PFPL Supplement: Lazy and Eager PCF

Robert Harper
September 17, 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. 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

\[1\text{Its efficiency is another matter: replicating a computation by substitution for a variable with many occurrences may cause the same computation to be repeated at each occurrence.}\]
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

The syntax of expressions is that of PCF. In addition a program, $p$, is of the form $\text{ret}(e)$, where $e$ is an expression. The statics defines two forms of typing judgment:

1. $\Gamma \vdash e : \tau$, $e$ is an open value of type $\tau$ under $\Gamma$;

2. $\Gamma \vdash p \sim \tau$, $p$ is a computation of type $\tau$ under $\Gamma$.

The rules defining these judgments are given in Figure 1.

These judgments satisfy the following substitution principles according to which variables may be replaced by values preserving typing.

**Lemma 2.1** (By-Name 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 $\Gamma, x : \tau \vdash p \sim \tau'$, then $\Gamma \vdash [e/x]p \sim \tau'$.
\begin{align*}
\Gamma, x : \tau & \vdash x : \tau & \text{(1a)} \\
\Gamma & \vdash z : \text{nat} & \text{(1b)} \\
\Gamma & \vdash e : \text{nat} \\
\Gamma & \vdash s(e) : \text{nat} & \text{(1c)} \\
\Gamma & \vdash e : \text{nat} & \Gamma & \vdash e_0 : \tau & \Gamma, x : \text{nat} & \vdash e_1 : \tau \\
\Gamma & \vdash \text{ifz}\{\tau\}(e; e_0; x.e_1) : \tau & \text{(1d)} \\
\Gamma, x : \tau_1 & \vdash e : \tau_2 \\
\Gamma & \vdash \lambda\{\tau_1\}(x.e_2) : \tau_1 \rightarrow \tau_2 & \text{(1e)} \\
\Gamma & \vdash e_1 : \tau_2 \rightarrow \tau & \Gamma & \vdash e_2 : \tau_2 \\
\Gamma & \vdash \text{ap}(e_1; e_2) : \tau & \text{(1f)} \\
\Gamma, x : \tau & \vdash e : \tau \\
\Gamma & \vdash \text{fix}\{\tau\}(x.e) : \tau & \text{(1g)} \\
\Gamma & \vdash e : \tau \\
\Gamma & \vdash \text{ret}(e) \sim \tau & \text{(1h)} \\
\end{align*}

Figure 1: PCF By Name: Statics
The by-name dynamics is defined on programs to reduce the given expression until it is no longer reducible, thereby obtaining the “answer” of the computation. Because all expressions are values under by-name, there is nothing to force their evaluation unless induced by the need to determine the answer of a complete program. Doing so may incur divergence, but otherwise there is nothing to force the evaluation of expressions.\textsuperscript{2} The dynamics of PCF by-name is given in Figure 2.

Notice that the arguments to functions are passed without evaluation (because they are already values!) Doing so replicates work that may have to be done when the complete program is executed. If the parameter is not used, the argument is never evaluated, but it can be that the same expression is evaluated more than once. Such duplicate effort can be mitigated using memoization; see Chapter 36 for specifics.

The by-name variant of PCF has an appealing simplicity. However, it suffers from an irrepairable semantic defect. In particular, the expression

\[ \omega \triangleq \text{fix}\{\text{nat}\}(x.x(x)) \]

has type nat, even though it is not a natural number. It may be thought of as an infinite stack of successors, which is thereby larger than any finite stack of successors—it is an “infinite” natural number! Consequently, the principle of mathematical induction is not valid for the type nat. It is instead the type of co-natural numbers, which includes \( \omega \) as a value. Whereas the type nat is the smallest type closed under zero and successor, the type 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. (Note well the “circularity” of this observation!)

Although it is well and good to have a type of co-natural numbers, having these does not mitigate the absence of a type of natural numbers. Is it merely an oversight? No! There is no way to define the natural numbers in by-name PCF. In fact, it is impossible to define any inductive type whatsoever, not even the booleans! This is a fundamental semantic deficiency of by-name languages.\textsuperscript{3}

\textsuperscript{2}Compare the interactive top-level in the GHC Haskell compiler, which arbitrarily forces values of so-called monadic type.

