# PFPL Supplement: PCF By Value*

Robert Harper

Fall, 2019

## 1 Introduction

Languages such as T or F are *total* in that every well-formed expression has a unique value. Thus, the fact that a program is well-typed suffices to ensure its termination. But typing is only a sufficient condition for termination; it can be onerous to formulate a program in a total language. Both theoretical and practical experience shows that it will ever be thus, for the simple reason that a given type system can only encode certain termination proofs, and cannot capture them all.

Languages such as PCF are *partial* in that a well-typed program need not terminate. In fact, in such languages it is rather easy to write programs that don't. More importantly, it is easy to write programs that do terminate, but whose proof of termination is not easily captured by a type system. It becomes the responsibility of programmers to ensure that their programs terminate as part of the larger problem of proving that they work as intended.

When non-termination is possible the distinction between evaluation orders for expressions becomes significant. For example, the application of a constant function that does not examine its argument will terminate under by-name, even when applied to a divergent argument, but will be divergent under by-value under such conditions. Indeed, such considerations motivated the development of *lazy* languages, which seek to avoid evaluation of expressions unless their values are certainly needed to determine the outcome of execution.

To achieve laziness it is necessary to treat *unevaluated computations* as if they were values. In particular, to allow for examples such as the constant

function, it is necessary to substitute possibly divergent computations for variables. This means that variables are not just placeholders for expressions that *have* values, but also for expressions that *do not*. But how can a placeholder hold a place for nothing at all? The trick is to arrange that undefined expressions are tantamount to values of their type, and can be substituted for variables. Unfortunately, doing so has the undesirable consequence that there are *three* booleans, rather than two, the two expected values, plus the special undefined value. Consequently, it is not valid to reason by cases on whether a boolean variable is either `true` or `false`! Similarly, the putative type of natural numbers has as elements not just the numerals, but also the undefined value, values built from it by application of successor, and a value consisting of an infinite stack of successors! The principle of mathematical induction is wildly inapplicable for such a type– in fact, there cannot be a type of natural numbers, or booleans, in such a setting!

By contrast by-value, or *eager*, languages consider variables to range only over values of their type. Thus, there are two booleans, and mathematical induction is valid for the type `nat`. The formulation of **PCF** given in *PFPL* does not bring out this fundamental difference between the by-value and by-name interpretations of the language. To bring this out properly it is helpful to reformulate **PCFv** that draws a *modal* distinction between values and computations. In this formulation variables range only over values, and the modality makes explicit the requirement that computations be evaluated before being used. The modal formulation has the addditional advantage that it scales naturally to account for exceptions, a form of computation that requires explicit sequencing, and for parallelism, which requires a generalization of the modality to allow for simultaneous evaluation of computations. (These extensions are described in described in Harper (2019) and Harper (2018).)

*Acknowledgement:* Thanks to Todd Wilson for helpful discussions about laziness and eagerness in **PCF**.

## 2 PCFv

The statics of **PCFv** draws a distinction between two *modes* of expression, *values* and *computations*. This separation is expressed in the statics by two forms of typing judgment, $\Gamma \vdash v : \tau$ for values and $\Gamma \vdash e \div \tau$ for computations, which are defined in Figure 1. General recursion is omitted by design; correspondingly, function values may be self-referential.

The meaning of variables is given by the following substitution principles:

**Lemma 2.1.** *1. If* $\Gamma \vdash v : \tau$ *and* $\Gamma, x : \tau \vdash v' : \tau'$, *then* $\Gamma \vdash [v/x]v' : \tau'$.

*2. If* $\Gamma \vdash v : \tau$ *and* $\Gamma, x : \tau \vdash e' \mathrel{\dot{\div}} \tau'$, *then* $\Gamma \vdash [v/x]e' \mathrel{\dot{\div}} \tau'$.

Thus, only values may be substituted for variables, whether in another value, or in a computation.

The dynamics of **PCFv** is given in Figure 2. 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.

It may appear, at first glance, that there is no way to compute the successor of anything other than a value. For example, if $e \mathrel{\dot{\div}} \mathtt{nat}$, then how is it possible to compute its successor? The apparent difficulty is that the typing rule for the successor demands that its argument be a value, and not a computation, so the expression $\mathtt{s}(\,e\,)$ is ill-formed when $e$ is a computation that is not also a value. The solution is to sequence the evaluation of the computation $e$ before the formation of the successor, which we may write as $\mathtt{let}(\,e\,;x\,.\,\mathtt{s}(\,x\,)\,)$.

But how is the `let` to be defined? There are two possibilities in **PCFv**, one uses a function, the other uses a conditional:

1. Define $\mathtt{let}(\,e\,;x\,.\,\mathtt{s}(\,x\,)\,)$ to be the application
$$\mathtt{ap}(\,\lambda\{\mathtt{nat}\}(\,x\,.\,\mathtt{s}(\,x\,)\,)\,;e\,).$$
2. Define $\mathtt{let}(\,e\,;x\,.\,\mathtt{s}(\,x\,)\,)$ to be the conditional
$$\mathtt{ifz}\{\mathtt{nat}\}(\,e\,;\mathtt{s}(\,\mathtt{z}\,)\,;x\,.\,\mathtt{s}(\,\mathtt{s}(\,x\,)\,)\,).$$

The first solution is more general in that it may be used to define $\mathtt{let}(\,e_1\,;x\,.\,e_2\,)$ in general in terms of application, whereas the second is applicable only to computations of natural numbers.

