtt{zero} \to \mathtt{suc}(\mathtt{zero})]\ [\mathtt{suc}(x) \to \mathtt{mult}\ n\ (\mathtt{fact}\ x)]$$

and $f : \mathtt{nat} \to \mathtt{nat} \vdash F : \mathtt{nat} \to \mathtt{nat}$ where

$$F \coloneqq \lambda n : \mathtt{nat}.\mathsf{case}_{\mathsf{nat}}\ n\ [\mathtt{zero} \to \mathtt{suc}(\mathtt{zero})]\ [\mathtt{suc}(x) \to \mathtt{mult}\ n\ (f\ x)]$$

where we assume that $\mathtt{mult} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$ is provided.

Then writing `fact` for the genuine *abbreviation*

$$\mathtt{fact} \coloneqq \mathtt{fix}\ f : \mathtt{nat} \to \mathtt{nat}\ \mathtt{is}\ F$$

we can compute

$$\begin{aligned}\mathtt{fact} &\longmapsto F[\mathtt{fact}/f] \\ &= \lambda n : \mathtt{nat}.\mathsf{case}_{\mathsf{nat}}\ n\ [\mathtt{zero} \to \mathtt{suc}(\mathtt{zero})]\ [\mathtt{suc}(x) \to \mathtt{mult}\ n\ (\mathtt{fact}\ x)]\end{aligned}$$

We can think of `fix` and thus `fact` as "expanding on demand," unfolding one additional round of F every time that evaluation encounters the term `fact`. (This is the sense in which `fact` is an infinite number of applications of $F(F(\ldots(F.)$ It is tedious but possible to check that (assuming `mult` is correct) the `fact` function will compute the factorial of any natural number.

Remark 7.14. In Section 1 we described the base case $\mathtt{fact}_0$ as the `error` function, but because there is no `error` in PCF, it is perhaps better to think of the base case as a function that diverges on all inputs.

Remark 7.15. In light of Remark 7.14, we can understand the infinitely-looping term $\mathtt{fix}\ x : \tau\ \mathtt{is}\ x$ as the limit of successive approximations of the identity function $\mathbf{I} : \tau \to \tau$. If we define $\mathtt{loop}_0 : \tau$ as a term that infinitely loops, we can see that the limit $\lim_{n\to\infty} \mathbf{I}^n(\mathtt{loop}_0)$ also infinitely loops. (Note that because `fix` is not restricted to function types, we can use it to compute fixed points of non-higher-order functions such as **I**!)

Remark 7.16. The term $(\mathtt{fix}\ x : \tau\ \mathtt{is}\ e)$ is roughly like `(letrec ([`x e`]) `x`)` in Racket, except that `letrec` only allows e to mention x in "guarded" positions in the sense that e must evaluate to a value without attempting to evaluate x. For example, `(letrec ([x (lambda (y) x)]) x)` is a valid Racket program that evaluates to a procedure, whereas `(letrec ([x (+ x x)]) x)` produces the error `x: undefined; cannot use before initialization`. In PCF, `fix` has no such restriction, as we have already seen in the case of $\mathtt{fix}\ x : \tau\ \mathtt{is}\ x$.

3.1 Structural recursion for natural numbers

The factorial function is a very well-behaved recursive function on the natural numbers. Not only does it terminate on all inputs $\cdot \vdash n : \mathtt{nat}$, but it makes no recursive calls in the `zero` case and exactly one recursive call in the $\mathtt{suc}(m)$ case, to `fact` m. Thus computing `fact` n requires exactly $n - 1$ fixed point unfoldings.

Contrast this with the (naïve) Fibonacci function, which terminates but has a more complex call graph:

```
(define (fib n)
  (if (or (zero? n) (= n 1))
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

or even the (iterated) Collatz function, whose recursive calls are even stranger and whose termination is an open problem.

