PFPL Supplement: Lazy and Eager PCF

Robert Harper

January 2, 2019

1 Introduction

In contrast to total languages such as T or F, in which every well-typed expression evaluates to a value, PCF and FPC are partial in that well-typed expressions need not have a value (they may diverge). Correspondingly, functions in total languages are total (defined on every input of the domain type), but in partial languages are partial (may be undefined on some well-typed input).

It may seem advantageous to preclude infinite loops, a common manifestation of programming errors. But to insist on totality means that the type checking amounts to proving termination, and it is clear from Gödel’s Theorem that there is no way for any decidable (or even semi-decidable) type system to certify all possible terminating programs as such. Thus, a total language must be, at least theoretically, incomplete, and is certain to be awkward for programming, because the code must be formulated in such a way that its termination is evident to the type checker. Given that the termination proof of even a small program can be arbitrarily complicated, this means that convoluted programs for simple functions are inevitable.

By allowing partiality the proof of termination is shifted from the code itself to a separate verification process, off-loading the complexities of the termination argument.

In a total language substitution is unrestricted: any expression (of suitable type) may be substituted for a variable (of that type). This is valid because every expression is tantamount to a value, and its value is all that matters for the overall behavior of the program.\footnote{Its efficiency is another matter: replicating a computation by substitution for a variable with many occurrences may cause the same computation to be repeated at each occurrence.} However, in a partial language some care must be taken with substitution to account for non-termination. There are two approaches to handling these difficulties, the
by-name (aka lazy, or non-strict) interpretation and the by-value (aka eager or strict) interpretation. In both cases variables range only over values; the distinction lies in what counts as a value.

1. Under the by-name interpretation every expression is a value, regardless of whether it would terminate when evaluated. The advantage is that any non-termination is avoided as long as possible, until the value of an expression is required for completion of a computation. The disadvantage is that non-terminating expressions must be regarded as values of their type, hobbling the expressiveness of the language (for example, there are three booleans!)

2. Under the by-value interpretation all values are fully evaluated and cannot induce non-termination. The advantage is that types retain their natural meanings (for example, there are only two booleans). The disadvantage is that expressions may be evaluated even when their values are not needed to complete the computation.

Although it may seem that the two interpretations are incomparable, each with its own advantages, but this is not true. In fact by-name computation is readily encodable in a by-value language using a modality, but the converse is impossible. Thus, by-value languages are, from this perspective, strictly superior to by-name languages.

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.\footnote{This is mitigated using \textit{by-need} evaluation, which memorizes results to avoid duplication.}

More significantly, though, the by-name variant suffers from an irreparable semantic defect. In particular, the expression

$$\omega \triangleq \text{fix}\{\text{n at}\}(x.s(x))$$

has type \text{n at}, 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{n at}. Indeed, under the by-name interpretation the type \text{n at} should be renamed to \text{co-nat}, the type of \text{co-natural numbers}. Whereas \text{n at} is the \text{smallest} type closed under zero and successor, the type \text{co-nat} is the \text{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 \text{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 \text{PCF} by making the successor strict (evaluate its argument). But this is not true either: the divergent expression \(\bot \triangleq \text{fix}\{\text{n at}\}(x.x)\) nevertheless inhabits the type \text{n at}, violating the principle of mathematical induction.

### 3  \text{PCF By Value}

In a by-value language there are two modes of expression, \(\Gamma \vdash e : \tau\) defining the \textit{open values}, and \(\Gamma \vdash e \leadsto \tau\) for \textit{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 that have already been evaluated. In contrast to the by-name variant fixed point computations are not permitted. Instead self-reference is confined to the values of certain types. In particular, \(\lambda\)-abstractions are generalized to admit self-reference, so that,
for example, the factorial function is definable using a self-referential \( \lambda \)-abstraction, rather than by a combination of \( \lambda \)-abstraction and \texttt{fix} as it would be in the by-name setting.

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' \approx \tau', then \( \Gamma \vdash [e/x]e' \approx \tau' \).

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.

Open values that happen to be closed are values in the sense of the dynamics.

**Lemma 3.2 (Open Values).** If \( \vdash e : \tau \), then \( e \text{ val} \).

The proof is by a straightforward induction on the statics, the variable case being ruled out by the emptiness of the context.

