# Smallstep Semantics for System PCF

Hengchu Zhang

December 18, 2016

**Abstract**

This project is focused on studying the 3 different smallstep semantics presented for system PCF in PFPL [1], and it explores some connections bewteen the two "lazy" semantics – Call-By-Name and Call-By-Need.

During the project, I have developed a Coq model of the language under Call-By-Value and Call-By-Need and proved the safety theorems of these semantics in Coq.

**Language**

$$
\begin{array}{rcl}
\langle e \rangle & ::= & \mathrm{x} \\
& | & \bullet \\
& | & a \\
& | & (e\ e) \\
& | & \lambda(x : \tau)\ .\ e \\
& | & z \\
& | & S\ e \\
& | & fix\ (x : \tau)\ .\ e
\end{array}
$$

$$
\begin{array}{rcl}
\langle \tau \rangle & ::= & \mathbb{N} \\
& | & \tau \to \tau
\end{array}
$$

Note that $a$ is a metavariable that stands for "symbols" into memoization tables, and $\bullet$ is a special construct used in Call-By-Need semantics. $\bullet$ represents the absence of a term in the memoization table to prevent and detect certain types of infinite reduction in Call-By-Need semantics.

**Formal Model in Coq**

In the beginning of the semester, we have studied the locally-nameless [2] representation as a formal foundation for defining syntactically correct programs of an object language in Coq. As an exercise to further my understanding of this representation, I used the same idea to define the syntax of System PCF in Coq.

The definition of language constructs in Coq is a direct translation of the grammar form given in the language section above.

The Call-By-Value definition of local closure is not surprising, with perhaps the definition for fixpoints being the only novel construct compared to what we have studied in class and here I give its definition in pseudo-Coq.

```
Inductive lc : exp -> Prop :=
| ...
| lc_fix : forall (L : atoms) t exp,
    (forall x, x `notin` L -> lc (open exp (exp_fvar x)))
    -> lc (exp_fix t exp).
```

We can see that it's in fact the same definition as the one given for abstractions because these two constructs capture variables in the same way.

Before we introduce the definition of local closure for Call-By-Need System PCF, we should first look at the definition of the smallstep semantics of Call-By-Need PCF. A step of reduction is no longer a binary relation between two terms of the langauge in Call-By-Need PCF. It involves evolutions in the memoization table, and intuitively the memoization table should only contain terms with valid syntax as well to avoid considering how invalid terms evaluate.

The smallstep relation in Call-By-Need is a relation from a triple of $(\Sigma, e, \mu)$ to $(\Sigma', e', \mu')$.

$\Sigma$ is a partial map from symbols $a$ to types $\tau$, and $\mu$ is a partial map from symbols $a$ to expressions $e$.

Intuitively, $\Sigma$'s map symbols to the expected types of the expressions these symbols refer to, and the $\mu$'s hold those expressions that are referred to by symbols in $e$. Those expressions in $\mu$ might also get stepped "in-place" by the Call-By-Need semantics.

Hence, we can see that we must include a definition of local closure for those expressions in the $\mu$ table in order to restrict our attention on well-formed expressions when studying the Call-By-Need semantics.

In pseudo-Coq notation, the definition of local closure for Call-By-Need is the following

```
Inductive lc : exp -> Prop :=
| lc_sym  : forall x, lc (exp_sym x)
| lc_blackhole : lc (exp_blackhole)
| ... (* The rest are the same as `lc` for Call-By-Value *).

Inductive lc_mu : mu_table -> Prop :=
| lc_mu_safe : forall mu, uniq mu
                     -> (forall x e, binds x e mu
                               -> lc e)
                     -> lc_mu mu.

Definition lc_by_need (e : exp) (mu : mu_table) : Prop :=
  lc e /\ lc_mu mu.
```

In other words, local closure requires both the expression $e$ and the table $\mu$ to be locally closed, and the $\mu$ table is defined to be locally closed if all the expressions stored in the table are locally closed. This allows us to precisely identify those valid terms that we can step over using the Call-By-Need smallstep semantics.