**Theorem 2.2** (Safety for **PCFv**). *1. If* $v : \tau$, *then* $v$ *val.*

*2. If* $e \mathrel{\dot{\div}} \tau$, *then either* $e : \tau$ *or* $e \longmapsto e'$ *for some* $e'$ *such that* $e' \mathrel{\dot{\div}} \tau$.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{1b}$$

$$\frac{\Gamma \vdash v : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(\, v \,) : \mathtt{nat}} \tag{1c}$$

$$\frac{\Gamma, x : \tau_1 \rightharpoonup \tau_2, y : \tau_1 \vdash e \div \tau_2}{\Gamma \vdash \mathtt{fun}\{\tau_1 \,;\, \tau_2\}(\, x \,.\, y \,.\, e \,) : \tau_1 \rightharpoonup \tau_2} \tag{1d}$$

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash v \div \tau} \tag{1e}$$

$$\frac{\Gamma \vdash e \div \mathtt{nat} \quad \Gamma \vdash e_0 \div \tau \quad \Gamma, x : \mathtt{nat} \vdash e_1 \div \tau}{\Gamma \vdash \mathtt{ifz}\{\tau\}(\, e \,;\, e_0 \,;\, x \,.\, e_1 \,) \div \tau} \tag{1f}$$

$$\frac{\Gamma \vdash e_1 \div \tau_2 \rightharpoonup \tau \quad \Gamma \vdash e_2 \div \tau_2}{\Gamma \vdash \mathtt{ap}(\, e_1 \,;\, e_2 \,) \div \tau} \tag{1g}$$

Figure 1: **PCFv**: Statics

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{2a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{s}(\ e\ )\ \mathsf{val}} \tag{2b}$$

$$\frac{}{\mathtt{fun}\{\tau_1\,;\tau_2\}(\ x\,.\,y\,.\,e\ )\ \mathsf{val}} \tag{2c}$$

$$\frac{e \longmapsto e'}{\mathtt{ifz}\{\tau\}(\ e\,;e_0\,;x\,.\,e_1\ ) \longmapsto \mathtt{ifz}\{\tau\}(\ e'\,;e_0\,;x\,.\,e_1\ )} \tag{2d}$$

$$\frac{}{\mathtt{ifz}\{\tau\}(\ \mathtt{z}\,;e_0\,;x\,.\,e_1\ ) \longmapsto e_0} \tag{2e}$$

$$\frac{\mathtt{s}(\ e\ )\ \mathsf{val}}{\mathtt{ifz}\{\tau\}(\ \mathtt{s}(\ e\ )\,;e_0\,;x\,.\,e_1\ ) \longmapsto [e/x]e_1} \tag{2f}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(\ e_1\,;e_2\ ) \longmapsto \mathtt{ap}(\ e_1'\,;e_2\ )} \tag{2g}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(\ e_1\,;e_2\ ) \longmapsto \mathtt{ap}(\ e_1\,;e_2'\ )} \tag{2h}$$

$$\frac{e_2\ \mathsf{val}}{\mathtt{ap}(\ \mathtt{fun}\{\tau_1\,;\tau_2\}(\ x\,.\,y\,.\,e\ )\,;e_2\ ) \longmapsto [\mathtt{fun}\{\tau_1\,;\tau_2\}(\ x\,.\,y\,.\,e\ ),e_2/x,y]e} \tag{2i}$$

Figure 2: **PCFv**: Dynamics

# 3 Computation Modality

$$\frac{\Gamma \vdash e \mathrel{\dot{\div}} \tau}{\Gamma \vdash \mathtt{comp}(\, e \,) : \mathtt{comp}(\, \tau \,)} \tag{3a}$$

$$\frac{\Gamma \vdash v_1 : \mathtt{comp}(\, \tau_1 \,) \quad \Gamma, x : \tau_1 \vdash e_2 \mathrel{\dot{\div}} \tau_2}{\Gamma \vdash \mathtt{bnd}(\, v_1 \,;\, x \,.\, e_2 \,) \mathrel{\dot{\div}} \tau_2} \tag{3b}$$

$$\frac{e \longmapsto e'}{\mathtt{bnd}(\, \mathtt{comp}(\, e \,) \,;\, x \,.\, e_2 \,) \longmapsto \mathtt{bnd}(\, \mathtt{comp}(\, e' \,) \,;\, x \,.\, e_2 \,)} \tag{3c}$$

$$\frac{e \ \mathsf{val}}{\mathtt{bnd}(\, \mathtt{comp}(\, e \,) \,;\, x \,.\, e_2 \,) \longmapsto [e/x]e_2} \tag{3d}$$

Figure 3: **PCFv**: Computation Modality

The distinction between the by-name and by-value formulations of **PCF** is in the meanings of variables. In the by-name case variables range over unevaluated computations; in the by-value case variables range only over values. When variables range over computations it makes sense to admit $\mathtt{fix}\{\tau\}(\, x \,.\, e \,)$ at any type—even if it is divergent when evaluated, it still makes sense to form the unrolling, $[\mathtt{fix}\{\tau\}(\, x \,.\, e \,)/x]e$. But when variables range over values, it is not sensible to perform this substitution. Instead, recursive functions are defined as values, and recursion is unrolled when such a function is applied, using only value substitution.