```
(define (collatz n)
  (cond
    [(= n 1) 1]
    [(even? n) (collatz (/ n 2))]
    [(odd? n) (collatz (add1 (* 3 n)))]))
```

Functions on the natural numbers such as `fact`—whose recursion pattern directly mirrors the structure of the natural numbers—

```
(define (f n)
  (if (zero? n)
      ...
      (... (f (sub1 n)) ...)))
```

are called *structurally recursive*. Structural recursion is much less powerful than arbitrary or *general recursion*, and although general-purpose programming languages typically do not distinguish between these forms of recursion, it is quite useful to do so. In particular, structurally-recursive functions terminate on all inputs, requiring exactly $n - 1$ unfoldings to compute their value on n.

There is a famous language, Gödel's System T [Har16, Chapter 9], which replaces PCF's `fix` with a structural recursion principle for `nat` called `natrec`. This principle is similar to $\mathtt{case}_{\mathtt{nat}}$ but provides the $\mathtt{suc}(n)$ case with both n *and*

the result of the recursive call on n.

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma, x : \mathsf{nat}, y : \tau \vdash e_s : \tau}{\Gamma \vdash \mathsf{natrec}(e, e_z, x.y.e_s) : \tau}\ \textsc{nat-elim}$$

$$\frac{e \longmapsto e'}{\mathsf{natrec}(e, e_z, x.y.e_s) \longmapsto \mathsf{natrec}(e', e_z, x.y.e_s)}$$

$$\frac{}{\mathsf{natrec}(\mathsf{zero}, e_z, x.y.e_s) \longmapsto e_z}$$

$$\frac{v\ \mathsf{val}}{\mathsf{natrec}(\mathsf{suc}(v), e_z, x.y.e_s) \longmapsto e_s[v/x][\mathsf{natrec}(v, e_z, x.y.e_s)/y]}$$

Exercise 7.17. Show that `natrec` is definable in PCF in terms of `fix` and $\mathsf{case}_{\mathsf{nat}}$.

Remark 7.18. Perhaps the starkest difference between structural recursion and general recursion is that the former always terminates whereas the latter may not. However, even *terminating* general recursive definitions are harder to analyze: as the Collatz example shows, they may make arbitrarily many recursive calls, whereas we have an exact bound on the number of recursive calls required to evaluate a structurally-recursive function.

That said, PCF functions satisfy a *compactness* property [Har16, Theorem 47.15] which roughly states that *if* $\cdot \vdash e : \mathsf{nat}$ terminates, then every `fix` term inside of e can be replaced by some finite approximation. That is, every `fix` in a terminating computation must only be unfolded finitely many times.

3.2 Fixed points and laziness

Call-by-name PCF is quite different from call-by-value PCF. In particular, call-by-name PCF has infinite values like $\infty = \mathsf{suc}(\infty)$.

The operational semantics presented above are arguably a combination of call-by-name (for `fix` itself) and call-by-value (for everything else), but it is actually a bit tricky to present a fully call-by-value PCF in which `fix` is not restricted to function types.

4 Isorecursive types

Let us revisit Remark 7.2, in which we said that the nat-CASE rule

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma, x : \mathsf{nat} \vdash e_s : \tau}{\Gamma \vdash \mathsf{case}_{\mathsf{nat}}\ e\ [\mathsf{zero} \to e_z]\ [\mathsf{suc}(x) \to e_s] : \tau}\ \textsc{nat-case}$$

looks like a special case of case in which nat is the sum type unit + nat (where the left disjunct corresponds to zero and the right disjunct corresponds to suc). Indeed, if we add products and sums to PCF, we can define an isomorphism nat ≅ unit + nat:

$$\mathsf{to} : \mathsf{nat} \to (\mathsf{unit} + \mathsf{nat})$$
$$\mathsf{to} \coloneqq \lambda n : \mathsf{nat}.\mathsf{case}_{\mathsf{nat}}\ n\ [\mathsf{zero} \to \mathsf{inl}(())]\ [\mathsf{suc}(x) \to \mathsf{inr}(x)]$$
$$\mathsf{from} : (\mathsf{unit} + \mathsf{nat}) \to \mathsf{nat}$$
$$\mathsf{from} \coloneqq \lambda z : (\mathsf{unit} + \mathsf{nat}).\mathsf{case}\ z\ [\mathsf{inl}(x) \to \mathsf{zero}]\ [\mathsf{inr}(y) \to \mathsf{suc}(y)]$$

We can actually think of nat ≅ unit + nat as the "defining isomorphism" of nat, in much the same way that

