CS 430/530
Formal Semantics

Zhong Shao

Yale University
Department of Computer Science

Recursive Functions and Types

April 1, 2025

PCF: T + General Recursion (A Language for Partial Recursive Functions)

The source of partiality in **PCF** is the concept of *general recursion*, which permits the solution of equations between expressions. The price for admitting solutions to all such equations is that computations may not terminate—the solution to some equations might be undefined (divergent). In **PCF**, the programmer must make sure that a computation terminates; the type system does not guarantee it. The advantage is that the termination proof need not be embedded into the code itself, resulting in shorter programs.

For example, consider the equations

$$f(0) \triangleq 1$$
$$f(n+1) \triangleq (n+1) \times f(n).$$

Intuitively, these equations define the factorial function. They form a system of simultaneous equations in the unknown f, which ranges over functions on the natural numbers. The function we seek is a *solution* to these equations—a specific function $f : \mathbb{N} \to \mathbb{N}$ such that the above conditions are satisfied.

System PCF

A solution to such a system of equations is a fixed point of an associated functional (higher-order function). To see this, let us re-write these equations in another form:

$$f(n) \triangleq \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

Re-writing yet again, we seek f given by

$$n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

Now define the *functional* F by the equation $F(f) = f'$, where f' is given by

$$n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

System PCF

Why should an operator such as F have a fixed point? The key is that functions in **PCF** are *partial*, which means that they may diverge on some (or even all) inputs. Consequently, a fixed point of a functional F is the limit of a series of approximations of the desired solution obtained by iterating F. Let us say that a partial function ϕ on the natural numbers, is an *approximation* to a total function f if $\phi(m) = n$ implies that $f(m) = n$. Let $\bot\colon \mathbb{N} \rightharpoonup \mathbb{N}$ be the totally undefined partial function—$\bot(n)$ is undefined for every $n \in \mathbb{N}$. This is the "worst" approximation to the desired solution f of the recursion equations given above. Given any approximation ϕ of f, we may "improve" it to $\phi' = F(\phi)$. The partial function ϕ' is defined on 0 and on $m+1$ for every $m \geq 0$ on which ϕ is defined. Continuing, $\phi'' = F(\phi') = F(F(\phi))$ is an improvement on ϕ', and hence a further improvement on ϕ. If we start with $\bot$ as the initial approximation to f, then pass to the limit

$$\lim_{i \geq 0} F^{(i)}(\bot),$$

we will obtain the least approximation to f that is defined for every $m \in \mathbb{N}$, and hence is the function f itself. Turning this around, if the limit exists, it is the solution we seek.

System PCF Syntax

The syntax of **PCF** is given by the following grammar:

Typ	τ	$::=$	$\mathtt{nat}$	$\mathtt{nat}$	naturals
			$\mathtt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	partial function
Exp	e	$::=$	x	x	variable
			$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(e)$	$\mathtt{s}(e)$	successor
			$\mathtt{ifz}\{e_0; x.e_1\}(e)$	$\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$	zero test
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda\,(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\mathtt{fix}\{\tau\}(x.e)$	$\mathtt{fix}\, x:\tau\, \mathtt{is}\, e$	recursion

System PCF Statics

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{19.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{19.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{19.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathtt{ifz}\{e_0; x.e_1\}(e) : \tau} \tag{19.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}\{\tau_1\}(x.e) : \mathtt{parr}(\tau_1; \tau_2)} \tag{19.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{parr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau} \tag{19.1f}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}\{\tau\}(x.e) : \tau} \tag{19.1g}$$

