With the introduction of PCF we have introduced the possibility of divergence, or undefinedness, in a programming language. In contrast to languages such as T or F in which every well-typed expression terminates, in a language such as PCF there are expressions that do not terminate. Consequently, functions are inherently partial in that they may be undefined for some inputs, rather than total as they are in terminating languages. The trade-off is that total languages require the termination proof to be baked into the code itself, which can lead to extremely awkward programming, whereas in partial languages the termination proof is external to the program itself, allowing for very slick code that must be proved to be properly defined.

In a total language there is no sense in distinguishing expressions from values in the sense that every expression has a value, so why bother insisting that certain expressions be values as such. Variables range over the values of a type, but all expressions have a value, so we can freely substitute any expression for a variable without fear of compromising the meaning of a program. And so we do. As an aside, let us mention, though, that this cavalier attitude is appropriate only insofar as correctness is concerned, but is not at all justified when considering efficiency. For example, if I substitute a value for a variable that occurs multiple times, there is no overhead because the value always evaluates to itself wherever it occurs. But if I substitute a computation that stands for a value for a variable, then that computation is replicated as many times as the variable occurs, and hence must be repeated every time it is used. This is clearly a bad idea, which can be remedied using a technique called memoization that we shall consider later in the course. For now, let us ignore efficiency and focus only on correctness.

When divergence is possible, substituting a computation for a variable is not equivalent to substituting a computation for it, precisely because the computation may not have a value. Considering this issue carefully leads to two classes of languages, the lazy, aka non-strict, aka by-name, languages, and the eager, aka strict, aka by-value, languages. The principal difference between them lies in the range of significance of variables of a type:

1. In a by-name language variables range over the computations of a type, regardless of whether they terminate.
2. In a by-value language variables range over the values of a type, which are of course fully evaluated.

It is also possible to describe the distinction in another way that is sometimes helpful to guide intuition. According to this view variables always range over the values of a type, but by-name languages stretch the meaning of a value to include divergence (!), whereas by-value languages retain the more familiar meaning of a value as a fully-evaluated expression.

Thus, in a by-name language, all expressions are values, whereas in a by-value language only certain expressions are values. Put another way, in a by-name language there is only one mode of expression, all are computations (or all are values, according to taste), whereas in a by-value language there are two modes of expression, values and computations.

The mode distinction can be expressed as follows. In a by-name language there is exactly one mode of typing
judgment, $\Gamma \vdash e : \tau$, stating that $e$ is a computation of type $\tau$. The judgment is defined so as to ensure that any expression may be substituted for any variable, regardless of whether it diverges:

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

The statics of the by-name variant of PCF is given in Figure 1. It is easy to check that the just-stated substitution property is valid for this language.

In a by-value language there are two modes of typing, $\Gamma \vdash e : \tau$ for values, and $\Gamma \vdash e \sim \tau$ for computations. These judgments are defined to ensure that any value may be substituted for any variable of suitable type, preserving typing:

1. If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau'$.
2. If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' \sim \tau'$, then $\Gamma \vdash [e/x]e' \sim \tau'$.

The statics of the by-value variant of PCF is given in Figure 3. It is again easy to check that the required substitution principles are valid.

The dynamics of the two variants is chosen to cohere with the intended meaning of values and computations. In the by-name variant (Figure 2) arguments are passed to functions unevaluated, which is permissible because variables range over general computations, whereas in the by-value variant arguments are evaluated before they are passed because variables stand for (closed) values at run-time. Similarly, the successor function is evaluated lazily in the by-name variant, because there is no reason to insist on evaluating the predecessor until we are sure it is actually needed—and it can be passed in unevaluated form to the successor branch of the conditional. Alternatively, in the by-value variant (Figure 4) the successor is evaluated eagerly, because the predecessor might be extracted by a conditional, and it must be a value for this to make sense.