The two concepts of variables may be reconciled in **PCFv** by introducing the type $\mathtt{comp}(\, \tau \,)$ whose values are encapsulated computations of the form $\mathtt{comp}(\, e \,)$ in which $e$ is an unevaluated computation. The elimination form for the modality is the computation $\mathtt{bnd}(\, e_1 \,;\, x \,.\, e_2 \,)$, which evaluates the encapsulated computation $e_1$, then passes its value to the computation $e_2$. The statics and dynamics of this extension of **PCFv** are given in Figure 3.

The "base case" for computations is given by Rule (1e) in which values are implicitly regarded as degenerate computations. An alternative is to make this inclusion explicit by replacing Rule 1e with the following rule:

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathtt{ret}(\, v \,) \mathrel{\dot{\div}} \tau} \,. \tag{4}$$

When closed, the computation $\mathtt{ret}(\, e \,)$ is deemed a final state in the dynamics.

The computation modality may be used to define `let` by the equation

$$\texttt{let}(\, e_1\,;x_1\,.\,e_2\,) \triangleq \texttt{bnd}(\,\texttt{comp}(\,e_1\,)\,;x_1\,.\,e_2\,),$$

with derived statics

$$\frac{\Gamma \vdash e_1 \mathrel{\div\!\!\cdot} \tau_1 \qquad \Gamma, x_1 : \tau_1 \vdash e_2 \mathrel{\div\!\!\cdot} \tau_2}{\Gamma \vdash \texttt{let}(\, e_1\,;x\,.\,e_2\,) \mathrel{\div\!\!\cdot} \tau_2}\,.$$

With this in mind there is clearly no loss of generality in restricting the principal arguments of conditionals and applications to be values:

$$\frac{\Gamma \vdash v : \texttt{nat} \quad \Gamma \vdash e_0 \mathrel{\div\!\!\cdot} \tau \quad \Gamma, x : \texttt{nat} \vdash e_1 \mathrel{\div\!\!\cdot} \tau}{\Gamma \vdash \texttt{ifz}\{\tau\}(\, v\,;e_0\,;x\,.\,e_1\,) \mathrel{\div\!\!\cdot} \tau} \tag{5a}$$

$$\frac{\Gamma \vdash v_1 : \tau_2 \rightharpoonup \tau \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \texttt{ap}(\, v_1\,;v_2\,) \mathrel{\div\!\!\cdot} \tau} \tag{5b}$$

The dynamics may be correspondingly simplified by eliminating rules (2d), (2g), and (2h). The original forms are definable using `let`:

$$\begin{aligned}
\texttt{s}^*(\, e\,) &\triangleq \texttt{let}(\, e\,;x\,.\,\texttt{s}(\, x\,)\,)\\
\texttt{ifz}^*\{\tau\}(\, e\,;e_0\,.\,x\,) &\triangleq \texttt{let}(\, e\,;y\,.\,\texttt{ifz}\{\tau\}(\, y\,;e_0\,;x\,.\,e_1\,)\,)\\
\texttt{ap}^*(\, e_1\,;e_2\,) &\triangleq \texttt{let}(\, e_1\,;x_1\,.\,\texttt{let}(\, e_2\,;x_2\,.\,\texttt{ap}(\, x_1\,;x_2\,)\,)\,)
\end{aligned}$$

When the inclusion of values as computations is made explicit, it is similarly convenient to define

$$\texttt{ret}^*(\, e\,) \triangleq \texttt{let}(\, e\,;x\,.\,\texttt{ret}(\, x\,)\,).$$

It is easy to check that these derived forms give rise to the expected statics and dynamics.

The purpose of the modal formulation is to separate the sequencing of evaluation of the principal arguments of an elimination form from the crucial rule defining the action of an elimination form on an introductory form. For writing examples it is rather tedious to use the modal formulation directly; the starred forms given above allow for principal arguments to be computations, not just values, which is much more convenient.

## References


Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Types and parallelism. Supplement to Harper (2016), Fall 2018. URL https://www.cs.cmu.edu/~rwh/pfpl/supplements/par.pdf.

Robert Harper. Exceptions: Control and data. Supplement to Harper (2016), Fall 2019. URL https://www.cs.cmu.edu/~rwh/pfpl/supplements/exceptions.pdf.

Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, TLCA '99, pages 228–242, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-65763-0. URL http://dl.acm.org/citation.cfm?id=645894.671755.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. doi: 10.1017/S0960129501003322. URL https://doi.org/10.1017/S0960129501003322.