### Typing Under Call-By-Need

With the introduction of symbols and memoization tables in Call-By-Need System PCF, we also need to introduce new typing judgements for them.

The typing judgement for symbols simply performs a lookup in the $\Sigma$ table, and assigns this symbol that type.

$$\textsc{Symbol}\ \frac{a : \tau \in \Sigma}{\Gamma \vdash_\Sigma a : \tau}$$

For the memoization table $\mu$, we should consider it typeable only when all of its inhabitants match the types assigned in a corresponding table $\Sigma$.

$$\textsc{Memoization table}\ \frac{\forall\ a : \tau \in \Sigma, \mu(a) = e \neq \bullet \Rightarrow \vdash_{\Sigma'} e : \tau}{\vdash_{\Sigma'} \mu : \Sigma}$$

Note that we have treated $\bullet$ separately. This frees us from having to introduce a separate typing rule on the term level for $\bullet$'s, and it's OK to not consider typing for $\bullet$'s because they only appear while we step an expression "in-place" in the $\mu$ table, thus the type of the expression under inspection will be captured directly through term level typing.

### Call-By-Need Reduction

As we have briefly mentioned in the section on local closure, step relations in Call-By-Need System PCF are between triples of the form $(\Sigma, e, \mu)$. This is foreign compared to the usual smallstep semantics we have seen previously. Here, it is presented in informal inference rules as a reference for further discussion below.

Definition of values:

$$\textsc{Zero}\ \frac{}{z\ \mathrm{val}_{\Sigma}} \qquad \textsc{Succ}\ \frac{}{(s\ a)\ \mathrm{val}_{\Sigma, a:\mathbb{N}}} \qquad \textsc{Abstraction}\ \frac{}{\lambda(x:\tau)\ .\ e\ \mathrm{val}_{\Sigma}}$$

Definition of reduction rules:

$$\textsc{Lookup}\ \frac{e\ \mathrm{val}_{\Sigma, a:\tau}}{(\Sigma, a:\tau), a, (\mu, a:e) \to (\Sigma, a:\tau), e, (\mu, a:e)} \qquad \textsc{In-Place}\ \frac{(\Sigma, a:\tau), e, (\mu, a:\bullet) \to (\Sigma, a:\tau), e', (\mu, a:\bullet)}{(\Sigma, a:\tau), a, (\mu, a:e) \to (\Sigma, a:\tau), a, (\mu, a:e')}$$

$$\textsc{Succ}\ \frac{}{\Sigma, s(e), \mu \to (\Sigma, a:\mathbb{N}), s(a), (\mu, a:e)} \qquad \textsc{App}\ \frac{\Sigma, e_1, \mu \to \Sigma', e_1', \mu'}{\Sigma, e_1 e_2, \mu \to \Sigma', e_1' e_2, \mu'}$$

$$\textsc{Beta}\ \frac{}{\Sigma, ((\lambda(x:\tau)\ .\ e)e_2), \mu \to (\Sigma, a:\tau), [a/x]e, (\mu, a:e_2)}$$

$$\textsc{Fixpoint}\ \frac{}{\Sigma, fix(x:\tau)\ .\ e, \mu \to (\Sigma, a:fix(x:\tau)\ .\ e), [a/x]e, (\mu, a:fix(x:\tau)\ .\ e)}$$

### A Relation Between Call-By-Name And Call-By-Need

Even though System PCF under two different smallstep semantics are morally different languages, one would still hope that there is some connection between these semantics. Intuitively, if a well-typed System PCF terminates, then the other semantics should give the same "value" as the result of the execution. In this section, we aim to make this intuition precise.

First, let's consider a helpful function *apply*, whose definition is the following in pseudo-Coq.