The type \texttt{nat} in the by-value variant of \texttt{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 \texttt{conat} of co-natural numbers, which contains co-zero, written \( z \), a self-referential co-successor, written \( a(x.e) \), and an analogue of the zero-test.\(^3\) Their statics and dynamics are given in Figure 5. Notice that \( \omega \triangleq a(x.x) \) is the infinite co-natural number, and is a value of type \texttt{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 \texttt{PCF} may be bridged by introducing the \texttt{computation modality}, \texttt{comp} (\( \tau \)). The type \texttt{comp} (\( \tau \)) is the type of unevaluated, or suspended, computations of type

\(^3\)The notation is chosen to suggest that co-natural numbers are the natural numbers “backwards” in a certain sense.
τ. Its elements are introduced by the expression \( \text{comp}(e) \), where \( e \) is an unevaluated computation, and eliminated by \( \text{let}(e_1; x.e_2) \), which forces the evaluation of the computation given by \( e_1 \) and passes its value, if any, to \( e_2 \). The statics and dynamics of the computation modality in by-value \( \textbf{PCF} \) is given in Figure 6.

The computation modality may be used to express all sequencing of sub-computations: there is no loss of generality in restricting the principal arguments of elimination forms to open values. Thus, the statics of conditionals and applications may be reformulated so that the principal arguments are required to be values:

\[
\Gamma \vdash e : \tau \quad \Gamma \vdash e_0 \leadsto \tau \quad \Gamma, x: \text{nat} \vdash e_1 \leadsto \tau \\
\Gamma \vdash \text{ifz}\{\tau\}(e_1; e_0; x.e_1) \leadsto \tau \quad \text{(6a)}
\]

\[
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{ap}(e_1; e_2) \leadsto \tau \quad \text{(6b)}
\]

The dynamics is changed accordingly, there now being no need to evaluate the principal arguments of eliminators: rules 4d, 4g, and 4h are rendered otiose and may therefore be omitted.

Another benefit of the computation modality is that it is natural to encode the by-name variant of \( \textbf{PCF} \) within the modal formulation. For example, the type of call-by-name partial functions is 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 \( \textbf{PCF} \) may be interpreted into the by-value formulation using a successor whose argument is a computation of a co-natural number.

Similarly, the computation modality may be used to decompose the type of partial functions into a combination of total functions and computations:

\[
\tau_1 \rightarrow \tau_2 \triangleq \tau_1 \rightarrow \text{comp}(\tau_2).
\]

That is, a partial function from \( \tau_1 \) to \( \tau_2 \) is a total function that, when applied to a value of type \( \tau_1 \) yields a suspended computation of type \( \tau_2 \). It is up to the caller to then bind the result of that computation to a variable for use within the remaining part of the program. Thus, divergence is isolated to the elimination form of the modality.

The most significant advantage of the modal formulation lies in its generalization to account for parallelism (Harper, 2018b) and for exceptions (Harper, 2018a).
References


\[
\Gamma, x : \tau \vdash x : \tau \tag{1a}
\]
\[
\Gamma \vdash z : \text{nat} \tag{1b}
\]
\[
\Gamma \vdash e : \text{nat} \\
\Gamma \vdash s(e) : \text{nat} \tag{1c}
\]
\[
\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat} \vdash e_1 : \tau \\
\Gamma \vdash \text{ifz}\{\tau\}(e; e_0; x.e_1) : \tau \tag{1d}
\]
\[
\Gamma, x : \tau_1 \vdash e : \tau_2 \\
\Gamma \vdash \lambda\{\tau_1\}(x.e_2) : \tau_1 \to \tau_2 \tag{1e}
\]
\[
\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{ap}(e_1; e_2) : \tau \tag{1f}
\]
\[
\Gamma, x : \tau \vdash e : \tau \\
\Gamma \vdash \text{fix}\{\tau\}(x.e) : \tau \tag{1g}
\]

Figure 1: PCF By Name: Statics
\[
\begin{align*}
\text{\texttt{val}} & \quad (2a) \\
\text{\texttt{val}}(e) & \quad (2b) \\
\lambda \{ \tau \} (x.e) & \quad (2c) \\
\text{ifz}\{\tau\}(e; e_0; x.e_1) & \quad \text{ifz}\{\tau\}(e'; e_0; x.e_1) \quad (2d) \\
\text{ifz}\{\tau\}(z; e_0; x.e_1) & \quad e_0 \quad (2e) \\
\text{ifz}\{\tau\}(s(e); e_0; x.e_1) & \quad [e/x]e_1 \quad (2f) \\
\text{ap}(e_1; e_2) & \quad \text{ap}(e'_1; e_2) \quad (2g) \\
\text{ap}(\lambda \{ \tau \} (x.e); e_2) & \quad [e_2/x]e \quad (2h) \\
\text{fix}\{\tau\}(x.e) & \quad [\text{fix}\{\tau\}(x.e)/x]e \quad (2i)
\end{align*}
\]