Lemma 19.1. *If* $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

System PCF Dynamics

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{19.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{s}(e)\ \mathsf{val}} \tag{19.2b}$$

$$\frac{}{\mathtt{lam}\{\tau\}(x.e)\ \mathsf{val}} \tag{19.2c}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{s}(e) \longmapsto \mathtt{s}(e')}\right] \tag{19.3a}$$

$$\frac{e \longmapsto e'}{\mathtt{ifz}\{e_0; x.e_1\}(e) \longmapsto \mathtt{ifz}\{e_0; x.e_1\}(e')} \tag{19.3b}$$

$$\frac{}{\mathtt{ifz}\{e_0; x.e_1\}(\mathtt{z}) \longmapsto e_0} \tag{19.3c}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{ifz}\{e_0; x.e_1\}(\mathtt{s}(e)) \longmapsto [e/x]e_1} \tag{19.3d}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{19.3e}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{19.3f}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\mathtt{lam}\{\tau\}(x.e); e_2) \longmapsto [e_2/x]e} \tag{19.3g}$$

$$\frac{}{\mathtt{fix}\{\tau\}(x.e) \longmapsto [\mathtt{fix}\{\tau\}(x.e)/x]e} \tag{19.3h}$$

System PCF Safety

Theorem 19.2 (Safety)**.**

1. *If* $e : \tau$ *and* $e \longmapsto e'$, *then* $e' : \tau$.
2. *If* $e : \tau$, *then either* e val *or there exists* e' *such that* $e \longmapsto e'$.

Proof The proof of preservation is by induction on the derivation of the transition judgment. Consider rule (19.3h). Suppose that $\mathtt{fix}\{\tau\}(x.e) : \tau$. By inversion and substitution we have $[\mathtt{fix}\{\tau\}(x.e)/x]e : \tau$, as required. The proof of progress proceeds by induction on the derivation of the typing judgment. For example, for rule (19.1g) the result follows because we may make progress by unwinding the recursion. □

System PCF Definitional Equality

Definitional equality for the call-by-name variant of **PCF**, written $\Gamma \vdash e_1 \equiv e_2 : \tau$, is the strongest congruence containing the following axioms:

$$\frac{}{\Gamma \vdash \mathtt{ifz}\{e_0; x.e_1\}(\mathtt{z}) \equiv e_0 : \tau} \tag{19.4a}$$

$$\frac{}{\Gamma \vdash \mathtt{ifz}\{e_0; x.e_1\}(\mathtt{s}(e)) \equiv [e/x]e_1 : \tau} \tag{19.4b}$$

$$\frac{}{\Gamma \vdash \mathtt{fix}\{\tau\}(x.e) \equiv [\mathtt{fix}\{\tau\}(x.e)/x]e : \tau} \tag{19.4c}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}\{\tau_1\}(x.e_2); e_1) \equiv [e_1/x]e_2 : \tau} \tag{19.4d}$$

These rules suffice to calculate the value of any closed expression of type `nat`: if $e : \mathtt{nat}$, then $e \equiv \overline{n} : \mathtt{nat}$ iff $e \longmapsto^* \overline{n}$.

System PCF Definability

Let us write $\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e$ for a recursive function within whose body, $e : \tau_2$, are bound two variables, $y : \tau_1$ standing for the argument and $x : \tau_1 \rightharpoonup \tau_2$ standing for the function itself. The dynamic semantics of this construct is given by the axiom

$$\frac{}{(\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e)(e_1) \longmapsto [\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e,\, e_1/x,\, y]e}\ .$$

That is, to apply a recursive function, we substitute the recursive function itself for x and the argument for y in its body.

Recursive functions are defined in **PCF** using recursive functions, writing

$$\mathtt{fix}\, x : \tau_1 \rightharpoonup \tau_2\, \mathtt{is}\, \lambda\, (y : \tau_1)\, e$$

for $\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e$. We may easily check that the static and dynamic semantics of recursive functions are derivable from this definition.