```
fact = (λ (n) (if (zero? n) 1 (* n (fact (sub1 n)))))
```

is the defining equation of fact. We say that nat is a *recursive type*—a type defined in terms of itself—in much the same way that fact is a *recursive term*. The reason that the suc branch of $\mathsf{case}_{\mathsf{nat}}$ binds a variable of type nat is that the suc constructor builds a natural number out of another natural number, in much the same way that fact computes factorials in terms of fact.

Following our analysis of recursive terms in PCF, we want to "solve for nat," turning it into an isomorphism

$$\mathsf{nat} \cong \textit{[some type expression not mentioning } \mathsf{nat}\textit{]}$$

and we can recast this problem as finding a fixed point of a *type-level function* $F(\tau) \coloneqq \mathsf{unit} + \tau$.

Remark 7.19. There are of course a few differences between nat and fact, but the situation is more similar than it may seem. First, obviously, one is a type and one is a term. Secondly, nat is *isomorphic*, not equal, to a type expression involving itself. (But there is a variation of this story in which nat is actually equal to unit + nat.) Thirdly, unlike higher-order functions, we do not have a preexisting notion of "type-level function." (Nor will we introduce one, although it does make sense.)

Fixed points (up to isomorphism) of type equations are known as *isorecursive types*. We will write $\mu\alpha.(\mathtt{unit} + \alpha)$ for the least solution to the isomorphism $\tau \cong \mathtt{unit} + \tau$. (Roughly speaking, imagine this as the type $\mathtt{fix}\ \alpha$: "tp" $\mathtt{is}\ \mathtt{unit} + \alpha$.) Being a solution to that isomorphism means that it satisfies:

$$\mu\alpha.\mathtt{unit} + \alpha \cong \mathtt{unit} + (\mu\alpha.\mathtt{unit} + \alpha)$$

We write `unfold` for the function going from left to right because it "unfolds" the equation, and $\mathtt{fold}_{\mu\alpha.\mathtt{unit}+\alpha}$ for the function going from right to left.

Remark 7.20. There is another version of recursive types, called *equirecursive types*, which satisfy an *equation* $\mathtt{nat} = \mathtt{unit} + \mathtt{nat}$ [Pie02, Chapter 20]. These have some benefits but are somewhat more complicated both theoretically and practically.

4.1 Syntax

We extend the syntax of PCF with product types, sum types, one new type former, and two new term formers.

Types	$\tau ::=$	$\vdots$	$\vdots$	$\vdots$
		$\mathtt{rec}(\alpha.\tau)$	$\mu\alpha.\tau$	isorecursive types
Terms	$e ::=$	$\vdots$	$\vdots$	$\vdots$
		$\mathtt{fold}(\alpha.\tau, e)$	$\mathtt{fold}_{\mu\alpha.\tau}(e)$	fold
		$\mathtt{unfold}(e)$	$\mathtt{unfold}(e)$	unfold

The syntax chart above introduces a new feature to our languages: binding structure at the type level. The type constructor $\mu\alpha.\tau$ binds a *type variable* α that may occur in the type expression τ. Up until now, the collection of types has been specified by an inductively-defined judgment τ ty, but now that we have type variables we must replace τ ty with the hypothetical judgment

$$\Delta \vdash \tau\ \mathsf{ty}$$

where $\Delta = \alpha\ \mathsf{ty}, \beta\ \mathsf{ty}, \ldots$ is a context of type variables.

