Strong Normalization for
Simply Typed Call-by-Push-Value

CIS 6700: Type Systems

Jonathan Chan

8 May 2025

1 Introduction

The goal of this project is to mechanize a proof of strong normalization for call-by-push-value (CBPV). It follows work by Forster, Schäfer, Spies, and Stark [2] on mechanizing the metatheory of CBPV in Rocq, but instead adapts the proof technique from POPLMark Reloaded [1].

Both proofs, and mine, follow the same sequence of steps: we define a logical relation that semantically interpret types as sets of terms, show that these sets are backward closed by reduction and that they contain only strongly normalizing terms, prove a fundamental soundness theorem that places well typed terms in the semantic interpretations of their types, and conclude that well typed terms must be strongly normalizing. The difference lies in *how strong normalization* (and reduction) is defined. The Rocq mechanization uses a traditional definition of strong normalization as an accessibility relation, while POPLMark Reloaded and this project use an inductive characterization of strong normalization, then prove it sound with respect to the traditional definition.

In contrast to both prior works, which use Rocq, Agda, or Beluga, this project is mechanized in Lean. A secondary purpose of this project is to assess the capabilities of Lean for PL metatheory, especially when making heavy use of mutually defined inductive types. I have implemented a mutual induction tactic for Lean[1], and the mutual values and computations of CBPV, combined with the even more complex mutual inductives from POPLMark Reloaded, make it an attractive test case to test the tactic's robustness.

This report is divided in roughly the same shape as the proof development.[2] Section 2 introduces the syntax and the typing rules for CBPV (`Syntax.lean`, `Typing.lean`). Then the inductive characterization of strong normalization is defined in Section 3 (`NormalInd.lean`), followed by the logical relation in Section 4 (`OpenSemantics.lean`). The central theorem of strong normalization is proven in Section 5 as a corollary of the fundamental theorem for the logical relation (`Soundness.lean`, `Normalization.lean`). To ensure that the inductive characterization is correct, Section 6 shows that it implies the traditional definition of strong normalization (`NormalAcc.lean`). This latter definition and proof depends on small-step reduction on open terms, whose properties are omitted here (`Reduction.lean`). In general, I gloss over details about substitution and reduction, since I'm interested in presenting the structure of the strong normalization proof and not the minutiae of syntax and binding. Finally, I discuss the merits of this proof technique and the challenges posed by using Lean in Section 7.

2 Syntax and Typing

The grammar of the CBPV for this project is given below. Terms and their types are divided between *values* and *computations*. Values are variables, unit, injections into a sum type, or a thunked computation, while computations are forcing thunks, functions, applications, returning values, binding returned values, and case analysis on sums. There are only value variables and no computation variables, since they represent terms

---

that require no more work to be done on them ready to be substituted in, and typing contexts similarly only contain value types. I use t to refer to terms that may be either values or computations.

Although the syntax is presented nominally, the mechanization uses an unscoped de Bruijn indexed representation of variables, along with simultaneous substitutions σ mapping variables to values, with $\cdot$ as the identity substitution. Applying a substitution is represented as $v[\sigma]$ and $m[\sigma]$, and implemented in terms of renamings, which are mappings from variables to other variables.

$$
\begin{aligned}
A &::= \mathsf{Unit} \mid A_1 + A_2 \mid \mathsf{U}\ B & \textit{value types} \\
B &::= A \to B \mid \mathsf{F}\ A & \textit{computation types} \\
v, w &::= x \mid \mathsf{unit} \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \mathsf{thunk}\ m & \textit{values} \\
m, n &::= \mathsf{force}\ v \mid \lambda x.\, m \mid m\ v \mid \mathsf{return}\ v & \textit{computations} \\
&\mid \mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n \\
&\mid \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} \\
\Gamma &::= \cdot \mid \Gamma, x : A & \textit{typing contexts} \\
\sigma &::= \cdot \mid \sigma, x \mapsto v & \textit{simultaneous substitutions}
\end{aligned}
$$

This is not the usual complete CBPV language, since it's missing both value tuples and computation tuples. I exclude them because they are syntactically not much more interesting than returns, whose eliminator is shaped like pattern matching on a singleton value tuple, and than thunks, whose eliminator is shaped like projection on a singleton computation tuple. In contrast, I do include sums, which represent the only way a computation can branch.

The typing rules below are standard for the values and computations included. The judgements for values and computations are defined mutually, just as are the types and the terms.

$$\boxed{\Gamma \vdash v : A} \hspace{6cm} \textit{value typing}$$

$$
\begin{array}{ccccc}
\text{T-\textsc{Var}} & & \text{T-\textsc{Inl}} & \text{T-\textsc{Inr}} & \text{T-\textsc{Thunk}} \\
\dfrac{x : A \in \Gamma}{\Gamma \vdash x : A} &
\text{T-\textsc{Unit}} \atop \dfrac{}{\Gamma \vdash \mathsf{unit} : \mathsf{Unit}} &
\dfrac{\Gamma \vdash v : A_1}{\Gamma \vdash \mathsf{inl}\ v : A_1 + A_2} &
\dfrac{\Gamma \vdash v : A_2}{\Gamma \vdash \mathsf{inr}\ v : A_1 + A_2} &
\dfrac{\Gamma \vdash m : B}{\Gamma \vdash \mathsf{thunk}\ m : \mathsf{U}\ B}
\end{array}
$$

$$\boxed{\Gamma \vdash m : B} \hspace{6cm} \textit{computation typing}$$

$$
\begin{array}{cccc}
\text{T-\textsc{Force}} & \text{T-\textsc{Lam}} & \text{T-\textsc{App}} & \text{T-\textsc{Ret}} \\
\dfrac{\Gamma \vdash v : \mathsf{U}\ B}{\Gamma \vdash \mathsf{force}\ v : B} &
\dfrac{\Gamma, x : A \vdash m : B}{\Gamma \vdash \lambda x.\, m : A \to B} &
\dfrac{\Gamma \vdash m : A \to B \quad \Gamma \vdash v : A}{\Gamma \vdash m\ v : B} &
\dfrac{\Gamma \vdash v : A}{\Gamma \vdash \mathsf{return}\ v : \mathsf{F}\ A}
\end{array}
$$

$$
\begin{array}{cc}
 & \text{T-\textsc{Case}} \\
\text{T-\textsc{Let}} & \dfrac{\begin{array}{l}\Gamma \vdash v : A_1 + A_2 \\ \Gamma, y : A_1 \vdash m : B \\ \Gamma, z : A_2 \vdash n : B\end{array}}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} : B} \\
\dfrac{\Gamma \vdash m : \mathsf{F}\ A \quad \Gamma, x : A \vdash n : B}{\Gamma \vdash \mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n : B} &
\end{array}
$$

3 Strong Normalization as an Inductive Definition

The idea behind the inductive characterization of strong normalization is to describe case by case when a term is strongly normal, rather than showing *a posteriori* which terms are strongly normal. This is done in conjunction with defining strongly neutral terms, which are blocked from β-reduction, and strong head reduction, which expands the set from normal forms to terms which must reduce to normal forms.