System PCF Definability

The primitive recursion construct of **T** is defined in **PCF** using recursive functions by taking the expression

$$\texttt{rec}\, e\, \{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x)\, \texttt{with}\, y \hookrightarrow e_1\}$$

to stand for the application $e'(e)$, where e' is the general recursive function

$$\texttt{fun}\, f(u{:}\texttt{nat}){:}\tau\, \texttt{is}\, \texttt{ifz}\, u\, \{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \hookrightarrow [f(x)/y]e_1\}.$$

In general, functions definable in **PCF** are partial in that they may be undefined for some arguments. A partial (mathematical) function, $\phi : \mathbb{N} \rightharpoonup \mathbb{N}$, is *definable* in **PCF** iff there is an expression $e_\phi : \texttt{nat} \rightharpoonup \texttt{nat}$ such that $\phi(m) = n$ iff $e_\phi(\overline{m}) \equiv \overline{n} : \texttt{nat}$. So, for example, if ϕ is the totally undefined function, then e_ϕ is any function that loops without returning when it is applied.

System PCF Definability

It is informative to classify those partial functions ϕ that are definable in **PCF**. The *partial recursive functions* are defined to be the primitive recursive functions extended with the *minimization* operation: given $\phi(m, n)$, define $\psi(n)$ to be the least $m \geq 0$ such that (1) for $m' < m$, $\phi(m', n)$ is defined and non-zero, and (2) $\phi(m, n) = 0$. If no such m exists, then $\psi(n)$ is undefined.

Theorem 19.3. *A partial function ϕ on the natural numbers is definable in* **PCF** *iff it is partial recursive.*

Proof sketch Minimization is definable in **PCF**, so it is at least as powerful as the set of partial recursive functions. Conversely, we may, with some tedium, define an evaluator for expressions of **PCF** as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Therefore, **PCF** does not exceed the power of the set of partial recursive functions. □

System PCF Definability

Church's Law states that the partial recursive functions coincide with the set of effectively computable functions on the natural numbers—those that can be carried out by a program written in any programming language that is or will ever be defined.[1] Therefore, **PCF** is as powerful as any other programming language with respect to the set of definable functions on the natural numbers.

The universal function ϕ_{univ} for **PCF** is the partial function on the natural numbers defined by

$$\phi_{univ}(\ulcorner e \urcorner)(m) = n \text{ iff } e(\overline{m}) \equiv \overline{n} : \texttt{nat}.$$

In contrast to **T**, the universal function ϕ_{univ} for **PCF** is partial (might be undefined for some inputs). It is, in essence, an interpreter that, given the code $\overline{\ulcorner e \urcorner}$ of a closed expression of type $\texttt{nat} \rightharpoonup \texttt{nat}$, simulates the dynamic semantics to calculate the result, if any, of applying it to the $\overline{m}$, obtaining $\overline{n}$. Because this process may fail to terminate, the universal function is not defined for all inputs.

System PCF Definability

By Church's Law, the universal function is definable in **PCF**. In contrast, we proved in Chapter 9 that the analogous function is *not* definable in **T** using the technique of diagonalization. It is instructive to examine why that argument does not apply in the present setting. As in Section 9.4, we may derive the equivalence

$$e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \equiv \mathtt{s}(e_\Delta(\overline{\ulcorner e_\Delta \urcorner}))$$

for **PCF**. But now, instead of concluding that the universal function, e_{univ}, does not exist as we did for **T**, we instead conclude for **PCF** that e_{univ} diverges on the code for e_Δ applied to its own code.

System FPC of Recursive Types

The language **FPC** has products, sums, and partial functions inherited from the preceding development, extended with the new concept of recursive types. The syntax of recursive types is defined as follows:

Typ	τ	$::=$	t	t	self-reference
			$\mathtt{rec}(t.\tau)$	$\mathtt{rec}\, t\, \mathtt{is}\, \tau$	recursive type
Exp	e	$::=$	$\mathtt{fold}\{t.\tau\}(e)$	$\mathtt{fold}(e)$	fold
			$\mathtt{unfold}(e)$	$\mathtt{unfold}(e)$	unfold

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}(t.\tau)\ \mathsf{type}} \tag{20.1}$$

The statics of folding and unfolding is given by the following rules:

$$\frac{\Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}\{t.\tau\}(e) : \mathtt{rec}(t.\tau)} \tag{20.2a}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{20.2b}$$