\textsuperscript{3}It is often suggested that if the successor were made to evaluate its argument, then the natural numbers would be recovered. But that is not true: the divergent expression \( \bot \triangleq \text{fix}\{\text{nat}\}(x.x) \) is nevertheless a value of the type, violating the principle of mathematical induction.
\[
e \mapsto e'
\]

\[
\text{ifz}\{\tau\}(e; e_0; x.e_1) \mapsto \text{ifz}\{\tau\}(e'; e_0; x.e_1)
\]  
(2a)

\[
\text{ifz}\{\tau\}(z; e_0; x.e_1) \mapsto e_0
\]  
(2b)

\[
\text{ifz}\{\tau\}(s(e); e_0; x.e_1) \mapsto [e/x]e_1
\]  
(2c)

\[
e_1 \mapsto e'_1
\]

\[
ap(e_1; e_2) \mapsto ap(e'_1; e_2)
\]  
(2d)

\[
ap(\lambda\{\tau\}(x.e); e_2) \mapsto [e_2/x]e
\]  
(2e)

\[
\text{fix}\{\tau\}(x.e) \mapsto [\text{fix}\{\tau\}(x.e)/x]e
\]  
(2f)

\[
e \mapsto \text{ret}(e)\text{ done}
\]  
(2g)

\[
e \mapsto e'
\]

\[
\text{ret}(e) \mapsto \text{ret}(e')
\]  
(2h)

Figure 2: PCF By Name: Dynamics
3 PCF By Value

In contrast to the by-name case, the by-value variant of PCF has relatively few values, and relatively many computations. The statics of by-value PCF is given in Figure 3. Open values include variables, as before, but are otherwise limited to zero, the successor of an open value, and a (self-referential) \( \lambda \)-abstraction. In contrast to the by-name variant fixed point computations are not considered; instead, \( \lambda \)-abstractions are generalized to admit self-reference.

Value substitution preserves typing in the by-value setting as well.

**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 \( \Gamma, x : \tau \vdash e' \sim \tau' \), then \( \Gamma \vdash [e/x]e' \sim \tau' \).

The dynamics of by-value PCF 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:

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

The type \( \text{nat} \) in the by-value variant of PCF is indeed the type of natural numbers; its closed values are precisely the numerals \( z, s(z), \) and so on. It is also possible to define a type \( \text{conat} \) 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.\(^4\) 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 \( \text{conat} \). 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 PCF may be bridged by introducing the *computation modality*, \( \text{comp}(\tau) \). The type

\(^4\)The notation is chosen to suggest that co-natural numbers are the natural numbers “backwards” in a certain sense.
\( \Gamma, x : \tau \vdash x : \tau \quad (3a) \)

\[ \Gamma \vdash z : \text{nat} \quad (3b) \]

\[ \Gamma \vdash e : \text{nat} \]
\[ \Gamma \vdash s(e) : \text{nat} \]

\[ \Gamma, x : \tau_1 \rightarrow \tau_2, y : \tau_1 \vdash e \sim \tau_2 \]
\[ \Gamma \vdash \text{fun}\{\tau_1; \tau_2\}(x.y.e) : \tau_1 \rightarrow \tau_2 \quad (3d) \]

\[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash e \sim \tau \quad (3e) \]

\[ \Gamma \vdash e \sim \tau \quad (3f) \]
\[ \Gamma \vdash e_0 \sim \tau \]
\[ \Gamma, x : \text{nat} \vdash e_1 \sim \tau \]
\[ \Gamma \vdash \text{ifz}\{\tau\}(e;e_0;x.e_1) \sim \tau \]

\[ \Gamma \vdash e_1 \sim \tau_2 \rightarrow \tau \quad (3g) \]
\[ \Gamma \vdash e_2 \sim \tau_2 \]
\[ \Gamma \vdash \text{ap}(e_1;e_2) \sim \tau \]

Figure 3: PCF By Value: Statics
\( \overline{z \text{ val}} \) \hspace{1cm} (4a)

\( \overline{e \text{ val}} \)

\( \overline{s(e) \text{ val}} \) \hspace{1cm} (4b)

\( \overline{\text{fun}\{\tau_1;\tau_2\}(x.y.e) \text{ val}} \) \hspace{1cm} (4c)

\( e \mapsto e' \)

\( \text{ifz}\{\tau\}(e;e_0;x.e_1) \mapsto \text{ifz}\{\tau\}(e';e_0;x.e_1) \) \hspace{1cm} (4d)

\( \text{ifz}\{\tau\}(z;e_0;x.e_1) \mapsto e_0 \) \hspace{1cm} (4e)