The general recipe for defining these for CBPV is as follows:

- The only strongly neutral values are variables.

- Strongly neutral computations are eliminators whose head positions are strongly neutral, while all other subterms are strongly normal.
- Strongly normal values are constructors whose subterms are strongly normal, or strongly neutral values, *i.e.* variables.
- Strongly normal computations are constructors whose subterms are strongly normal, or strongly neutral computations, or computations which reduce to strongly normal computations (backward closure).
- Strong head reduction consists of β reductions for all eliminators around the corresponding constructors, and congruent reductions in head positions.

Additionally, strong reduction requires premises asserting that the subterms that may "disappear" after reduction be strongly normal so that backward closure actually closes over strongly normal terms. These subterms are the values that get substituted into a computation, which may disappear if the computation never actually uses the binding, as well as computation branches not taken.

Because we're dealing with CBPV, values have no β reductions, so there's no need for head reduction of values as there are no heads. Furthermore, there is only one strongly neutral value, so we inline the definition as a variable where needed, but also write it as $v \in \mathsf{SNe}$ for symmetry as appropriate. Otherwise, the remaining four judgements are defined mutually below.

$\boxed{m \in \mathsf{SNe}}$ *strongly neutral computations*

$$\frac{}{\text{SNe-Force}} \qquad \frac{m \in \mathsf{SNe} \qquad v \in \mathsf{SN}}{\text{SNe-App}}$$

$$\frac{}{\mathsf{force}\ x \in \mathsf{SNe}} \qquad \frac{m \in \mathsf{SNe} \qquad v \in \mathsf{SN}}{m\ v \in \mathsf{SNe}} \qquad \frac{m \in \mathsf{SNe} \qquad n \in \mathsf{SN}}{\mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n \in \mathsf{SNe}}$$

SNe-Let

$$\frac{m \in \mathsf{SN} \qquad n \in \mathsf{SN}}{\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} \in \mathsf{SNe}}$$

SNe-Case

$\boxed{v \in \mathsf{SN}}$ *strongly normal values*

$$\frac{}{x \in \mathsf{SN}}\ \text{SN-Var} \qquad \frac{}{\mathsf{unit} \in \mathsf{SN}}\ \text{SN-Unit} \qquad \frac{v \in \mathsf{SN}}{\mathsf{inl}\ v \in \mathsf{SN}}\ \text{SN-Inl} \qquad \frac{v \in \mathsf{SN}}{\mathsf{inr}\ v \in \mathsf{SN}}\ \text{SN-Inr} \qquad \frac{m \in \mathsf{SN}}{\mathsf{thunk}\ m \in \mathsf{SN}}\ \text{SN-Thunk}$$

$\boxed{m \in \mathsf{SN}}$ *strongly normal computations*

$$\frac{m \in \mathsf{SN}}{\lambda x.\,m \in \mathsf{SN}}\ \text{SN-Lam} \qquad \frac{v \in \mathsf{SN}}{\mathsf{return}\ v \in \mathsf{SN}}\ \text{SN-Ret} \qquad \frac{m \in \mathsf{SNe}}{m \in \mathsf{SN}}\ \text{SN-SNe} \qquad \frac{m \rightsquigarrow n \qquad n \in \mathsf{SN}}{m \in \mathsf{SN}}\ \text{SN-Red}$$

$\boxed{m \rightsquigarrow n}$ *strong head reduction*

$$\frac{}{\mathsf{force}\ (\mathsf{thunk}\ m) \rightsquigarrow m}\ \text{SR-Thunk} \qquad \frac{v \in \mathsf{SN}}{(\lambda x.\,m)\ v \rightsquigarrow m[x \mapsto v]}\ \text{SR-Lam} \qquad \frac{v \in \mathsf{SN}}{\mathsf{let}\ x \leftarrow (\mathsf{return}\ v)\ \mathsf{in}\ m \rightsquigarrow m[x \mapsto v]}\ \text{SR-Ret}$$

$$\frac{v \in \mathsf{SN} \qquad n \in \mathsf{SN}}{\mathsf{case}\ (\mathsf{inl}\ v)\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} \rightsquigarrow m[y \mapsto v]}\ \text{SR-Inl}$$

$$\frac{v \in \mathsf{SN} \qquad m \in \mathsf{SN}}{\mathsf{case}\ (\mathsf{inr}\ v)\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} \rightsquigarrow n[z \mapsto v]}\ \text{SR-Inr}$$

$$\frac{m \rightsquigarrow n}{m\ v \rightsquigarrow n\ v}\ \text{SR-App} \qquad \frac{m \rightsquigarrow m'}{\mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n \rightsquigarrow \mathsf{let}\ x \leftarrow m'\ \mathsf{in}\ n}\ \text{SR-Let}$$

$$\boxed{m \rightsquigarrow^* n} \qquad\qquad\qquad \textit{reflexive, transitive closure of head reduction}$$

$$\frac{}{m \rightsquigarrow^* m} \text{\textsc{SRs-Refl}}
\qquad\qquad
\text{\textsc{SRs-Trans}} \atop \frac{m \rightsquigarrow m' \qquad m' \rightsquigarrow^* n}{m \rightsquigarrow^* n}$$

We also need the reflexive, transitive closure of head reduction, defined as a separate inductive above. Now we show a few simple lemmas about it, along with an inversion lemma for forcing (other inversion lemmas hold, but this is the only one we need). I present them below as judgement pseudorules using a dashed line to indicate that they are admissible rules. These are all proven either by construction or by induction on the first premise.

$$\text{\textsc{SRs-Once}} \atop \frac{m \rightsquigarrow n}{m \rightsquigarrow^* n}
\qquad
\text{\textsc{SRs-Trans'}} \atop \frac{m \rightsquigarrow^* m' \qquad m' \rightsquigarrow^* n}{m \rightsquigarrow^* n}
\qquad
\text{\textsc{SRs-App}} \atop \frac{m \rightsquigarrow^* n}{m\,v \rightsquigarrow^* n\,v}
\qquad
\text{\textsc{SRs-Let}} \atop \frac{m \rightsquigarrow^* m'}{\mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n \rightsquigarrow^* \mathsf{let}\ x \leftarrow m'\ \mathsf{in}\ n}$$

$$\text{\textsc{SN-Reds}} \atop \frac{m \rightsquigarrow^* n \qquad n \in \mathsf{SN}}{m \in \mathsf{SN}}
\qquad\qquad
\text{\textsc{SN-Force-inv}} \atop \frac{\mathsf{force}\ v \in \mathsf{SN}}{v \in \mathsf{SN}}$$

The most important property of strongly normal terms that we need is antirenaming, which states that undoing a renaming does not change the term's normalization or reduction behaviour. A crucial property that follows is extensionality of applications, which is an inversion lemma specifically when the application argument is a variable.

