A Mechanized Proof in Coq of the Type Soundness of Core L^3

Milestone 1

Yawar Raza

Review

- Type Soundness
  - "Is this type system *really* detecting all the type-errors a program can have?"
  - Counterexamples can be hard to find. Let's prove it mathematically instead!
- Mechanization
  - "Is this hand-written proof *really* correct and free of mistakes?"
  - Errors in proves can be hard to find. Let's have the computer check the proof!

Section 1

Introduction to L^3

Overview of L^3

- L^3 is introduced by the paper *L^3: A Linear Language with Locations*, by Ahmed, Fluet, and Morrisett.
- The goal of L^3 is to support *strong updates*. A strong update assigns a value of a different type to a reference cell.
- To make this sound, L^3 uses a linear type system; linear values must be used exactly once.
- L^3 uses linear "capability" values to ensure that reads are made using the most up-to-date type of the cell's contents.
- The paper presents both Core L^3 and Extended L^3, the latter having additional functionality.

An Example L^3 Program

$$
\begin{aligned}
\ulcorner \rho, n_1 \urcorner &= \texttt{new}\ * \\
\langle c_1, \hat{p} \rangle &= n_1 \\
\langle \hat{p}_a, \hat{p}_b \rangle &= \texttt{dupl}\ \hat{p} \\
!p_a &= \hat{p}_a \\
\langle c_2, u \rangle &= \texttt{swap}\ c_1\ p_a\ (\lambda x.\ x) \\
!p_b &= \hat{p}_b \\
\ulcorner \rho', f \urcorner &= \texttt{free}\ \ulcorner \rho, \langle c_2, p_b \rangle \urcorner \\
& f\ u
\end{aligned}
$$

$$
\begin{aligned}
\hat{p}, \hat{p}_a, \hat{p}_b &: !(\mathsf{Ptr}\ \rho) \\
p_a, p_b &: \mathsf{Ptr}\ \rho \\
c_1 &: \mathsf{Cap}\ \rho\ \mathbf{I} \\
c_2 &: \mathsf{Cap}\ \rho\ (\mathbf{I} \multimap \mathbf{I}) \\
u &: \mathbf{I} \\
f &: \mathbf{I} \multimap \mathbf{I}
\end{aligned}
$$

Section 2

Locally Nameless Representation

Issues with Named Variables

- α-equivalence
  - $\lambda x.\ x$ and $\lambda y.\ y$ are equivalent on paper.
  - `lam "x" (var "x")` and `lam "y" (var "y")` are not equal in Coq.
- Capturing substitution
  - $(\lambda y.\ \lambda z.\ y)\ x$ evaluates to $\lambda z.\ x$. Note that x is *free*, while y is *bound*.
  - The α-equivalent $(\lambda y.\ \lambda x.\ y)\ x$ evaluates to $\lambda x.\ x$, which is not equivalent.
  - Substitution *captured* the free x and made it bound because it had the same name as an inner lambda parameter.
- Paper notation can assume implicit renaming, but mechanized proofs cannot implicitly rename variables.

de Brujin Indices

- Variables don't have names. Expressions refer to parameters using a number.
- 0 refers to the parameter of the inner-most containing lambda, 1 to the next inner-most, etc.
  - $\lambda y.\ \lambda x.\ x$ turns into $\lambda.\ \lambda.\ 0$.
  - $\lambda y.\ \lambda x.\ y$ turns into $\lambda.\ \lambda.\ 1$.
- The numbers are contextual: $(\lambda.\ 0)\ (\lambda.\ 0)$ corresponds to $(\lambda x.\ x)\ (\lambda y.\ y)$.
- Expressions have a unique representation; no need for α-equivalence.

Free Variables

- How is $\lambda x.\, y$ represented?
- Open terms are evaluated in conjunction with an *environment* mapping variables to the values they hold.
  - A named-variable environment can be a list of name-value pairs, e.g. $[(y, 4), (z, 6)]$.
- de Brujin indices solution
  - Free variables are represented by indices into an (ordered) environment.
  - Environment is a list of values without variable names.
- *Locally nameless* solution
  - Keep free variables as names.
  - L^3 needs to split the environment into arbitrary partitions, so indices into the environment wouldn't be stable.

Infrastructure for Locally Nameless

- Syntax separates free variables (having variable names) and bound variables (having de Brujin indices).
- Locally-closed predicate
  - Reject expressions that use indices that are too big to refer to a lambda parameter.
- Opening
  - Substituting the outermost bound variable with a free variable or other expression.
- Variable closing
  - Turning a free variable into the outermost bound variable.

Milestones

- Complete
  - Mechanized the syntax, most of the locally nameless infrastructure, and some of the operational semantics.
- Milestone 2
  - Mechanizing the rest of the operational semantics, the type system, and the semantic interpretations.
- Milestone 3
  - Mechanizing most of the type soundness proof; may require focusing on helper lemmas instead of the primary proof cases.