PFPL Supplement: Lazy and Eager PCF

Robert Harper

October, 2018

1 Introduction

With the introduction of \textbf{PCF} we have introduced the possibility of divergence, or undefinedness, in a programming language. In contrast to languages such as \textbf{T} or \textbf{F} in which every well-typed expression terminates, in a language such as \textbf{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 \textit{computations} of a type, regardless of whether they terminate.

2. In a \textit{by-value} language variables range over the \textit{values} of a type, which are of course fully evaluated.

Put another way, variables only ever range over values, but in a by-name language all expressions are values, even the divergent ones, whereas in a by-value language only certain expressions are values, the rest are proper computations.
\[ \Gamma, x : \tau \vdash x : \tau \]  

(1a)

\[ \Gamma \vdash z : \text{nat} \]  

(1b)

\[ \Gamma \vdash e : \text{nat} \]  

(1c)

\[ \Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat} \vdash e_1 : \tau \]  

(1d)

\[ \Gamma \vdash \text{ifz}\{\tau\}(e; e_0; x.e_1) : \tau \]  

(1e)

\[ \Gamma, x : \tau_1 \vdash e : \tau_2 \]  

(1f)

\[ \Gamma \vdash \text{ap}(e_1; e_2) : \tau \]  

(1g)

\[ \Gamma, x : \tau \vdash e : \tau \]  

Figure 1: PCF By Name: Statics

2 PCF By Name

In a by-name language there is only one form of expression, which is defined by the typing judgment, \( \Gamma \vdash e : \tau \), stating that \( e \) is an expression of type \( \tau \). This judgment is defined in Figure 1. This judgment satisfies the following substitution principle stating that any expression may be substituted for a variable, regardless of whether it is convergent or not.

**Lemma 2.1 (By-Name Substitution).** If \( \Gamma \vdash e : \tau \) and \( \Gamma, x : \tau \vdash e' : \tau' \), then \( \Gamma \vdash [e/x]e' : \tau' \).

The by-name dynamics is defined (see Figure 2) so that arguments are passed to functions in unevaluated form, and the argument to a successor is evaluated only as necessary. These choices are only possible because variables range over any expression of their type. Be careful, though! Because variables can occur any number of times in an expression, substitution replicates work—the same expression might need to be evaluated many times. This is a win when the number of times happens to be zero, but otherwise work is duplicated.\(^1\)

More significantly, though, the by-name variant suffers from an irrepairable semantic defect. In particular, the expression \( \omega \triangleq \text{fix}\{\text{nat}\}(x.s(x)) \) has type \( \text{nat} \), even though it is not a natural number. It may be thought of as an infinite stack of successors, which is larger than any finite stack of successors. Thus, bizarrely, the principal of mathematical induction is not valid for the type \( \text{nat} \). Indeed, under the by-name interpretation the type \( \text{nat} \) should be renamed to \( \text{conat} \), the type of co-natural numbers. Whereas \( \text{nat} \) is the smallest type closed under zero and successor, the type \( \text{conat} \) is the largest type consistent with being either zero or the successor of another co-natural number. Thus \( \omega \) is a co-natural number, because it is the successor of another, namely itself.

It is well and good to have a type of co-natural numbers, but this does not address the absence of a type of natural numbers. In fact, there is no way to define the natural numbers in by-name

\(^1\)This is mitigated using by-need evaluation, which memorizes results to avoid duplication.
\[
\begin{align*}
\text{2a} & \quad \overline{z \text{ val}} \\
\text{2b} & \quad \overline{s(e) \text{ val}} \\
\text{2c} & \quad \overline{\lambda \{ \tau \}(x.e) \text{ val}} \\
& \quad e \mapsto e' \\
& \quad \text{if} \{ \tau \}(e;e_0;x.e_1) \mapsto \text{if} \{ \tau \}(e';e_0;x.e_1) \\
\text{2d} & \quad \text{if} \{ \tau \}(z;e_0;x.e_1) \mapsto e_0 \\
\text{2e} & \quad \text{if} \{ \tau \}(s(e);e_0;x.e_1) \mapsto [e/x]e_1 \\
& \quad e_1 \mapsto e'_1 \\
& \quad \text{ap}(e_1;e_2) \mapsto \text{ap}(e'_1;e_2) \\
\text{2f} & \quad \text{ap}(\lambda \{ \tau \}(x.e);e_2) \mapsto [e_2/x]e \\
\text{2g} & \quad \text{fix} \{ \tau \}(x.e) \mapsto \text{fix} \{ \tau \}(x.e)/x e
\end{align*}
\]