Lemma 1 (Antirenaming). *Let σ be a renaming, i.e. a substitution mapping variables to variables. Then the following hold:*

1. *If $m[\sigma] \in \mathsf{SNe}$ then $m \in \mathsf{SNe}$.*
2. *If $v[\sigma] \in \mathsf{SN}$ then $v \in \mathsf{SN}$.*
3. *If $m[\sigma] \in \mathsf{SN}$ then $m \in \mathsf{SN}$.*
4. *If $m[\sigma] \rightsquigarrow n$ then there exists some n' such that $n = n'[\sigma]$ and $m \rightsquigarrow n'$.*

Proof. By mutual induction on the four derivations of $m[\sigma] \in \mathsf{SNe}$, $v[\sigma] \in \mathsf{SN}$, $m[\sigma] \in \mathsf{SN}$, and $m[\sigma] \rightsquigarrow n$. $\square$

Corollary 2 (Extensionality). *If $m\,x \in \mathsf{SN}$ then $m \in \mathsf{SN}$.*

Proof. By induction on the derivation of strongly normal values. The possible cases are SN-SNe and SN-Red. Destructing the premise of the former gives $m \in \mathsf{SNe}$, and we conclude using rule SN-SNe again. In the latter case, we have $m\,x \rightsquigarrow n$ and $n \in \mathsf{SN}$; destructing on the reduction yields the possible cases rules SR-Lam and SR-App. In the first case, we have $m[x \mapsto v] \in \mathsf{SN}$, so the goal holds by Antirenaming. In the second case, we have $m \rightsquigarrow n$ and $n\,x \in \mathsf{SN}$, so by the induction hypothesis, we have $n \in \mathsf{SN}$, and we conclude using rule SN-Red. $\square$

4   Logical Relation on Open Terms

The next step is to define a logical relation that semantically interprets types as sets of open terms. The key property we need from these sets is that they contain only strongly normal terms. Because we are working with open terms to prove normalization and not just termination of evaluation of closed terms, we need to consider variables and strongly neutral terms. Having no other information about them other than their strong neutrality, we require that the interpretation sets always contain all strongly neutral terms.

The logical relation on simple types can be defined by induction on the structure of the type. However, I want to maximize the amount of mutual inductives used in this project, so we instead define the logical relation as an inductive binary relation between the type and the set of terms of its interpretation, denoted as $[\![A]\!] \searrow P$ and $[\![B]\!] \searrow P$ below. I use set builder notation to define the sets and set membership, but they

are implemented in the mechanization as functions that take terms and return propositions and as function applications. Here, the conjunctions ($\wedge$), disjunctions ($\vee$), equalities ($=$), implications ($\Rightarrow$), and universal ($\forall$) and existential ($\exists$) quantifiers are part of the metatheory, not part of the object language.

$$\boxed{[\![A]\!] \searrow P} \hspace{3cm} \textit{semantic interpretation of value types}$$

$$\frac{}{[\![\mathsf{Unit}]\!] \searrow \{v \mid v \in \mathsf{SNe} \vee v = \mathsf{unit}\}} \text{ LR-Unit} \hspace{2cm} \frac{[\![B]\!] \searrow P}{[\![\mathsf{U}\ B]\!] \searrow \{v \mid \mathsf{force}\ v \in P\}} \text{ LR-U}$$

$$\text{LR-Sum} \quad \frac{[\![A_1]\!] \searrow P \qquad [\![A_2]\!] \searrow Q}{[\![A_1 + A_2]\!] \searrow \{v \mid v \in \mathsf{SNe} \vee (\exists w.\ v = \mathsf{inl}\ w \wedge w \in P) \vee (\exists w.\ v = \mathsf{inr}\ w \wedge w \in Q)\}}$$

$$\boxed{[\![B]\!] \searrow P} \hspace{3cm} \textit{semantic interpretation of computation types}$$

$$\text{LR-F} \quad \frac{[\![A]\!] \searrow P}{[\![\mathsf{F}\ A]\!] \searrow \{m \mid (\exists n.\ m \rightsquigarrow^* n \wedge n \in \mathsf{SNe}) \vee (\exists v.\ m \rightsquigarrow^* \mathsf{return}\ v \wedge v \in P)\}}$$

$$\text{LR-Arr} \quad \frac{[\![A]\!] \searrow P \qquad [\![B]\!] \searrow Q}{[\![A \rightarrow B]\!] \searrow \{m \mid \forall v.\ v \in P \Rightarrow m\ v \in Q\}}$$

The terms in the interpretation of a type can be characterized in two different ways: by what constructor of that type they reduce to, or by how they act when eliminated by an eliminator for that type. For the former, we need to explicitly include the possibility of the term being strongly neutral. I chose the following characterizations because they seemed the simplest to me, but the alternate choices likely work as well.

- Values in the interpretation of the unit type are either variables or the unit value.
- Values are in the interpretation of the $\mathsf{U}\ B$ type if they can be forced to computations in the interpretation of the B type.
- Values in the interpretation of the sum type $A_1 + A_2$ are either variables, left injections whose values are in the interpretation of A_1, or right injections whose values are in that of A_2.
- Computations in the interpretation of the $\mathsf{F}\ A$ type reduce to either a neutral computation or a return whose value is in the interpretation of A.
- Computations are in the interpretation of the function type $A \rightarrow B$ if applying them to values in the interpretation of A yields computations in the interpretation of B.

By this description, it sounds like the logical relation can be presented directly as a relation between a type and a term; this presentation is given in Figure 1. Unfortunately, this is not well defined, since the logical relation appears in a negative position (to the left of an implication) in the premise of rule LR'-Arr. The alternate presentation can be interpreted as a function match on the term and the type and returning the conjunction of its premises, but I wanted to use a mutually defined inductive definition.