The correspondence between ABT notation and well-formedness rules precisely mirrors that of ordinary terms with binding, but just to be explicit, we write a few of the rules for $\Delta \vdash \tau$ ty below:

$$\frac{}{\Delta, \alpha\ \mathsf{ty} \vdash \alpha\ \mathsf{ty}} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{ty} \qquad \Delta \vdash \tau_2\ \mathsf{ty}}{\Delta \vdash \tau_1 \to \tau_2\ \mathsf{ty}} \qquad \frac{\Delta, \alpha\ \mathsf{ty} \vdash \tau\ \mathsf{ty}}{\Delta \vdash \mu\alpha.\tau\ \mathsf{ty}} \qquad \ldots$$

As with other hypothetical judgments, we can define a notion of capture-avoiding substitution of a type for a type variable, and the well-formed type judgment satisfies weakening and substitution lemmas.

Lemma 7.21 (Weakening for types)**.** *If* $\Delta \vdash \tau\ \mathsf{ty}$ *then* $\Delta, \alpha\ \mathsf{ty} \vdash \tau\ \mathsf{ty}$.

Lemma 7.22 (Substitution for types)**.** *If* $\Delta, \alpha\ \mathsf{ty} \vdash \tau\ \mathsf{ty}$ *and* $\Delta \vdash \tau'\ \mathsf{ty}$ *then* $\Delta \vdash \tau[\tau'/\alpha]\ \mathsf{ty}$.

In this language, terms never contain free type variables. Type annotations in terms such as λ must be closed types, and although the type annotation in `fold` does bind a type variable α, that is the only type variable allowed:

$$\frac{\alpha\ \mathsf{ty} \vdash \tau\ \mathsf{ty} \qquad \Gamma \vdash e\ \mathsf{tm}}{\Gamma \vdash \mathtt{fold}_{\mu\alpha.\tau}(e)\ \mathsf{tm}}$$

4.2 Type system

The new typing rules are fairly straightforward: `fold` and `unfold` enact the two directions of the isomorphism $\mu\alpha.\tau \cong \tau[(\mu\alpha.\tau)/\alpha]$.

$$\frac{\Gamma \vdash e : \tau[(\mu\alpha.\tau)/\alpha]}{\Gamma \vdash \mathtt{fold}_{\mu\alpha.\tau}(e) : \mu\alpha.\tau}\ \mu\text{-INTRO} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathtt{unfold}(e) : \tau[(\mu\alpha.\tau)/\alpha]}\ \mu\text{-ELIM}$$

Note that the judgment $\Gamma \vdash e : \tau$ ranges only over *closed* types $\cdot \vdash \tau\ \mathsf{ty}$; we can check by Lemma 7.22 that the above rules preserve this invariant.

Example 7.23. Abbreviating $\mathtt{nat} \coloneqq \mu\alpha.\mathtt{unit} + \alpha$, we have

$$\lambda n : \mathtt{nat}.\mathtt{unfold}(n) : \mathtt{nat} \to (\mathtt{unit} + \mathtt{nat})$$
$$\lambda x : (\mathtt{unit} + \mathtt{nat}).\mathtt{fold}_{\mu\alpha.\mathtt{unit}+\alpha}(x) : (\mathtt{unit} + \mathtt{nat}) \to \mathtt{nat}$$

4.3 Operational semantics

The operational semantics are perhaps even more straightforward.

$$\frac{v\ \mathsf{val}}{\mathtt{fold}_{\mu\alpha.\tau}(v)\ \mathsf{val}} \qquad \frac{e \longmapsto e'}{\mathtt{fold}_{\mu\alpha.\tau}(e) \longmapsto \mathtt{fold}_{\mu\alpha.\tau}(e')}$$

$$\frac{e \longmapsto e'}{\mathtt{unfold}(e) \longmapsto \mathtt{unfold}(e')} \qquad \frac{v\ \mathsf{val}}{\mathtt{unfold}(\mathtt{fold}_{\mu\alpha.\tau}(v)) \longmapsto v}$$

4.4 Type safety

Yes.

5 Programming with isorecursive types

Now that we have a general mechanism for solving type equations, we can recover PCF's $\mathtt{nat}$ as a special case. We define:

$$\begin{aligned}
\mathtt{nat} &\coloneqq \mu\alpha.\mathtt{unit} + \alpha \\
\mathtt{zero} &\coloneqq \mathtt{fold}_{\mu\alpha.\mathtt{unit}+\alpha}(\mathtt{inl}_{\mathtt{unit}+\mathtt{nat}}(())) \\