Figure 2: PCF By Name: Dynamics
\[
\begin{align*}
\Gamma, x: \tau &\vdash x: \tau \quad (3a) \\
\Gamma &\vdash z: \text{nat} \\
\Gamma &\vdash e: \text{nat} \\
\Gamma &\vdash s(e): \text{nat} \\
\Gamma, x: \tau_1 \rightarrow \tau_2, y: \tau_1 &\vdash e \sim \tau_2 \\
\Gamma &\vdash \text{fun}\{\tau_1; \tau_2\}(x.y.e): \tau_1 \rightarrow \tau_2 \quad (3d) \\
\Gamma &\vdash e: \tau \\
\Gamma &\vdash e \sim \tau \\
\Gamma &\vdash e \sim \tau \quad \Gamma &\vdash e_0 \sim \tau \quad \Gamma, x: \text{nat} &\vdash e_1 \sim \tau \\
\Gamma &\vdash \text{ifz}\{\tau\}(e; e_0; x.e_1) \sim \tau \\
\Gamma &\vdash e_1 \sim \tau_2 \rightarrow \tau \quad \Gamma &\vdash e_2 \sim \tau_2 \\
\Gamma &\vdash \text{ap}(e_1; e_2) \sim \tau \\
\end{align*}
\]

Figure 3: PCF By Value: Statics
\[
\begin{align*}
\text{val} & \\
\text{fun} & \\
\text{s}(\text{e}) & \text{val}
\end{align*}
\]

\[(4a)\]

\[
\begin{align*}
\text{val} & \\
\text{ifz} & \rightarrow \text{e'}
\end{align*}
\]

\[(4b)\]

\[
\begin{align*}
\text{fun} & \rightarrow \text{fun} & \text{e} & \rightarrow \text{e'}
\end{align*}
\]

\[(4c)\]

\[
\begin{align*}
\text{ifz} & \rightarrow \text{ifz} & \text{e} & \rightarrow \text{e'}
\end{align*}
\]

\[(4d)\]

\[
\begin{align*}
\text{ifz} & \rightarrow \text{ifz} & \text{e} & \rightarrow \text{e'}
\end{align*}
\]

\[(4e)\]

\[
\begin{align*}
\text{ifz} & \rightarrow \text{ifz} & \text{e} & \rightarrow \text{e'}
\end{align*}
\]

\[(4f)\]

\[
\begin{align*}
\text{ifz} & \rightarrow \text{ifz} & \text{e} & \rightarrow \text{e'}
\end{align*}
\]

\[(4g)\]

\[
\begin{align*}
\text{ifz} & \rightarrow \text{ifz} & \text{e} & \rightarrow \text{e'}
\end{align*}
\]

\[(4h)\]

\[
\begin{align*}
\text{ifz} & \rightarrow \text{ifz} & \text{e} & \rightarrow \text{e'}
\end{align*}
\]

\[(4i)\]

Figure 4: PCF By Value: Dynamics
\[ \Gamma \vdash s : \text{conat} \] (5a)

\[ \Gamma, x : \text{conat} \vdash e : \text{conat} \]
\[ \Gamma \vdash s(x.e) : \text{conat} \] (5b)

\[ \Gamma \vdash e \sim \text{conat} \quad \Gamma \vdash e_0 \sim \tau \quad \Gamma, x : \text{conat} \vdash e_1 \sim \tau \]
\[ \Gamma \vdash \text{if}_e(e; e_0; x.e_1) \sim \tau \] (5c)

\[ e \mapsto e' \]
\[ \text{if}_e(e; e_0; x.e_1) \mapsto \text{if}_e(e'; e_0; x.e_1) \] (5d)

\[ \text{if}_e(s;x.e_1) \mapsto e_0 \] (5e)

\[ \text{if}_e(a(y.e); e_0; x.e_1) \mapsto [a(y.e)/y][e/x]e_1 \] (5f)

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

\[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash \text{ret}(e) \sim \tau \] (7a)

\[ \Gamma \vdash e \sim \tau \]
\[ \Gamma \vdash \text{comp}(e) : \text{comp}(\tau) \] (7b)

\[ \Gamma \vdash e_1 : \text{comp}(\tau) \quad \Gamma, x : \tau_1 \vdash e_2 \sim \tau_2 \]
\[ \Gamma \vdash \text{let}(e_1; x.e_2) \sim \tau_2 \] (7c)

\[ e \mapsto e' \]
\[ \text{let}(\text{comp}(e); x.e_2) \mapsto \text{let}(\text{comp}(e'); x.e_2) \] (7d)

\[ e \text{ val} \]
\[ \text{let}(\text{comp}(e); x.e_2) \mapsto [e/x]e_2 \] (7e)

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