$$\boxed{v \in [\![A]\!]} \hspace{0.5cm} \boxed{m \in [\![B]\!]} \hspace{3cm} \textit{semantic inhabitance of types}$$

$$\frac{}{x \in [\![\mathsf{Unit}]\!]} \text{ LR'-Unit-var} \quad \frac{}{\mathsf{unit} \in [\![\mathsf{Unit}]\!]} \text{ LR'-Unit-unit} \quad \frac{}{x \in [\![A_1 + A_2]\!]} \text{ LR'-Sum-var} \quad \frac{v \in [\![A_1]\!]}{\mathsf{inl}\ v \in [\![A_1 + A_2]\!]} \text{ LR'-Sum-inl} \quad \frac{v \in [\![A_2]\!]}{\mathsf{inr}\ v \in [\![A_1 + A_2]\!]} \text{ LR'-Sum-inr}$$

$$\frac{\mathsf{force}\ v \in [\![B]\!]}{v \in [\![\mathsf{U}\ B]\!]} \text{ LR'-Force} \quad \frac{m \rightsquigarrow^* n \qquad n \in \mathsf{SNe}}{m \in [\![\mathsf{F}\ A]\!]} \text{ LR'-F-var} \quad \frac{m \rightsquigarrow^* \mathsf{return}\ v \qquad v \in [\![A]\!]}{m \in [\![\mathsf{F}\ A]\!]} \text{ LR'-F-ret} \quad \frac{\forall v.\ v \in [\![A]\!] \Rightarrow m\ v \in [\![B]\!]}{m \in [\![A \rightarrow B]\!]} \text{ LR'-Arr}$$

Figure 1: Alternate presentation of the logical relation

Using the inductive presentation of the logical relation, there are three easy properties to show: interpretability, which states that all types have an interpretation; determinism, which states that the interpretation indeed behaves like a function from types to sets of terms; and backward closure, which states that the interpretations of computation types are backward closed under multi-step head reduction. The last property is why rule LR'-F-VAR needs to include computations that *reduce to* strongly neutral terms or returns, not merely ones that are such terms.

Lemma 3 (Interpretability).

1. *Given A, there exists P such that $[\![A]\!] \searrow P$.*
2. *Given B, there exists P such that $[\![B]\!] \searrow P$.*

Proof. By mutual induction on the types A and B. $\qquad\square$

Lemma 4 (Determinism).

1. *If $[\![A]\!] \searrow P$ and $[\![A]\!] \searrow Q$ then $P = Q$.*
2. *If $[\![B]\!] \searrow P$ and $[\![B]\!] \searrow Q$ then $P = Q$.*

Proof. By mutual induction on the first derivations of the logical relation, then by cases on the second derivations. $\qquad\square$

Lemma 5 (Backward closure). *Given $[\![B]\!] \searrow P$ and $m \rightsquigarrow^* n$, if $n \in P$ then $m \in P$.*

Proof. By induction on the derivation of the logical relation. $\qquad\square$

The final key property is adequacy, which states that the interpretations of types must contain all strongly neutral terms and must contain only strongly normal terms. Such sets are said to be *reducibility candidates*.

Definition 1 (Reducibility candidates). A reducibility candidate is a set of terms P where, given a term t, if $t \in \mathsf{SNe}$ then $t \in P$, and if $t \in P$ then $t \in \mathsf{SN}$.

Lemma 6 (Adequacy).

1. *If $[\![A]\!] \searrow P$ then P is a reducibility candidate.*
2. *If $[\![B]\!] \searrow P$ then P is a reducibility candidate.*

Proof. By mutual induction on the derivations of the logical relation. Rule SN-FORCE-INV is used in the S-U case, while rule SN-REDS is used in the rule S-F case. In the S-ARR case on $A \rightarrow B$, where $[\![A]\!] \searrow P$ and $[\![B]\!] \searrow Q$, to show the second part of its interpretation being a reducibility candidate, we are given m such that for every $v \in P$, $m\,v \in Q$, and the goal is to show that $m \in \mathsf{SN}$. By the induction hypotheses, picking an arbitrary variable x, we have that $x \in P$ (since it is neutral) and that $m\,x \in \mathsf{SN}$. Then the goal holds by Extensionality. $\qquad\square$

5   Semantic Typing and the Fundamental Theorem

Now that we know that the interpretations contain only strongly normal terms, our goal is to show that well typed terms inhabit the interpretations of their types. We first define what it means for a substitution to be semantically well formed with respect to a context.

Definition 2 (Semantic well-formedness of substitutions). A substitution σ is well formed with respect to a context Γ, written $\Gamma \vDash \sigma$, if for every $x : A \in \Gamma$ and $[\![A]\!] \searrow P$, we have $x[\sigma] \in P$. In short, the substitution maps variables in the context to values in the interpretations of their types.

These judgements can be built up inductively, as demonstrated by the below admissible rules, which are proven by cases.

$\boxed{\Gamma \vDash \sigma}$ *admissible semantic substitution well-formedness*

$$
\text{S-NIL} \quad \frac{-\ -}{\Gamma \vDash \cdot}
\qquad
\text{S-CONS} \quad \frac{\Gamma \vDash \sigma \qquad \llbracket A \rrbracket \searrow P \qquad v \in P}{\Gamma, x : A \vDash \sigma, x \mapsto v}
$$

Then we can define semantic typing in terms of the logical relation, using semantic well formedness of substitutions to handle the context.

Definition 3 (Semantic typing)**.**

1. v semantically has type A under context Γ, written $\Gamma \vDash v : A$, if for every σ such that $\Gamma \vDash \sigma$, there exists an interpretation $\llbracket A \rrbracket \searrow P$ such that $v[\sigma] \in P$.
2. m semantically has type B under context Γ, written $\Gamma \vDash m : B$, if for every σ such that $\Gamma \vDash \sigma$, there exists an interpretation $\llbracket B \rrbracket \searrow P$ such that $m[\sigma] \in P$.

Semantic typing follows exactly the same shape of rules as syntactic typing, so I present them here as admissible rules. All the hard work happens in these lemmas; the fundamental theorem of soundness of syntactic typing with respect to semantic typing then follows directly. Normalization holds as a corollary.

$\boxed{\Gamma \vDash v : A}$ *admissible semantic value typing*

$$
\text{S-VAR} \quad \frac{x : A \in \Gamma}{\Gamma \vDash x : A}
\quad
\text{S-UNIT} \quad \frac{}{\Gamma \vDash \mathsf{unit} : \mathsf{Unit}}
\quad
\text{S-INL} \quad \frac{\Gamma \vDash v : A_1}{\Gamma \vDash \mathsf{inl}\ v : A_1 + A_2}
\quad
\text{S-INR} \quad \frac{\Gamma \vDash v : A_2}{\Gamma \vDash \mathsf{inr}\ v : A_1 + A_2}
\quad
\text{S-THUNK} \quad \frac{\Gamma \vDash m : B}{\Gamma \vDash \mathsf{thunk}\ m : \mathsf{U}\ B}
$$