*Definition.*
```
Fixpoint apply (mu : mu_table) (e : exp) : exp :=
  match e with
    | a                    => apply mu (lookup mu a)
    | exp_app e1 e2        => exp_app (apply mu e1) (apply mu e2)
    | exp_abs typ e_body   => exp_abs typ (apply mu e_body)
    | exp_zero             => exp_zero
    | exp_succ e_pred      => exp_succ (apply mu e_pred)
    | exp_fix typ e_fixbody => exp_fix typ (apply mu e_fixbody)
  end.
```

It simply recurses down the structure of the expression $e$, and fill-in the symbols by what they point to in the table $\mu$.

**Lemma 1.** *Apply and substitution commute*

$\forall\ e,\ t,\ x,\ \mu,$ *if* $e,\ t,\ \mu$ *are all locally closed:*

$$apply(\mu, [t/x]e) = [apply(\mu, t)/x]apply(\mu, e)$$

*Proof.* Proceed by induction on $e$.

- Variable $e = y$
- Zero $e = z$

  In both of these cases, *apply* is a no-op. The equation trivially holds.
- Symbol $e = a$.

  In this case, LHS = $apply(\mu, a)$, and RHS = $[apply(\mu, t)/x]apply(\mu, a)$.

  Now, since $\mu$ is locally closed, the term that $a$ refers to in $\mu$ must not contain any unbound free variables (such as $x$). Thus, the substitution is a no-op, and we have LHS = RHS = $apply(\mu, a)$.
- Application $e = e_1 e_2$.

$$\begin{aligned}
apply(\mu, [t/x]e) &= apply(\mu, [t/x]e_1 e_2) \\
&= apply(\mu, [t/x]e_1 [t/x]e_2) \\
&= apply(\mu, [t/x]e_1)\ apply(\mu, [t/x]e_2) && \text{By definition of } apply \\
&= [apply(\mu, t)/x]apply(\mu, e_1)\ [apply(\mu, t)/x]apply(\mu, e_2) && \text{By IH} \\
&= [apply(\mu, t)/x]apply(\mu, e_1 e_2) \\
&\text{By definition of substitution and } apply
\end{aligned}$$

- Abstraction $e = \lambda(y : \tau)\ .\ e_1$.

$$\begin{aligned}
apply(\mu, [t/x]e) &= apply(\mu, \lambda(y : \tau)\ .\ [t/x]e_1) \\
&= \lambda(y : \tau)\ .\ apply([t/x]e_1) && \text{By definition of } apply \\
&= \lambda(y : \tau)\ .\ [apply(\mu, t)/x]apply(\mu, e_1) && \text{By IH} \\
&= [apply(\mu, t)/x]apply(\mu, \lambda(y : \tau)\ .\ e_1) && \text{By definition of } apply
\end{aligned}$$

- Fixpoint, this case is exactly the same as abstraction.

□

*Definition.* Let $\rightarrow$ denote a Call-By-Name step relation, and $\rightarrow_\sigma$ denote a Call-By-Need step relation.

**Theorem 1.** *Call-By-Need simulates Call-By-Name.*

$\forall\ e, \vdash e : \tau,\ e \rightarrow e',$ *then* $\forall\ e_1,$ *if* $\vdash_\Sigma e_1 : \tau,\ \vdash_\Sigma \mu : \Sigma$ *and* $apply(\mu, e_1) = e,$ *then there exists a sequence of Call-By-Need evaluation steps* $(\Sigma, e_1, \mu) \rightarrow_\sigma^* (\Sigma', e_1', \mu')$ *such that* $apply(\mu', e_1') = e'$.

*Proof.* Proof by induction on the typing derivation.

- $e = z$, and $\tau = \mathbb{N}$
- $e = s(e_0)$, and $\tau = \mathbb{N}$
- $e = \lambda(x : \tau_L)\ .\ e_R$, and $\tau = \tau_L \rightarrow \tau_R$.

  In these three cases, $e$ is considered a value, and it doesn't step under Call-By-Name semantics.

- $e = e_L\ e_R$, and $\vdash e_L : \tau_R \to \tau$, $\vdash e_R : \tau_R$.