Figure 2: PCF By Name: Dynamics

PCF, nor is it possible to define any inductive type, not even the booleans! It is often suggested that the natural numbers may be defined in by-name PCF by making the successor strict (evaluate its argument). But this is not true either: the divergent expression \( \bot \triangleq \text{fix} \{ \text{nat} \}(x.x) \) inhabits the type nat, violating the principle of mathematical induction.

3 PCF By Value

In a by-value language there are two modes of expression, \( \Gamma \vdash e : \tau \) defining the open values, and \( \Gamma \vdash e \triangleq \tau \) for computations. These judgments are defined in Figure 3. The open values are defined to include variables, because these can only ever be replaced by other values. The computations are defined as those expressions that may require evaluation to determine their values. Therefore all values are computations, having already been evaluated. But notice that, in contrast to the by-name variant, fixed point computations are not permitted in general. Instead \( \lambda \)-abstraction is generalized to a self-referential function expression.

It is easy to check that value substitution preserves typing in the by-value setting.

Lemma 3.1 (By-Value Substitution). 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 \( \Gamma, x : \tau \vdash e' \triangleq \tau' \), then \( \Gamma \vdash [e/x]e' \triangleq \tau' \).
\( \Gamma, x : \tau \vdash x : \tau \) \hfill (3a)

\( \Gamma \vdash z : \text{nat} \) \hfill (3b)

\( \Gamma \vdash e : \text{nat} \)

\( \Gamma \vdash s(e) : \text{nat} \) \hfill (3c)

\( \Gamma, x : \tau_1 \rightarrow \tau_2, y : \tau_1 \vdash e \sim \tau_2 \)

\( \Gamma \vdash \text{fun}\{\tau_1; \tau_2\}(x.y.e) : \tau_1 \rightarrow \tau_2 \) \hfill (3d)

\( \Gamma \vdash e : \tau \)

\( \Gamma \vdash e \sim \tau \) \hfill (3e)

\( \Gamma \vdash e_0 \sim \tau \)

\( \Gamma, x : \text{nat} \vdash e_1 \sim \tau \)

\( \Gamma \vdash \text{ifz}\{\tau\}(e; e_0; x.e_1) \sim \tau \) \hfill (3f)

\( \Gamma \vdash e_1 \sim \tau_2 \rightarrow \tau \)

\( \Gamma \vdash e_2 \sim \tau_2 \)

\( \Gamma \vdash \text{ap}(e_1; e_2) \sim \tau \) \hfill (3g)

The corresponding dynamics is given in Figure 4. Function applications evaluate their argument before the call, and the successor is given an eager interpretation. Moreover, application of a self-referential function provides the function itself, a value, along with the argument value, to the body of the function, unrolling the recursion on demand. It would not be possible to give a by-value dynamics for general recursion, precisely because doing so would require substitution of a non-value for a variable.

The type \text{nat} in the by-value variant of \text{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 \text{conat} of co-natural numbers, which contains co-zero, written \( s \), a self-referential co-successor, written \( s(x.e) \), and an analogue of the zero-test.\(^2\) Their statics and dynamics are given in Figure 5. Notice that \( \omega \triangleq s(x.x) \) is the infinite co-natural number, and is a value of type \text{conat}. The conditional unrolls the recursion to determine the predecessor, which is a value.