$\boxed{\Gamma \vDash m : B}$ *admissible semantic computation typing*

$$
\text{S-FORCE} \quad \frac{\Gamma \vDash v : \mathsf{U}\ B}{\Gamma \vDash \mathsf{force}\ v : B}
\quad
\text{S-ARR} \quad \frac{\Gamma, x : A \vDash m : B}{\Gamma \vDash \lambda x.\, m : A \to B}
\quad
\text{S-APP} \quad \frac{\Gamma \vDash m : A \to B \qquad \Gamma \vDash v : A}{\Gamma \vDash m\ v : B}
\quad
\text{S-RET} \quad \frac{\Gamma \vDash v : A}{\Gamma \vDash \mathsf{return}\ v : \mathsf{F}\ A}
$$

$$
\text{S-LET} \quad \frac{\Gamma \vDash m : \mathsf{F}\ A \qquad \Gamma, x : A \vDash n : B}{\Gamma \vDash \mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n : B}
\qquad
\text{S-CASE} \quad \frac{\Gamma \vDash v : A_1 + A_2 \qquad \Gamma, y : A_1 \vDash m : B \qquad \Gamma, z : A_2 \vDash n : B}{\Gamma \vDash \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} : B}
$$

Proof. By construction using prior lemmas, in particular rule SRS-LET in case S-LET; Interpretability in cases S-VAR, S-INL, S-INR, and S-LAM; Determinism in cases S-LAM and S-APP; Backward closure in cases S-THUNK, S-LAM, S-LET, and S-CASE; and Adequacy in cases S-LAM, S-LET, and S-CASE. □

Theorem 7 (Soundness)**.**

1. *If* $\Gamma \vdash v : A$ *then* $\Gamma \vDash v : A$.
2. *If* $\Gamma \vdash m : B$ *then* $\Gamma \vDash m : B$.

Proof. By mutual induction on the syntactic typing derivations, using the admissible semantic typing rules. □

Corollary 8 (Normalization)**.**

1. *If* $\Gamma \vdash v : A$ *then* $v \in \mathsf{SN}$.
2. *If* $\Gamma \vdash m : B$ *then* $m \in \mathsf{SN}$.

Proof. By Soundness, using the identity substitution and rule S-NIL, the well typed terms inhabit the semantic interpretations of their types. Then by Adequacy, they are also strongly normalizing. □

6 Strong Normalization as an Accessibility Relation

How do we know that our definition of strongly normal terms is correct? What are the properties that define its correctness? It seems to be insufficient to talk about strongly neutral and normal terms alone.

For instance, we could add another rule asserting that $m \in \mathsf{SN}$ if $m \rightsquigarrow m$, which doesn't appear to violate any of the existing properties we require. Then $(\lambda x.\,(\mathsf{force}\ x)\ x)\ (\mathsf{thunk}\ (\lambda x.\,(\mathsf{force}\ x)\ x))$, which reduces to itself, would be considered a "strongly normal" term, and typing rules that can type this term (perhaps with a recursive type) would still be sound with respect to our logical relation.

To try to rule out this kind of mistake in the definition, we might want to prove that strongly normal terms never loop, *i.e.* $m \in \mathsf{SN} \wedge m \rightsquigarrow^* m$ is impossible. However, this does not cover diverging terms that grow forever. Furthermore, $m \rightsquigarrow n$ isn't really a full reduction relation on its own, since it's defined mutually with strongly normal terms, and it doesn't describe reducing anywhere other than in the head position.

The solution we take from POPLMark Reloaded is to prove that SN is sound with respect to a traditional presentation of strongly normal terms based on full small-step reduction $\boxed{v \rightsquigarrow w}$, $\boxed{m \rightsquigarrow n}$ of values and computations. I omit here the definitions of these reductions, but they encompass all β-reduction rules and are congruent over all subterms, including under binders. Traditional strongly normal terms, written $v \in \mathsf{sn}$, $m \in \mathsf{sn}$, are accessibility relations with respect to small-step reduction. Terms are inductively defined as strongly normal if they step to strongly normal terms, so terms which don't step are trivially normal. This rules out looping and diverging terms, since they never stop stepping.

$\boxed{v \in \mathsf{sn}}$ $\boxed{m \in \mathsf{sn}}$ $\hfill$ *strongly normalizing terms*

$$\frac{\text{SN-VAL}}{\forall w.\, v \rightsquigarrow w \Rightarrow w \in \mathsf{sn}} \qquad\qquad \frac{\text{SN-COM}}{\forall n.\, m \rightsquigarrow n \Rightarrow n \in \mathsf{sn}}$$
$$\frac{}{v \in \mathsf{sn}} \qquad\qquad\qquad \frac{}{m \in \mathsf{sn}}$$

The inversion lemmas we need are easily proven by induction.

$$\frac{\text{SN-APP-INV1}}{m\ v \in \mathsf{sn}} \qquad \frac{\text{SN-APP-INV2}}{m\ v \in \mathsf{sn}} \qquad \frac{\text{SN-LET-INV1}}{\mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n \in \mathsf{sn}} \qquad \frac{\text{SN-LET-INV2}}{\mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n \in \mathsf{sn}}$$
$$\frac{}{m \in \mathsf{sn}} \qquad\quad \frac{}{v \in \mathsf{sn}} \qquad\quad \frac{}{m \in \mathsf{sn}} \qquad\qquad\qquad \frac{}{n \in \mathsf{sn}}$$

Proof. By induction on the premises of strongly normal terms, using congruence of single-step reduction. $\square$

In contrast, while congruence rules for SN hold by definition, they require some work to prove for sn. The strategy is to mirror the inductive characterization, and define corresponding notions of neutral terms and head reduction, with strongly neutral terms being those both neutral and strongly normal. As before, variables are the only neutral value, but we write $v \in \mathsf{ne}$ to say that v is a variable for symmetry.

$\boxed{m \rightsquigarrow_{\mathsf{sn}} n}$ $\hfill$ *strong head reduction*

$$\frac{\text{SR-THUNK}}{\mathsf{force}\ (\mathsf{thunk}\ m) \rightsquigarrow_{\mathsf{sn}} m} \qquad \frac{\text{SR-LAM} \quad v \in \mathsf{sn}}{(\lambda x.\,m)\ v \rightsquigarrow_{\mathsf{sn}} m[x \mapsto v]} \qquad \frac{\text{SR-RET} \quad v \in \mathsf{sn}}{\mathsf{let}\ x \leftarrow (\mathsf{return}\ v)\ \mathsf{in}\ m \rightsquigarrow_{\mathsf{sn}} m[x \mapsto v]}$$

$$\frac{\text{SR-INL} \qquad v \in \mathsf{sn} \quad n \in \mathsf{sn}}{\mathsf{case}\ (\mathsf{inl}\ v)\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} \rightsquigarrow_{\mathsf{sn}} m[y \mapsto v]} \qquad \frac{\text{SR-INR} \qquad v \in \mathsf{sn} \quad m \in \mathsf{sn}}{\mathsf{case}\ (\mathsf{inr}\ v)\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} \rightsquigarrow_{\mathsf{sn}} n[z \mapsto v]}$$

$$\frac{\text{SR-APP} \quad m \rightsquigarrow_{\mathsf{sn}} n}{m\ v \rightsquigarrow_{\mathsf{sn}} n\ v} \qquad\qquad \frac{\text{SR-LET} \quad m \rightsquigarrow_{\mathsf{sn}} m'}{\mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n \rightsquigarrow_{\mathsf{sn}} \mathsf{let}\ x \leftarrow m'\ \mathsf{in}\ n}$$