\( \overline{\text{s(e) val}} \)

\( \text{ifz}\{\tau\}(s(e);e_0;x.e_1) \mapsto [e/x]e_1 \) \hspace{1cm} (4f)

\( e_1 \mapsto e'_1 \)

\( \text{ap}(e_1;e_2) \mapsto \text{ap}(e'_1;e_2) \) \hspace{1cm} (4g)

\( e_1 \text{ val} \quad e_2 \mapsto e'_2 \)

\( \text{ap}(e_1;e_2) \mapsto \text{ap}(e_1;e'_2) \) \hspace{1cm} (4h)

\( e_2 \text{ val} \)

\( \text{ap}(\text{fun}\{\tau_1;\tau_2\}(x.y.e);e_2) \mapsto [\text{fun}\{\tau_1;\tau_2\}(x.y.e),e_2/x,y]e \) \hspace{1cm} (4i)

Figure 4: PCF By Value: Dynamics
\[\begin{align*}
\Gamma \vdash \mathsf{s} : \text{conat} \\
\Gamma, x : \text{conat} \vdash e : \text{conat} \\
\Gamma \vdash \mathsf{a}(x.e) : \text{conat}
\end{align*}\]  \hspace{1cm} (5a)

\[\begin{align*}
\Gamma \vdash e \triangleright \text{conat} & \quad \Gamma \vdash e_0 \triangleright \tau & \quad \Gamma, x : \text{conat} \vdash e_1 \triangleright \tau \\
\hline
\Gamma \vdash \text{if} z \{ \tau \} \{ e ; e_0 ; x.e_1 \} \triangleright \tau
\end{align*}\]  \hspace{1cm} (5b)

\[\begin{align*}
e & \mapsto e' \\
\text{if} \mathsf{s}(e ; e_0 ; x.e_1) & \mapsto \text{if} \mathsf{s}(e' ; e_0 ; x.e_1)
\end{align*}\]  \hspace{1cm} (5c)

\[\begin{align*}
\text{if} \mathsf{s}(s ; e_0 ; x.e_1) & \mapsto e_0
\end{align*}\]  \hspace{1cm} (5d)

\[\begin{align*}
\text{if} \mathsf{s}(a(y.e) ; e_0 ; x.e_1) & \mapsto \left[\overline{a(y.e) / y / x} / e_1\right]
\end{align*}\]  \hspace{1cm} (5e)

\[\begin{align*}
\text{if} \mathsf{s}(a(y.e) ; e_0 ; x.e_1) & \mapsto \left[\overline{a(y.e) / y / x} / e_1\right]
\end{align*}\]  \hspace{1cm} (5f)

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

\(\text{comp}(\tau)\) 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 \(\text{let}(e_1 ; x.e_2)\), 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 **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:

\[\begin{align*}
\Gamma \vdash e : \tau & \quad \Gamma \vdash e_0 \triangleright \tau & \quad \Gamma, x : \text{nat} \vdash e_1 \triangleright \tau \\
\hline
\Gamma \vdash \text{if} \{ \tau \} \{ e ; e_0 ; x.e_1 \} \triangleright \tau
\end{align*}\]  \hspace{1cm} (6a)

\[\begin{align*}
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau & \quad \Gamma \vdash e_2 : \tau_2 \\
\hline
\Gamma \vdash \text{ap}(e_1 ; e_2) \triangleright \tau
\end{align*}\]  \hspace{1cm} (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.
\[
\begin{align*}
\Gamma \vdash e : \tau \\
\Gamma \vdash \text{ret}(e) \dashv \tau \\
\Gamma \vdash e \dashv \tau \\
\Gamma \vdash \text{comp}(e) : \text{comp}(\tau) \\
\Gamma \vdash e_1 : \text{comp}(\tau), \Gamma, x : \tau_1 \vdash e_2 \dashv \tau_2 \\
\Gamma \vdash \text{let}(e_1; x.e_2) \dashv \tau_2 \\
\text{let}(\text{comp}(e); x.e_2) \mapsto \text{let}(\text{comp}(e'); x.e_2) \\
\text{let}(\text{comp}(e); x.e_2) \mapsto [e/x]e_2
\end{align*}
\]

Figure 6: PCF By Value: Computation Modality

Another benefit of the computation modality is that it is natural to encode the by-name variant of 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 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


