PFPL Supplement: PCF By-Name*

Robert Harper

Fall 2021

1 Introduction

The by-name formulation of PCF, notated PCFn, specifies that variables range over (potentially unevaluated) computations, and that the successor operation be evaluated lazily. Correspondingly, applications are evaluated “by name” in the sense that arguments are passed in unevaluated form. When enriched with products, sums, and recursive types, the same principles apply, so that pairs, injections, and folds are values irrespective of the evaluation status of their constituent expressions.

The interest in PCF stems from its allegedly “mathematical” nature in that, for example, functions are applied by substitution, and projections from pairs evaluate to the selected component, irrespective of its evaluation status. However, the correspondence with mathematics works only for the negative connectives (functions, products), and does not apply to their positive counterparts. In particular the type nat of PCFn is not the type of natural numbers, for it contains the fixed point of successor, a “point at infinity” that the natural numbers lack. There is no recourse. Even if successor were eager, it would still be the case that the fixed point of the identity, commonly called “bottom,” is a value of every type. The situation with PCFv is exactly dual: the positive types are mathematically well-behaved, but the negatives are not. The tie-breaker is that PCFv is equipped with a type of unevaluated computations, accounting for laziness, but there is no corresponding representation of eagerness within PCFn.

The comparison between the two on “mathematical” grounds thus fails. That can only be expected, because in mathematical contexts there are no “undefined” or “looping” or “infinite” expressions, precisely the ones that give rise to the discrepancy between by-name and by-value. A more appropriate comparison is based on efficiency. PCFn and its extensions are often described as “lazy languages,” with the implication that they do as little work as possible to achieve a result. In particular if a computation is never used, it is never evaluated, saving work. Superficially, this seems like an advantage, but as the above arguments show, it is a semantic liability. Leaving that aside, PCFn is not as lazy as possible: if a computation is replicated, it is re-evaluated each time it is used, wasting effort. To avoid this, lazy languages make use of a by need dynamics that avoids such repetitions using memoization, a form of mutable storage. Memoization trades space for time; the storage overhead is substantial, and can be crippling, but without it, PCFn is all but useless.

The purpose of this supplement is to define the by-need dynamics of PCFn without recourse to state, which needlessly complicates matters. The trick is to use non-determinism to “guess” (infallibly!) whether a computation will ever be needed to determine the final answer of a program.

*© 2023 Robert Harper. All Rights Reserved.
Doing so separates the concept from its implementation using state, as detailed in Chapter 36 of PFPL.

# 2 PCFn By-Need

It is convenient to extend PCFn with the expression \( \text{let}(e_1; x . e_2) \), which binds the variable \( x \) within \( e_2 \) to the expression \( e_1 \). The dynamics of ifz and app is re-defined in terms of let as follows:

\[
\begin{align*}
\text{LET-SUBST} & : \quad \text{let}(e_1; x . e_2) \mapsto \{e_1/x\}e_2 \\
\text{LET-STEP} & : \quad e_1 \mapsto e_1' \\
\end{align*}
\]

The extension with sum types would be handled similarly to the conditional.

With this alteration in place the by-need dynamics of PCFn can be given using two transition rules for let's:

\[
\begin{align*}
\text{LET-SUBST} & : \quad \text{let}(e_1; x . e_2) \mapsto \{e_1/x\}e_2 \\
\text{LET-STEP} & : \quad e_1 \mapsto e_1' \\
\end{align*}
\]

Both rules apply to a given let, which means that the dynamics is non-deterministic\(^1\): for a given expression \( e \) there can be several expressions \( e' \) such that \( e \mapsto e' \). Nevertheless, every expression \( e \) of observable answer type has at most one value \( v \) such that \( e \mapsto^* v \). To see this, observe that uses of LET-STEP may be eliminated in preference to LET-SUBST by replicating the transitions on \( e_1 \) wherever it may occur in \( e_2 \), without changing the outcome. Doing so removes the non-determinism, ensuring that all routes to a value from a given expression have the same result. Even if an expression has a value, by any of several routes, it may also lead to divergence. For example, the expression \( \text{let}(\text{fix}[\tau](x.x);x.z) \) has a divergent transition sequence in which Rule LET-STEP is chosen repeatedly, to the exclusion of LET-SUBST, which would immediately terminate with the value \( z \). Were the value of \( x \) required in the body of the let, divergence would be inevitable, but because it is not, it is merely possible.