$\boxed{m \in \mathsf{ne}}$

$$
\frac{}{\text{force } x \in \mathsf{ne}} \text{ NE-Force}
\qquad
\frac{m \in \mathsf{ne}}{m\ v \in \mathsf{ne}} \text{ NE-App}
\qquad
\frac{m \in \mathsf{ne}}{\text{let } x \leftarrow m \text{ in } n \in \mathsf{ne}} \text{ NE-Let}
\qquad
\frac{}{\text{case } x \text{ of } \{\text{inl } y \Rightarrow m \ ; \text{ inr } z \Rightarrow n\} \in \mathsf{ne}} \text{ NE-Case}
$$

Definition 4 (Strongly neutral computations)**.** A strongly neutral computation, written $m \in \mathsf{sne}$, is a computation that is both neutral and strongly normalizing, *i.e.* $m \in \mathsf{ne}$ and $m \in \mathsf{sn}$.

The admissible congruence rules for sn mirror exactly the congruence constructors for SNe and SN, replacing these by sne and sn in each premise. Proving them additionally requires showing that small-step reduction preserves neutrality.

Lemma 9 (Preservation)**.**

1. *If $v \rightsquigarrow w$ and $v \in \mathsf{ne}$ then $w \in \mathsf{ne}$.*
2. *If $m \rightsquigarrow n$ and $m \in \mathsf{ne}$ then $n \in \mathsf{ne}$.*

Proof. By mutual induction on the single-step reductions. $\qquad\square$

$\boxed{v \in \mathsf{sn}}$ *admissible strongly normal values*

$$
\frac{-\ -\ -}{x \in \mathsf{sn}} \text{ SN-Var}
\qquad
\frac{-\ -\ -\ -\ -}{\text{unit} \in \mathsf{sn}} \text{ SN-Unit}
\qquad
\frac{v \in \mathsf{sn}}{\text{inl } v \in \mathsf{sn}} \text{ SN-Inl}
\qquad
\frac{v \in \mathsf{sn}}{\text{inr } v \in \mathsf{sn}} \text{ SN-Inr}
\qquad
\frac{m \in \mathsf{sn}}{\text{thunk } m \in \mathsf{sn}} \text{ SN-Thunk}
$$

$\boxed{m \in \mathsf{sn}}$ *admissible strongly normal computations*

$$
\frac{m \in \mathsf{sn}}{\lambda x.\, m \in \mathsf{sn}} \text{ SN-Lam}
\qquad
\frac{v \in \mathsf{sn}}{\text{return } v \in \mathsf{sn}} \text{ SN-Ret}
\qquad
\frac{v \in \mathsf{sn}}{\text{force } v \in \mathsf{sn}} \text{ SN-Force}
\qquad
\frac{m \in \mathsf{sne} \qquad v \in \mathsf{sn}}{m\ v \in \mathsf{sn}} \text{ SN-App}
\qquad
\frac{m \in \mathsf{sne} \qquad n \in \mathsf{sn}}{\text{let } x \leftarrow m \text{ in } n \in \mathsf{sn}} \text{ SN-Let}
$$

$$
\frac{v \in \mathsf{sne} \qquad m \in \mathsf{sn} \qquad n \in \mathsf{sn}}{\text{case } v \text{ of } \{\text{inl } y \Rightarrow m \ ; \text{ inr } z \Rightarrow n\} \in \mathsf{sn}} \text{ SN-Case}
$$

Proof. Rules SN-App, SN-Let, and SN-Case are proven by double induction on the two derivations of strongly normal terms. Intuitively, we want to show that if the conclusion steps, then it steps to a strongly normal term, knowing by the induction hypotheses that if their subterms reduce, then they also reduce to strongly normal terms. Neutrality of the term in head position eliminates cases where the conclusion β-reduces, leaving only the congruent reductions. Because single-step reduction only steps in one subterm, we only need the induction hypothesis for that reducing subterm, so the double induction is really a lexicographic induction on the two derivations. We additionally require Preservation to carry along neutrality when the heads reduce in cases SN-App and SN-Let. All other cases are direct by construction or by induction on their sole premise. $\qquad\square$

The only missing correponding rule is that for rule SN-Red, which backward closes strongly normal terms under head reduction. Proving this for sn requires much more work and many more intermediate lemmas. To show backward closure under β-reduction, or *head expansion*, we first need antisubstitution to be able to undo substitutions.

Lemma 10 (Antisubstitution (sn))**.** *If $m[x \mapsto v] \in \mathsf{sn}$ and $v \in \mathsf{sn}$ then $m \in \mathsf{sn}$.*

Proof. By induction on the derivation of $m[x \mapsto v] \in \mathsf{sn}$. $\qquad\square$

$\boxed{m \in \mathsf{sn}}$ *head expansion*

$$\frac{m \in \mathsf{sn}}{\mathsf{force}\ (\mathsf{thunk}\ m) \in \mathsf{sn}}\ \text{\small SN-Force-Thunk}$$

$$\frac{v \in \mathsf{sn} \quad m[x \mapsto v] \in \mathsf{sn}}{(\lambda x.\ m)\ v \in \mathsf{sn}}\ \text{\small SN-App-Lam}$$

$$\frac{v \in \mathsf{sn} \quad m[x \mapsto v] \in \mathsf{sn}}{\mathsf{let}\ x \leftarrow (\mathsf{return}\ v)\ \mathsf{in}\ m \in \mathsf{sn}}\ \text{\small SN-Let-Ret}$$

$$\frac{v \in \mathsf{sn} \quad m[y \mapsto v] \in \mathsf{sn} \quad n \in \mathsf{sn}}{\mathsf{case}\ (\mathsf{inl}\ v)\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} \in \mathsf{sn}}\ \text{\small SN-Case-Inl}$$

$$\frac{v \in \mathsf{sn} \quad m \in \mathsf{sn} \quad n[z \mapsto v] \in \mathsf{sn}}{\mathsf{case}\ (\mathsf{inr}\ v)\ \mathsf{of}\ \{\mathsf{inl}\ y \Rightarrow m\ ;\ \mathsf{inr}\ z \Rightarrow n\} \in \mathsf{sn}}\ \text{\small SN-Case-Inr}$$

Proof. Rule SN-Force-Thunk holds directly by induction on the premise; the remaining proofs are more complex. First, given the premise $m[x \mapsto v] \in \mathsf{sn}$ (or $n[z \mapsto v]$), use Antisubstitution (sn) to obtain $m \in \mathsf{sn}$. Then proceed by double (or triple) induction on the derivations of $v \in \mathsf{sn}$ and $m \in \mathsf{sn}$ (and $n \in \mathsf{sn}$ for the case rules). Similarly to the admissible strongly normal rules, these are lexicographic inductions, except $m[x \mapsto v] \in \mathsf{sn}$ (or $n[z \mapsto v]$) is used to satisfy the β-reduction cases. □

Now remains backward closure under congruent reduction in head position. Because showing that a term is strongly normal involves single-step reduction, we need to ensure that head reduction and single-step reduction commute and behave nicely with each other. Proving commutativity is a lengthy proof that involves careful analysis of the structure of both reductions.

