# Developing a Mechanized Denotational Semantics for IMP


Penn Engineering
University *of* Pennsylvania

Elan Roth



COL 2025 – CURF Undergraduate Research Funding

Advised by Stephanie Weirich, Steve Zdancewic, and Nick Rioux


## IMP

- Simple imperative programming language developed in 1960s
- Turing complete (with While)
- Inductively defined types make it easy to work with

```
Inductive aexp : Type :=
 | ANum (n : nat)
 | AId (x : string)
 | APlus (a1 a2 : aexp)
 | AMinus (a1 a2 : aexp)
 | AMult (a1 a2 : aexp).

Inductive com : Type :=
 | CSkip
 | CAsgn (x : string) (a : aexp)
 | CSeq (c1 c2 : com)
 | CIf (b : bexp) (c1 c2 : com)
 | CWhile (b : bexp) (c : com).

Definition loop : com :=
 <{ while true do skip end }>.
```

```
Inductive bexp : Type
:=
 | BTrue
 | BFalse
 | BEq (a1 a2 : aexp)
 | BNeq (a1 a2 : aexp)
 | BLe (a1 a2 : aexp)
 | BGt (a1 a2 : aexp)
 | BNot (b : bexp)
 | BAnd (b1 b2 : bexp).

Definition factorial : com :=
 <{ Z := X;
    Y := 1;
    while Z <> 0 do
      Y := Y * Z;
      Z := Z - 1
    end }>.
```

## Operational Semantics

- How a computer actually runs a piece of code
- Evaluation and computation
- Uses $\Sigma$ as a set of all possible states where $\sigma \in \Sigma$ maps from variables to values

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 + a_1, \sigma\rangle \to n} \quad \textit{Plus}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 - a_1, \sigma\rangle \to n} \quad \textit{Minus}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 * a_1, \sigma\rangle \to n} \quad \textit{Multiply}$$

*where n is the *result of computing* $n_0$ and $n_1$

$$\frac{\langle b, \sigma\rangle \to \mathbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma''\rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma'}$$

## Denotational Semantics

### Motivation

- Using mathematics as a tool to endow meaning to IMP
- Built-in compositionality
- Compilation or translation to a "global language" by forgetting syntax

### Mathematical Definitions

$$\mathcal{A} : \mathbf{Aexp} \to (\Sigma \to \mathbb{N})$$
$$\mathcal{B} : \mathbf{Aexp} \to (\Sigma \to \mathbf{T})$$
$$\mathcal{C} : \mathbf{Aexp} \to (\Sigma \to \Sigma)$$

$$\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$$
$$\mathcal{A}[\![X]\!] = \{(\sigma, \sigma(X)) \mid \sigma \in \Sigma\}$$
$$\mathcal{A}[\![n_0 \star n_1]\!] = \{(\sigma, n_0 \star n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \;\&\; (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$

$$\mathcal{B}[\![a_0 = a_1]\!] = \{(\sigma, \mathbf{true}) \mid \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\} \cup \{(\sigma, \mathbf{false}) \mid \mathcal{A}[\![a_0]\!]\sigma \neq \mathcal{A}[\![a_1]\!]\sigma\}$$

### Conquering *While*

$w \equiv \mathbf{while}\ b\ \mathbf{do}\ c. \qquad w \sim \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$

$$\begin{aligned}
\mathcal{C}[\![w]\!] &= \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] \\
&= \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true}\ \&\ (\sigma, \sigma') \in C[\![c; w]\!]\} \cup \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\} \\
&= \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true}\ \&\ (\sigma, \sigma') \in C[\![w]\!] \circ C[\![c]\!]\} \cup \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\}
\end{aligned}$$

- Finding a fixed point of $\Gamma$
- Using recursion and self-reference, we can mathematically define a denotation to a potentially non-terminating procedure
- Proven to exist by Knaster-Tarski

$$\begin{aligned}
\Gamma(\phi) &= \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true}\ \&\ (\sigma, \sigma') \in \phi \circ C[\![c]\!]\} \cup \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\} \\
&= \{(\sigma, \sigma') \mid \exists \sigma''. \mathcal{B}[\![b]\!]\sigma = \mathbf{true}\ \&\ (\sigma, \sigma'') \in C[\![c]\!]\ \&\ (\sigma'', \sigma') \in \phi\} \cup \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\}
\end{aligned}$$

## Formalization in Coq

### Coq

- Interactive theorem prover
- Intimately connected to dependent types and type theory in general
- Implemented with a combination of Gallina and Ltac
- Allows for users to define their own objects and prove properties of them

### Implementing a Fixed Point Operator and Γ

```
Definition Fix (f : (A -> A -> Prop) -> (A -> A -> Prop)) :
          (A -> A -> Prop) :=
 fun st1 st2 => exists n, (apply_n_times n (fun x y => False)) st1 st2

Definition gamma (bd  : state -> bool -> Prop)
                 (cd  : state -> state -> Prop)
             (phi : state -> state -> Prop): state -> state -> Prop :=
 fun st1 st2 => (bd st1 true /\ (phi • cd) st1 st2)
                \/ (bd st1 false /\ st1 = st2).
```

### Proving Full Abstraction and Semantic Coincidence

```
Lemma aeval_deterministic : forall (a: aexp) (st: state),
(exists (n: nat), aeval st a = n) /\
(forall (m n: nat), (aeval st a = n /\ aeval st a = m) -> n = m).
```

- Proof that operational semantics always output a single, unique value
- Only possible for *aexp* and *bexp* due to *loop*

```
Lemma a_den_deterministic : forall (a: aexp) (st: state),
(exists (n: nat), a_den a st n) /\
(forall (m n: nat), (a_den a st n /\ a_den a st m) -> n = m).
```

- Proof that denotational semantics always output a single, unique value
- Again, only possible for *aexp* and *bexp* due to *loop*

```
Lemma aeval_equiv_a_den : forall (a: aexp) (st: state) (n: nat),
aeval st a = n <-> a_den a st n.

Lemma beval_equiv_b_den : forall (b: bexp) (st: state) (b': bool),
beval st b = b' <-> b_den b st b'.

Lemma ceval_equiv_c_den : forall (c: com) (st1 st2: state),
ceval st1 c = st2 <-> c_den c st1 st2.
```

- Main theorems that show Full Abstraction proving operational and denotational semantics will terminate equivalently
- Even stronger is that these semantics agree at each step

## Formalization in Coq

- Developing semantics for Programming Computable Functions (PCF) other than the existing Scott model constructed by Dana Scott
- Similar development for the Untyped Lambda Calculus
- Utilize tools such as ITrees, free monads, cartesian closedness