\mathtt{suc}(e) &\coloneqq \mathtt{fold}_{\mu\alpha.\mathtt{unit}+\alpha}(\mathtt{inr}_{\mathtt{unit}+\mathtt{nat}}(e)) \\
\mathtt{case}_{\mathtt{nat}}(e, e_z, x.e_s) &\coloneqq \mathtt{case}\ \mathtt{unfold}(e)\ [\mathtt{inl}(y) \to e_z]\ [\mathtt{inr}(x) \to e_s]
\end{aligned}$$

vindicating our earlier observation that $\mathtt{case}_{\mathtt{nat}}$ looks like ordinary $\mathtt{case}$.

But $\mathtt{nat}$ is far from the only interesting example of isorecursive types; in fact, we can now define a large class of types known as *algebraic data types*, named recursive sums of products that are a cornerstone of typed functional programming languages such as OCaml and Haskell. Examples of algebraic data types include lists, trees, and even binding-free expression languages such as the boolean language we encountered in the first week of the semester.

Example 7.24. For any type τ we can define the type of *lists of* τ, $\mathtt{list}(\tau)$, which satisfies the isomorphism $\mathtt{list}(\tau) \cong \mathtt{unit} + (\tau \times \mathtt{list}(\tau))$.

$$\begin{aligned}
\mathtt{list}(\tau) &\coloneqq \mu\alpha.\mathtt{unit} + (\tau \times \alpha) \\
\mathtt{nil} &\coloneqq \mathtt{fold}_{\mu\alpha.\mathtt{unit}+(\tau\times\alpha)}(\mathtt{inl}_{\mathtt{unit}+(\tau\times\mathtt{list}(\tau))}(())) \\
\mathtt{cons}(e, \ell) &\coloneqq \mathtt{fold}_{\mu\alpha.\mathtt{unit}+(\tau\times\alpha)}(\mathtt{inr}_{\mathtt{unit}+(\tau\times\mathtt{list}(\tau))}((e, \ell))) \\
\mathtt{case}_{\mathtt{list}(\tau)}(e, e_n, x.y.e_c) &\coloneqq \\
&\quad \mathtt{case}\ \mathtt{unfold}(e)\ [\mathtt{inl}(z) \to e_n]\ [\mathtt{inr}(w) \to e_c[\mathtt{fst}(w)/x][\mathtt{snd}(w)/y]]
\end{aligned}$$

Example 7.25. For any type τ, the type of *binary trees labeled with* τ, $\mathtt{tree}(\tau)$, satisfies the isomorphism $\mathtt{tree}(\tau) \cong \mathtt{unit} + \tau + (\mathtt{tree}(\tau) \times \mathtt{tree}(\tau))$.

Example 7.26. Our boolean expression language from the second lecture, $\mathtt{exp}$, satisfies the isomorphism $\mathtt{exp} \cong \mathtt{unit} + \mathtt{unit} + \mathtt{exp} + (\mathtt{exp} \times \mathtt{exp} \times \mathtt{exp})$.

in fact, fix can be defined using recursive types; see [Har16, Section 20.3].

References

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. "Full Abstraction for PCF". In: *Information and Computation* 163.2 (2000), pp. 409–470. ISSN: 0890-5401. DOI: 10.1006/inco.2000.2930.

[Gra+25] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. "Controlling unfolding in type theory". In: *Mathematical Structures in Computer Science* 35 (2025). DOI: 10.1017/S0960129525100327.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[HO00] J.M.E. Hyland and C.-H.L. Ong. "On Full Abstraction for PCF: I, II, and III". In: *Information and Computation* 163.2 (2000), pp. 285–408. ISSN: 0890-5401. DOI: 10.1006/inco.2000.2917.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 0-262-16209-1.

[Plo77] G.D. Plotkin. "LCF considered as a programming language". In: *Theoretical Computer Science* 5.3 (1977), pp. 223–255. ISSN: 0304-3975. DOI: 10.1016/0304-3975(77)90044-5.

[Sco93] Dana S. Scott. "A type-theoretical alternative to ISWIM, CUCH, OWHY". In: *Theoretical Computer Science* 121.1 (1993), pp. 411–440. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90095-B.