Lemma 11 (Commutativity). *If $m \rightsquigarrow n_1$ and $m \rightsquigarrow_{\mathsf{sn}} n_2$, then either $n_1 = n_2$, or there exists some m' such that $n_1 \rightsquigarrow_{\mathsf{sn}} m'$ and $n_2 \rightsquigarrow m'$.*

Proof. By induction on the derivation of $m \rightsquigarrow_{\mathsf{sn}} n_2$, then by cases on the derivation of $m \rightsquigarrow n_1$. □

$\boxed{m \in \mathsf{sn}}$ *backward closure in head position*

$$\frac{m \rightsquigarrow_{\mathsf{sn}} n \quad n\ v \in \mathsf{sn}}{m\ v \in \mathsf{sn}}\ \text{\small SN-App-bwd}$$

$$\frac{m \rightsquigarrow_{\mathsf{sn}} m' \quad \mathsf{let}\ x \leftarrow m'\ \mathsf{in}\ n \in \mathsf{sn}}{\mathsf{let}\ x \leftarrow m\ \mathsf{in}\ n \in \mathsf{sn}}\ \text{\small SN-Let-bwd}$$

Proof. First, use rules SN-App-inv1, SN-App-inv2, SN-Let-inv1, and SN-Let-inv2 to obtain $m \in \mathsf{sn}$ and $v \in \mathsf{sn}/n \in \mathsf{sn}$. Then proceed by double induction on the derivations of $m \in \mathsf{sn}$ and $v \in \mathsf{sn}/n \in \mathsf{sn}$, again as lexicographic induction. We want to show that if the conclusion steps, then it steps to a strongly normal term. Strong reduction of the head position eliminates the cases of β-reduction, leaving the cases where the head position steps or the other position steps. If the head position steps, we use Commutativity to join the strong reduction and the single-step reduction together, then use the first induction hypothesis. Otherwise, we use the second induction hypothesis. We need the last premise to step through either of the subterms, since we have no congruence rule for when the head position is not neutral. □

All of these lemmas at last give us backward closure.

Lemma 12 (Backward closure). *If $m \rightsquigarrow_{\mathsf{sn}} n$ and $n \in \mathsf{sn}$ then $m \in \mathsf{sn}$.*

Proof. By induction on the derivation of $m \rightsquigarrow_{\mathsf{sn}} n$. The cases correspond exactly to each of rules SN-Force-Thunk, SN-App-Lam, SN-Let-Ret, SN-Case-Inl, SN-Case-Inr, SN-App-bwd, and SN-Let-bwd. □

This gives us all the pieces to show that the inductive characterization is sound with respect to the traditional presentation of strongly normal terms, using neutral terms and head reduction as intermediate widgets to strengthen the induction hypotheses, although we can show soundness for neutrality independently. With soundness of strongly normal terms combined with soundness of syntactic typing, traditional strong normalization of well typed terms holds as a corollary.

Lemma 13 (Soundness (SNe)). *If $m \in \mathsf{SNe}$ then $m \in \mathsf{ne}$.*

Proof. By induction on the derivation of $m \in \mathsf{SNe}$. □

Theorem 14 (Soundness (SN)).

1. *If $m \in \mathsf{SNe}$ then $m \in \mathsf{sn}$.*
2. *If $v \in \mathsf{SN}$ then $v \in \mathsf{sn}$.*
3. *If $m \in \mathsf{SN}$ then $m \in \mathsf{sn}$.*
4. *If $m \rightsquigarrow n$ then $m \rightsquigarrow_{\mathsf{sn}} n$.*

Proof. By mutual induction on the derivations of $m \in \mathsf{SNe}$, $v \in \mathsf{SN}$, $m \in \mathsf{SN}$, and $m \rightsquigarrow n$. The cases for the first three correspond to the admissible strongly normal rules, using Soundness (SNe) as needed, except for the SN-RED case, which uses Backward closure. The cases for strong reduction hold by construction. □

Corollary 15. *If $\Gamma \vdash v : A$ then $v \in \mathsf{sn}$, and if $\Gamma \vdash m : B$ then $m \in \mathsf{sn}$.*

7 Discussion

7.1 Proof Structure of Strong Normalization

Overall, the inductive characterization of strongly normal terms is convenient to work with. Its congruence and inversion properties are exactly what are needed to show Backward closure and Adequacy of the logical relation, which are already built in by its inductive nature. Proving soundness itself relies mostly on the properties of the logical relation, along with a few more congruence properties of strong head reduction. Knowing the general recipe for constructing the inductive definition and for the structure of a proof by logical relations makes it easy to adapt the proof of strong normalization to simply typed CBPV. There are only a few pitfalls related to simply typed CBPV specifically:

- I originally included a corresponding notion of reduction for values in the mutual inductive which reduced under subterms. This is unnecessary not only because values have no β-reductions, but also because subterms of strongly normal values need to be strongly normal anyway, so there is no need to reduce subterms. Having reduction for values therefore probably characterizes the same set of strongly normal terms, but made proofs unnecessarily difficult to complete.
- Similarly, I originally included premises asserting other subterms of head reduction must be strongly normal even if they appear in the reduced term on the right-hand side. This changes what terms may reduce, but not what is included in the set of strongly normal terms, since rule SN-RED requires that the right-hand term is strongly normal anyway. The soundness proof still went through, but this design is hard to justify by first principles, and adds proof burden to the lemmas for the traditional presentation of strongly normal terms later on.
- The lack of a notion of reduction for values creates an asymmetry that initially led me to omit the fact that the computations in the interpretation of $\mathsf{F}\, A$ in rule LR-F must *reduce* to strongly neutral terms instead of merely *being* strongly neutral. This makes Backward closure impossible to prove, since strongly neutral terms are not backward closed.

The inductive characterization is easy to work with because all the hard work seems to lie in showing it sound with respect to the traditional presentation of strongly normal terms. This proof took about as much time as the entire rest of the development. Not only does it require setting up an entire single-step reduction relation and proving all of its tedious congruence, renaming, substitution, and commutativity lemmas, the intermediate lemmas for Soundness (SN) are long and tricky. In particular, the congruence, head expansion, and backward closure lemmas require double or even triple induction. Rules SN-APP-BWD and SN-LET-BWD are especially tricky, since their double inductions are on two auxiliary derivations produced from one of the actual premises, and that premise unintuitively needs to stick around over the course of the inductions. Without guidance from the POPLMark Reloaded paper, I would have been stuck on these two particular lemmas for much longer.