  We further case analyze the structure of $e \to e'$.

  - $e_L\ e_R \to e'_L\ e_R$ because of $e_L \to e'_L$.
    Thus, if we consider the possible structure of $e_1$, it is either a symbol, when supplied as argument to *apply* $\mu$ is $e_L\ e_R$, or an application of $e_{L_1}\ e_{R_1}$ whose LHS and RHS when supplied as arguments to *apply* $\mu$ will yield $e_L$ and $e_R$ respectively.

    In fact, we don't need to discuss the case when $e_1$ is immediately a symbol because the "Lookup" rule will allow us to use the first step to look up what the symbol refers to, and the following analysis is the same as the case when $e_1$ is an application. Thus, in the following developments, we'll omit discussing this case.

    By the typing judgements of Call-By-Need System PCF, we know that $e_{L_1}$ must have the same type as $e_L$, $\vdash_\Sigma e_{L_1} : \tau_R \to \tau$. And since $apply(\mu, e_{L_1}) = e_L$, the induction hypothesis applies, and we know $(\Sigma, e_{L_1}, \mu) \to^*_\sigma (\Sigma', e'_{L_1}, \mu')$ where $apply(\mu', e'_{L_1}) = e'_L$.

    Thus, by repeated application of the App rule of Call-By-Need semantics, we have

    $$(\Sigma, e_{L_1}\ e_{R_1}, \mu) \to^*_\sigma (\Sigma, e'_{L_1}\ e_{R_1}, \mu')$$

    where
    $apply(\mu, e'_{L_1}\ e_{R_1}) = e'_L\ e_R$.
  - $e_L = \lambda(x : \tau)\ .\ e_{Lbody}$, and $e_L\ e_R \to [e_R/x]e_{Lbody}$.

    Again, we can infer that $e_{L_1} = \lambda(x : \tau)\ .\ e'_{LBody}$, where
    $apply(\mu, e'_{LBody}) = e_{LBody}$, and $apply(\mu, e_{R_1}) = e_R$.

    Under the Call-By-Need semantics, we would step $e$ with

    $$(\Sigma, e, \mu) \to^*_\sigma (\Sigma', [a/x]e'_{LBody}, \mu')$$

    where $\mu'$ maps $a$ to $e_{R_1}$.

    Using the fact that apply and substitution commute, we have

    $$\begin{aligned} apply(\mu', [a/x]e'_{LBody}) &= [apply(\mu', a)/x]apply(\mu', e'_{LBody}) \\ &= [apply(\mu', e_{R_1})/x]e_{LBody} \\ &= [e_R/x]e_{LBody} \end{aligned}$$

  - $e = fix(x : \tau)\ .\ e_r$, and $e' = [fix(x : \tau)\ .\ e_r/x]e_r$.
    Then we know $e_1 = fix(x : \tau)\ .\ e_{r_1}$, where $apply(\mu, e_{r_1}) = e_r$.
    Considering the smallstep semantics for Call-By-Need, we know

    $$(\Sigma, fix(x : \tau)\ .\ e_{r_1}, \mu) \to_\sigma (\Sigma', [a/x]e_{r_1}, \mu')$$

    where $\mu'$ maps $a$ to $fix(x : \tau)\ .\ e_{r_1}$ and $\Sigma'$ maps $a$ to $\tau$.
    So we have the following equations

    $$\begin{aligned} apply(\mu', [a/x]e_{r_1}) &= [apply(\mu', a)/x]apply(\mu', e_{r_1}) \\ &= [fix(x : \tau)\ .\ apply(\mu', e_{r_1})/x]apply(\mu', e_{r_1}) \\ &= [fix(x : \tau)\ .\ e_r/x]e_r \\ &= e \end{aligned}$$

Thus, we have shown that each step of Call-By-Name reduction can be simulated using one or more steps of Call-By-Need reduction. □

**Theorem 2.** *Simulation*

*As a simple corrollary of the theorem above, we have the following result:* $\forall\ e, e \rightarrow^* v$, $v$ *is a value under Call-By-Name semantics, there exists* $\Sigma, \mu, t$, *such that*

$$(\emptyset, e, \emptyset) \rightarrow_\sigma^* (\Sigma, t, \mu)$$

*where* $apply(\mu, t) = v$.