The use of two transition rules for let may be explained as follows. Whenever a let is to be evaluated, simply “guess” whether or not the value of the bound expression will be required to determine the final outcome. If not, Rule LET-SUBST is applied, and the expression is never evaluated; otherwise Rule LET-STEP is applied, because it cannot hurt to share the effort of evaluating the bound expression among its usages, provided that there are any at all. Bear in mind that the decision of which rule to use cannot be made on whether \( x \) occurs free in the body of the let! For example, the variable \( x \) may occur in a conditional branch not taken in a terminating evaluation sequence. This is why guessing is required, rather than a simple test.

Among the transition sequences leading to a value, the shortest one defines the by-need evaluation. Thus, by definition, the by-need dynamics does the least amount of work among all possible sequences leading to a value, which is consistent with the term lazy evaluation. In a terminating by-need sequence the evaluation of a let either applies Rule LET-SUBST without ever stepping \( e_1 \), or else it applies Rule LET-STEP repeatedly until \( e_1 \) is a value, and then applies Rule LET-STEP by

\(^1\)A widespread barbarism for indeterminate.
substituting that value. It is never profitable to step \( e_1 \) less than fully before substituting, because either the value of \( e_1 \) is never needed in the first place, or it is needed at least once, in which case the remaining steps to a value must be taken. If it occurs more than once, these steps are repeated, and hence are not part of a by-need evaluation, or, if it occurs exactly once, it makes no difference whether those steps are taken before or after substitution.

Consequently, in a by-need evaluation a \texttt{let}-bound expression is either fully evaluated before substitution, or is never evaluated at all. Any other choice leads to a longer transition sequence to a value than necessary, and hence is not maximally lazy. This, then, justifies the use of memoization to implement the “guessing” used in a by-need dynamics. A \texttt{let}-bound variable is treated as write-at-most-once state that is initialized to the unevaluated expression, and updated with its value if ever it is needed, so that subsequent uses will only ever see its value, and never repeat the work.

As discussed in PFPL Chapter 36, the machinery of by-need evaluation may be used to catch certain forms of infinite loop using what are called black holes. Certain usages of \texttt{fix} are guaranteed to diverge, the simplest being \( \texttt{fix}[\tau](x . e) \), which unrolls to itself, but there are other examples, such as the function \( \texttt{fix}[\texttt{nat} \rightarrow \texttt{nat}](f . \texttt{ap}(f ; z)) \), which can be considered to “call itself too early” and must necessarily diverge. One sensible attitude towards such examples is that more subtle forms of divergence are always possible, so why distinguish these cases from those? Another attitude is to attempt to catch these cases at run-time, and to abort gracefully, rather than go into an infinite loop. This can be achieved by replacing the simple unrolling transition for general recursion with the following rules:

\[
\begin{align*}
\text{\textsc{fix-\texttt{val}}} & \quad e \triangleright \quad \text{\texttt{fix}}[\tau](x . e) \rightarrow \{\text{\texttt{fix}}[\tau](x . e)/x\}e \\
\text{\textsc{fix-\texttt{step}}} & \quad \{\bullet/x\}e \mapsto \{\bullet/x\}e' \\
\end{align*}
\]

The pseudo-expression \( \bullet \) is the aforementioned “black hole” from which no transition is possible.\(^2\) Thus, the role of Rule \textsc{fix-step} is to ensure that \( e \) is not immediately self-dependent (otherwise the black hole would stop evaluation with an error) before unrolling. The work done prior to unrolling is shared among all uses of the \texttt{fix}, reflecting another aspect of laziness that minimizes redundant effort.

References


\(^2\)To avoid ambiguity, strictly speaking there must be a distinct black hole for each \texttt{fix} expression, writing, say, \( \bullet_{e,e} \) for the black hole associated to \( \texttt{fix}[\tau](x . e) \).

\[3\]