The purpose of all the work done in Section 6 is to show that `sn` satisfies properties that otherwise hold by definition for `SN` as a constructor. If proving that the latter is sound with respect to the former already requires all this work showing that the former behaves exactly like the latter, then what is the point of using the inductive characterization? All constructions of `SN` could be directly replaced by the congruence and backward closure lemmas we have proven for `sn`.

Using the inductive characterization is beneficial only if we don't care about its soundness with respect to the traditional presentation, where strong normalization is not the end goal. For example, in the metatheory of dependent type theory, we care about normalization because we want to know that the type theory is consistent, and that definitional equality is decidable so that a type checker can actually be implemented. For the latter purpose, all we require is *weak normalization*: that there exists *some* reduction strategy that reduces terms to normal form. The shape of the inductive characterization makes it easy to show that a leftmost-outermost reduction strategy does so (`LeftmostOutermost.lean`). While weak normalization can be proven directly using the logical relation, generalizing to the inductive strong normalization is actually easier, in addition to being more modular and not tying the logical relation to the particularities of the chosen reduction strategy.

7.2 Lean for PL Metatheory

The proof development involved in this report consists of nine files, as mentioned throughout. Table 1 gives the number of non-blank, non-comment lines of code for each file, which roughly reflects the amount of proof effort I required. The `Syntax.lean` file involves a lot of renaming and substitution proofs, but I copied these from prior proof developments with minimal changes to adapt them to CBPV. The actual semantic proofs in `NormalInd.lean`, `OpenSemantics.lean`, and `Soundness.lean` took roughly equal amounts of time to complete. As said in Section 6, the proofs in `NormalAcc.lean` along with its dependency on `Reduction.lean` took nearly as much time to complete as the rest of the semantic proofs, while `LeftmostOutermost.lean` mostly mirrors `Reduction.lean`. The entire project, excluding this report, took about a week to complete.

File	LOC
`Syntax.lean`	252
`Typing.lean`	48
`NormalInd.lean`	192
`OpenSemantics.lean`	107
`Soundness.lean`	114
`Normalization.lean`	48
`Reduction.lean`	187
`NormalAcc.lean`	291
`LeftmostOutermost.lean`	276

Table 1: Lean development files and lines of code

One of the main technical challenges in mechanizing CBPV in Lean is dealing with mutual induction. While Lean currently supports mutually defined inductive types, the options for eliminating them are somewhat buggy and restricted. At the lower level, Lean generates independent eliminators (which they call *recursors*) for each of the mutual inductives, each with motives for all mutual inductives, similar to what `Scheme Induction for ...  with Induction for ...` generates in Rocq.

Lean's induction tactic produces applications of the appropriate recursor, but does not support mutual inductives. In cases where induction on only a single inductive from the mutual definitions is needed, the induction tactic can be supplied with a custom recursor, in this case the appropriate inductive's recursor with all other motives instantiated to trivial propositions. Unfortunately, this fails because the tactic cannot handle induction cases with different numbers of indices in the inductive type, even if those other cases are eventually trivial.

Lean supports structural recursion on mutual inductives, which is implemented via a recursor on an auxiliary inductive that enables strong induction. However, this feature is currently a little buggy, and fails

on some more complex inductive types.[3] Futhermore, writing proofs as recursive functions is unconventional, and isn't amenable to even simple automation that can automatically apply induction hypotheses.

Of course, it's possible to manually apply the recursors that Lean generates for each mutual inductive. This is inconvenient for several reasons: motives need to be explicitly specified, even when they're inferrable from the goals; cases with trivial motives still need to be explicitly proven; and induction cases are duplicated across the recursors. A partial solution, especially for the last inconvenience, is to generate the same kind of combined recursor that `Combined Scheme ...` does for Rocq, but it still requires explicit application. The mutual induction tactic I had implemented (outside of this project) aims to solve these issues.

In short, my mutual induction tactic targets multiple goals at a time. Applying the tactic on targets in mutually defined inductives from different goals applies their respective recursors, inferring motives from those goals, but deduplicates the induction cases, and introduces these cases as new subgoals. If a target and goal for a particular inductive is missing, the motive for that inductive is set as a trivially inhabited type, and all cases for that motive are automatically solved.

The mutual induction tactic has worked very well for this project. The mutual inductives involved are the value and computation terms and types (2 and 2), their typing judgements (2), the logical relation (2), the small-step reduction relation (2), and the mutually inductive definition of strongly neutral and normal terms and head reduction (4). The tactic was used for every proof by mutual induction mentioned here. Additionally, it was used twice for single induction (*i.e.* all other motives are trivial): rule SN-FORCE-INV and Extensionality.

Because the mutual induction tactic is merely a tactic and not a top-level construct, some amount of manipulation is still required to massage the proof state into one where the tactic can be used. Specifically, mutual theorems are first stated as some conjunction $(\forall v, P\ v) \land (\forall m, Q\ m)$, then split into two separate goals with targets v, m introduced into the context, before mutual induction is applied to v and m with inferred motives P and Q. This means that using the theorems individually requires projecting from the conjunction. There are discussions underway for introducing top-level syntax for mutual theorems, similar to Rocq's `Theorem ... with ...` vernacular, so that multiple goals are introduced automatically with their hypotheses.[4]

Aside from mutual induction, the other technical challenge is proving all the renaming and substitution lemmas. There is no library like Autosubst 2 [3] to automate defining and proving these lemmas. In multiple places, substitutions need to be massaged into different propositionally equal forms, and this massaging requires knowing exactly what equality is needed. For PL metatheory that doesn't involve mutual definitions but does involve some notion of binding and abstraction, the lack of a library for substitution automation is perhaps the biggest barrier to Lean being used for PL.

References

[1] A. Abel, G. Allais, A. Hameer, B. Pientka, A. Momigliano, S. Schäfer, and K. Stark. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming*, 29:e19, 2019. doi:10.1017/S0956796819000170.

[2] Y. Forster, S. Schäfer, S. Spies, and K. Stark. Call-by-push-value in Coq: operational, equational, and denotational theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 118–131, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362221. doi:10.1145/3293880.3294097.

[3] K. Stark, S. Schäfer, and J. Kaiser. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 166–180, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362221. doi:10.1145/3293880.3294101.

---

[3] https://github.com/leanprover/lean4/issues/1672
[4] https://leanprover.zulipchat.com/#narrow/channel/239415-metaprogramming-.2F-tactics/topic/mutual.20induction.20tactic/near/504421657