The by-name variant admits $\omega \equiv \text{fix } x : \text{nat is s}(x)$ as a value of type nat. It may be thought of as an infinite stack of successors, which is therefore larger than any finite stack of successors ending with zero—it is an infinite “natural number”! Of course this stretches the meaning of “natural number” to the breaking point—it invalidates the principle of mathematical induction. For example, zero is a finite number, and the successor of a finite number is again finite, but $\omega$ is not finite. Under a by-name interpretation the type nat should be renamed to conat, the type of co-natural numbers, the largest type whose elements are either zero or the successor of a co-natural number. Thus, $\omega$ is a conatural number, it being its own successor. Having a type of co-natural numbers is well and good, but notice that in the by-name variant, it is not possible to define the actual type of natural numbers, precisely because non-termination is regarded as a value of every type.

The by-value formulation of PCF defines the open values of each type to include variables, exactly because they range over values of their type. The computations are defined as those expressions that may require further evaluation. Every value is a computation, but, in contrast to the by-name case, fixed point computations are not permitted, because variables are constrained to range over values. Instead certain values are generalized to be self-referential. For example, the $\lambda$-abstraction is generalized to a function expression that may refer to itself, with the self-reference resolved on application.

The type nat in the by-value variant of PCF is indeed the type of natural numbers; its closed values are precisely the numerals $z, s(z), \ldots$. It is also possible to define a type conat of co-natural numbers, which contains co-zero, written $s$, and a self-referential co-successor, written $s(x.z.e)$, and an analogue of the zero-test. Their statics and dynamics are given in Figure 5. Notice that $\omega \equiv s(z.x.x)$ is the infinite co-natural number, and is a value. The conditional unrolls the recursion to determine the predecessor, which is required to be value.

The mode distinction between values and computations in by-value PCF may be bridged by introducing the

---

1 The notation is chosen to suggest that co-natural numbers are the natural numbers “backwards” in a certain sense.
The modality $\tau\text{ susp}$ whose values are unevaled, or suspended, computations of type $\tau$. These are activated by the computation $\text{bind} \ e_1 \leftarrow e\text{ in } xe\ e_2$, which forces the evaluation of the suspension $e_1$ and passes its value to $e_2$. The $\text{bind}$ construct provides a less ad hoc account of sequencing than the $\text{let}$ introduced earlier, because it arises as the elimination form of the suspended computation type. The statics and dynamics of the suspension modality are given in Figure 6.

An additional benefit of the suspension type is that it provides the means for embedding by-name PCF in by-value PCF. For example, we may distinguish the type of call-by-name partial functions from $\tau_1$ to $\tau_2$ as the type $\tau_1\text{ susp }\rightarrow \tau_2$, the type of functions that take a suspended computation as argument, whose value, if needed, is obtained using $\text{bind}$. No such converse embedding is possible, because of the insistence on divergence as a value in by-name PCF.

The upshot is that by-value PCF is strictly more expressive than by-name PCF, because it can account for both inductive and co-inductive types such as $\text{nat}$ and $\text{conat}$, and can account for call-by-name functions, as well as the “default” call-by-value form.
\[
\begin{align*}
\text{val} & \quad (2a) \\
\text{val}(e) & \quad (2b) \\
\lambda(x: \tau) e & \text{ val} \quad (2c) \\
\begin{array}{c}
e \mapsto e' \\
\text{ifz}\{\tau\}(e; e_0; x. e_1) \mapsto \text{ifz}\{\tau\}(e'; e_0; x. e_1)
\end{array} \quad (2d) \\
\begin{array}{c}
\text{ifz}\{\tau\}(z; e_0; x. e_1) \mapsto e_0 \\
\text{ifz}\{\tau\}(s(e); e_0; x. e_1) \mapsto [e/x]e_1
\end{array} \quad (2e) \\
\begin{array}{c}
e_1 \mapsto e_1' \\
e_1(e_2) \mapsto e_1'(e_2)
\end{array} \quad (2f) \\
(\lambda(x: \tau)e)(e_2) \mapsto [e_2/x]e \quad (2g) \\
\text{fix} x : \tau \\text{is } e \mapsto [\text{fix} x : \tau \text{is } e/x]e \quad (2h) \\
\end{align*}
\]
\[
\begin{align*}
\text{val} \quad & \quad \text{val} \\
\text{e val} \quad & \quad \text{e val} \\
\text{s(e) val} \quad & \quad \text{s(e) val}
\end{align*}
\]