*Proof.* By repeated application of Theorem 1.

### Caveats

During the project there were a few difficulties that I have not forseen, and in trying to deal with these difficulties I have incurred several limitations on the project which I will point out in this section.

- As readers might have noticed, the pseudo-Coq definition of *apply* will not be accepted by Coq for a few reasons:
  - $\mu$ might not contain the symbol $a$. In this report, all the proofs are carried out under the assumption that each $(\Sigma, e, \mu)$ triple are syntactically valid, meaning that they are locally closed, and there are no unbound symbols in the expression $e$. Hence, I think handling the case when $a$ is not bound in $\mu$ in the definition of *apply* would only complicate the problem without bringing interesting ideas.
  - A much more serious problem is that *apply* will not be accepted by Coq because it is not guaranteed to terminate.

    Consider a $\mu$ table that maps symbol $a$ to $a$ itself. *apply* would hopelessly keep trying to fill-in what $a$ points to but never get anywhere. Any other form of self-reference, direct or indirect, would cause this problem.

    Upon further observation, however, *apply* does not terminate precisely when the term would infinitely recurse under the judgements `loop` given in PFPL.

    I imagine that this issue could be resolved by selecting a more coarse relation between terms of Call-By-Name PCF and Call-By-Need PCF that looks beyond the syntactic structure. But I wasn't able to develop such a relation perhaps due to my lack of practice with logical relations. Hence, I have decided to elide this issue as well, and implicitly only analyze terms that would not `loop`.

- Careful readers might have also noticed that the step relation I have presented differs from the one in PFPL in the case of fixpoints.

  This is also related to the second issue of *apply* discussed above. *apply* would not terminate on any fixpoint had I chosen the definition given by PFPL, even if the fixpoint itself would terminate when evaluated!

  This is because *apply* does not consider the smallstep semantics when expanding symbols. Intuitively, a fixpoint that terminates eventually reaches a base case, perhaps with the help of an $if$ term, but *apply* would try to expand all occurences of symbols recursively in all branches of that $if$ term, resulting in an infinite recurse.

  One could perhaps improve the definition of *apply* and the analysis of how *apply* interacts with fixpoints to address this issue, but I have decided to change to definition of how fixpoints compute as a tradeoff for simplicity. The definition I have given is closer to how fixpoints evaluate under Call-By-Value and Call-By-Name semantics, which makes the proof of the relation between Call-By-Name and Call-By-Need evaluations much simpler.

- The proof presented in PFPL for progress required mutual induction. I experimented with a few approaches trying to replicate this proof in Coq, with the `Scheme` command and also tried developing depth-indexed derivations, and doing induction on sum of the depths of the two judgement that PFPL mutually induct on.

  In the case with `Scheme`, I wasn't able to generate the proper mutual induction principle that would help with proving progress. It seems that proving theorems by mutual induction in Coq might be a good technicality to for me explore further.

  With depth-indexed derivations, there was a few significant changes to the definition of typing judgements, which invalidated my previous proof script for a few critical lemmas that I have developed to help with proofs on perservation and progress. I wasn't able to completely re-develop all of those lemmas and prove the safety theorems under this new construction. Perhaps I could have engineered my proof structure to be more resilient to modifications, and this is also an area I would like to explore further in future formalization work.

  I will submit both a non-depth-indexed version of Coq script and a depth-indexed version as artifacts of this report, but latter includes a few proofs that don't quite go through.

# References


[1] R. Harper, *Practical Foundations For Programming Languages*. Cambridge University Press, 2012.

[2] A. Charguéraud, "The locally nameless representation," *Journal of Automated Reasoning*, 2011.