### 4 Computation Modality

The distinction between values and computations in by-value \text{PCF} may be bridged by introducing the \text{computation modality}, \( \text{comp}(\tau) \). The type \( \text{comp}(\tau) \) is the type of unevaluated, or suspended, computations of type \( \tau \). Its elements are introduced by the expression \( \text{comp}(e) \), where \( e \) is an unevaluated computation, and eliminated by \( \text{let}(\tau; e_1.x)e_2 \), which forces the evaluation of the computation given by \( e_1 \) and passes its value, if any, to \( e_2 \). All sequencing of evaluation may be made explicit using the computation modality. In particular, there is no loss of generality in restricting

---

\(^2\)The notation is chosen to suggest that co-natural numbers are the natural numbers “backwards” in a certain sense.
\[
\begin{align*}
\text{val} & \quad (4a) \\
\text{val} & \quad (4b) \\
\text{val} & \quad \text{fun}\{\tau_1; \tau_2\}(x.y.e) & \quad (4c) \\
\tau & \mapsto e' & \quad (4d) \\
\text{ifz}\{\tau\}(e; e_0; x.e_1) & \mapsto \text{ifz}\{\tau\}(e'; e_0; x.e_1) & \quad (4e) \\
\text{ifz}\{\tau\}(z; e_0; x.e_1) & \mapsto e_0 & \quad (4f) \\
\text{ifz}\{\tau\}(s(e); e_0; x.e_1) & \mapsto [e/x]e_1 & \quad (4g) \\
\text{ifz}\{\tau\}(s(e); e_0; x.e_1) & \mapsto [e/x]e_1 & \quad (4h) \\
\text{ap}(e_1; e_2) & \mapsto \text{ap}(e'_1; e_2) & \quad (4i) \\
\end{align*}
\]

Figure 4: PCF By Value: Dynamics

\[
\begin{align*}
\text{conat} & \quad (5a) \\
\text{conat} & \quad (5b) \\
\text{conat} & \quad (5c) \\
\text{conat} & \quad (5d) \\
\text{conat} & \quad (5e) \\
\text{conat} & \quad (5f) \\
\end{align*}
\]

Figure 5: PCF By Value: Co-Natural Numbers
\[
\begin{align*}
\Gamma & \vdash e : \tau \\
\Gamma & \vdash \text{ret}(e) \not\sim \tau \\
\Gamma & \vdash e \not\sim \tau \\
\Gamma & \vdash \text{comp}(e) : \text{comp}(\tau) \\
\Gamma & \vdash e_1 : \text{comp}(\tau) \\
\Gamma, x : \tau_1 & \vdash e_2 \not\sim \tau_2 \\
\Gamma & \vdash \text{let}(e_1; x.e_2) \not\sim \tau_2 \\
& \quad \text{(6a)}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e \not\sim \tau \\
\Gamma & \vdash \text{comp}(e) : \text{comp}(\tau) \\
\Gamma, x : \tau_1 & \vdash e_2 \not\sim \tau_2 \\
\Gamma & \vdash \text{let}(e_1; x.e_2) \not\sim \tau_2 \\
\quad \quad \text{(6b)}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e_1 : \text{comp}(\tau) \\
\Gamma & \vdash \text{let}(e_1; x.e_2) \not\sim \tau_2 \\
\quad \quad \text{(6c)}
\end{align*}
\]

\[
\begin{align*}
e & \mapsto e' \\
\text{let}(\text{comp}(e); x.e_2) & \mapsto \text{let}(\text{comp}(e'); x.e_2) \\
\quad \quad \text{(6d)}
\end{align*}
\]

\[
\begin{align*}
e & \text{ val} \\
\text{let}(\text{comp}(e); x.e_2) & \mapsto [e/x]e_2 \\
\quad \quad \text{(6e)}
\end{align*}
\]

Figure 6: **PCF** By Value: Computation Modality

the principal arguments of elimination forms to open values, rather than general computations. The statics and dynamics of the computation modality in by-value **PCF** is given in Figure 6.

Using the computation type it is possible to encode by-name **PCF**. For example, the type of partial functions called by name is given by the type \(\text{comp}(\tau_1) \rightarrow \tau_2\), which takes a suspended computation, rather than a value, of the domain type as argument. Similarly, the co-natural numbers of by-name **PCF** may be interpreted into the by-value formulation using a successor whose argument is a suspended computation of a co-natural number.

References