(4a)  

\[
\begin{align*}
\text{fun } x(y: \tau_1): \tau_2 \text{ is e val} \\
e \mapsto e' \\
\text{ifz} \{ \tau \} (e; e_0; x.e_1) \mapsto \text{ifz} \{ \tau \} (e'; e_0; x.e_1)
\end{align*}
\]

(4c)  

(4d)  

\[
\begin{align*}
\text{ifz} \{ \tau \} (z; e_0; x.e_1) \mapsto e_0 \\
s(e) \text{ val} \\
\text{ifz} \{ \tau \} (s(e); e_0; x.e_1) \mapsto [e/x]e_1
\end{align*}
\]

(4e)  

(4f)  

\[
\begin{align*}
e_1 \mapsto e'_1 \\
e_1(e_2) \mapsto e'_1(e_2) \\
e_1 \text{ val} \\
e_2 \mapsto e'_2 \\
e_1(e_2) \mapsto e_1(e'_2)
\end{align*}
\]

(4g)  

(4h)  

\[
\begin{align*}
\text{e2 val} \\
\text{(fun } x(y: \tau_1): \tau_2 \text{ is e)(e_2) } \mapsto \text{[fun } x(y: \tau_1): \tau_2 \text{ is } e, e_2/x, y/e]\end{align*}
\]

(4i)  

Figure 4: **PCF By Value: Dynamics**

\[
\begin{align*}
\Gamma \vdash s: \text{conat} \\
\Gamma, x: \text{conat} \vdash e : \text{conat} \\
\Gamma \vdash s(x.e) : \text{conat} \\
\Gamma \vdash e \bowtie \text{conat} \quad \Gamma \vdash e_0 \bowtie \tau \\
\Gamma, x: \text{conat} \vdash e_1 \bowtie \tau \\
\Gamma \vdash \text{ifs}(e; e_0; x.e_1) \bowtie \tau \\
e \mapsto e' \\
\text{ifs}(e; e_0; x.e_1) \mapsto \text{ifs}(e'; e_0; x.e_1)
\end{align*}
\]

(5a)  

(5b)  

(5c)  

(5d)  

\[
\begin{align*}
\text{ifs}(s; e_0; x.e_1) \mapsto e_0 \\
\text{ifs}(s(y.e); e_0; x.e_1) \mapsto [s(y.e)/y][e/x]e_1
\end{align*}
\]

(5e)  

(5f)  

Figure 5: **PCF By Value: Co-Natural Numbers**

5
\[
\begin{align*}
\Gamma &\vdash e : \tau \\
\Gamma &\vdash \text{susp}(e) : \tau \text{susp} \\
\Gamma &\vdash e_1 : \tau \text{susp} \quad \Gamma, x : \tau_1 &\vdash e_2 : \tau_2 \\
\Gamma &\vdash \text{bind}x \leftarrow e_1 \text{in } e_2 : \tau_2 \\
\end{align*}
\] (6a)

\[e \mapsto e' \]

\[
\begin{align*}
\text{bind}x \leftarrow \text{susp}(e) \text{in } e_2 &\mapsto \text{bind}x \leftarrow \text{susp}(e') \text{in } e_2 \\
\text{val} \quad e \mapsto [e/x]e_2 \\
\end{align*}
\] (6c, 6d)

Figure 6: \textbf{PCF} By Value: Suspension Modality