System FPC of Recursive Types

The dynamics of folding and unfolding is given by these rules:

$$\frac{[e\ \mathsf{val}]}{\mathtt{fold}\{t.\tau\}(e)\ \mathsf{val}} \tag{20.3a}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{fold}\{t.\tau\}(e) \longmapsto \mathtt{fold}\{t.\tau\}(e')}\right] \tag{20.3b}$$

$$\frac{e \longmapsto e'}{\mathtt{unfold}(e) \longmapsto \mathtt{unfold}(e')} \tag{20.3c}$$

$$\frac{\mathtt{fold}\{t.\tau\}(e)\ \mathsf{val}}{\mathtt{unfold}(\mathtt{fold}\{t.\tau\}(e)) \longmapsto e} \tag{20.3d}$$

Theorem 20.1 (Safety)**.** *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val, or there exists e' such that $e \longmapsto e'$.

Eager FPC for Inductive Types

Recursive types may be used to represent inductive types such as the natural numbers. Using an *eager* dynamics for **FPC**, the recursive type

$$\rho = \mathtt{rec}\, t\ \mathtt{is}\, [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow t]$$

satisfies the type equation

$$\rho \cong [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow \rho],$$

and is isomorphic to the type of eager natural numbers. The introduction and elimination forms are defined on ρ by the following equations:[1]

$$\mathtt{z} \triangleq \mathtt{fold}(\mathtt{z} \cdot \langle\rangle)$$

$$\mathtt{s}(e) \triangleq \mathtt{fold}(\mathtt{s} \cdot e)$$

$$\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\} \triangleq \mathtt{case}\, \mathtt{unfold}(e)\, \{\mathtt{z} \cdot \_ \hookrightarrow e_0 \mid \mathtt{s} \cdot x \hookrightarrow e_1\}.$$

Lazy FPC for Coinductive Types

On the other hand, under a lazy dynamics for **FPC**, the same recursive type ρ',

$$\mathtt{rec}\, t\ \mathtt{is}\, [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow t],$$

satisfies the same type equation,

$$\rho' \cong [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow \rho'],$$

but is not the type of natural numbers! Rather, it is the type `lnat` of lazy natural numbers introduced in Section 19.4. As discussed there, the type ρ' contains the "infinite number" ω, which is of course not a natural number.

Eager FPC for Inductive Types

Similarly, using an eager dynamics for **FPC**, the type `natlist` of lists of natural numbers is defined by the recursive type

$$\texttt{rec}\, t\, \texttt{is}\, [\texttt{n} \hookrightarrow \texttt{unit}, \texttt{c} \hookrightarrow \texttt{nat} \times t],$$

which satisfies the type equation

$$\texttt{natlist} \cong [\texttt{n} \hookrightarrow \texttt{unit}, \texttt{c} \hookrightarrow \texttt{nat} \times \texttt{natlist}].$$

The list introduction operations are given by the following equations:

$$\texttt{nil} \triangleq \texttt{fold}(\texttt{n} \cdot \langle\rangle)$$

$$\texttt{cons}(e_1; e_2) \triangleq \texttt{fold}(\texttt{c} \cdot \langle e_1, e_2 \rangle).$$

A conditional list elimination form is given by the following equation:

$$\texttt{case}\, e\, \{\texttt{nil} \hookrightarrow e_0 \mid \texttt{cons}(x; y) \hookrightarrow e_1\} \triangleq \texttt{case}\, \texttt{unfold}(e)\, \{\texttt{n} \cdot \_ \hookrightarrow e_0 \mid \texttt{c} \cdot \langle x, y \rangle \hookrightarrow e_1\},$$

where we have used pattern-matching syntax to bind the components of a pair for the sake of clarity.

Lazy FPC for Coinductive Types

Now consider the *same* recursive type, but in the context of a lazy dynamics for **FPC**. What type is it? If all constructs are lazy, then a value of the recursive type

$$\texttt{rec}\,t\,\texttt{is}\,[\texttt{n} \hookrightarrow \texttt{unit}, \texttt{c} \hookrightarrow \texttt{nat} \times t],$$

has the form $\texttt{fold}(e)$, where e is an unevaluated computation of the sum type, whose values are injections of unevaluated computations of either the unit type or of the product type $\texttt{nat} \times t$. And the latter consists of pairs of an unevaluated computation of a (lazy!) natural number, and an unevaluated computation of another value of this type. In particular, this type contains infinite lists whose tails go on without end, as well as finite lists that eventually reach an end. The type is, in fact, a version of the type of infinite streams defined in Chapter 15, rather than a type of finite lists as is the case under an eager dynamics.

Self-Reference & Self-Referential Type

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

The statics of these constructs is given by the following rules:

$$\frac{\Gamma, x : \texttt{self}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{self}\{\tau\}(x.e) : \texttt{self}(\tau)} \tag{20.4a}$$

$$\frac{\Gamma \vdash e : \texttt{self}(\tau)}{\Gamma \vdash \texttt{unroll}(e) : \tau} \tag{20.4b}$$

Self-Reference & Self-Referential Type

The dynamics is given by the following rule for unrolling the self-reference:

$$\frac{}{\texttt{self}\{\tau\}(x.e)\ \mathsf{val}} \tag{20.5a}$$

$$\frac{e \longmapsto e'}{\texttt{unroll}(e) \longmapsto \texttt{unroll}(e')} \tag{20.5b}$$

$$\frac{}{\texttt{unroll}(\texttt{self}\{\tau\}(x.e)) \longmapsto [\texttt{self}\{\tau\}(x.e)/x]e} \tag{20.5c}$$

Self-Reference & Self-Referential Type

The type $\mathtt{self}(\tau)$ is definable from recursive types. As suggested earlier, the key is to consider a self-referential expression of type τ to depend on the expression itself. That is, we seek to define the type $\mathtt{self}(\tau)$ so that it satisfies the isomorphism

$$\mathtt{self}(\tau) \cong \mathtt{self}(\tau) \rightharpoonup \tau.$$

We seek a fixed point of the type operator $t.t \rightharpoonup \tau$, where $t \notin \tau$ is a type variable standing for the type in question. The required fixed point is just the recursive type

$$\mathtt{rec}(t.t \rightharpoonup \tau),$$

which we take as the definition of $\mathtt{self}(\tau)$.

The self-referential expression $\mathtt{self}\{\tau\}(x.e)$ is the expression

$$\mathtt{fold}(\lambda\,(x : \mathtt{self}(\tau))\,e).$$

We may check that rule (20.4a) is derivable according to this definition. The expression $\mathtt{unroll}(e)$ is correspondingly the expression

$$\mathtt{unfold}(e)(e).$$

Self-Reference & Self-Referential Type

It is easy to check that rule (20.4b) is derivable from this definition. Moreover, we may check that

$$\texttt{unroll}(\texttt{self}\{\tau\}(y.e)) \longmapsto^* [\texttt{self}\{\tau\}(y.e)/y]e.$$

Self-Reference & Self-Referential Type

The self-referential type $\texttt{self}(\tau)$ can be used to define general recursion for *any* type. We may define $\texttt{fix}\{\tau\}(x.e)$ to stand for the expression

$$\texttt{unroll}(\texttt{self}\{\tau\}(y.[\texttt{unroll}(y)/x]e))$$

where the recursion at each occurrence of x is unrolled within e. It is easy to check that this verifies the statics of general recursion given in Chapter 19. Moreover, it also validates the dynamics, as shown by the following derivation:

$$\begin{aligned}\texttt{fix}\{\tau\}(x.e) &= \texttt{unroll}(\texttt{self}\{\tau\}(y.[\texttt{unroll}(y)/x]e)) \\ &\longmapsto^* [\texttt{unroll}(\texttt{self}\{\tau\}(y.[\texttt{unroll}(y)/x]e))/x]e \\ &= [\texttt{fix}\{\tau\}(x.e)/x]e.\end{aligned}$$

It follows that recursive types can be used to define a non-terminating expression of every type, $\texttt{fix}\{\tau\}(x.x)$.