CSE 505: Programming Languages

Lecture 12 — Safely Extending STLC: Progress, Preservation, Lets, Branches

Zach Tatlock
Fall 2013

Adding Stuff

Time to use STLC as a foundation for understanding other common language constructs

We will add things via a principled methodology thanks to a proper education

- Extend the syntax
- Extend the operational semantics
  - Derived forms (syntactic sugar), or
  - Direct semantics
- Extend the type system
- Extend soundness proof (new stuck states, proof cases)

In fact, extensions that add new types have even more structure

Review

\[ e ::= \lambda x.e \mid x \mid e \ e \mid c \]
\[ v ::= \lambda x.e \mid c \]
\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]
\[ \Gamma ::= \cdot \mid \Gamma, x : \tau \]

\[
\begin{array}{c}
(\lambda x. e) v \rightarrow e[v/x] \\
\end{array}
\]
\[
\begin{array}{c}
e_1 \rightarrow e'_1 \\
e_1 \ e_2 \rightarrow e'_1 \ e_2 \\
v \ e_2 \rightarrow v \ e'_2 \\
\end{array}
\]

\[ e[e'/x] : \text{capture-avoiding substitution of } e' \text{ for free } x \text{ in } e \]

\[
\begin{array}{c}
\Gamma, x : \tau_1 \vdash e : \tau_2 \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash c : \text{int} \\
\Gamma \vdash x : \Gamma(x) \\
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \\
\Gamma \vdash e_2 : \tau_2 \\
\end{array}
\]

\[ \Gamma \vdash e_1 \ e_2 : \tau_1 \]

Preservation: If \( \cdot \vdash e : \tau \) and \( e \rightarrow e' \), then \( \cdot \vdash e' : \tau \).
Progress: If \( \cdot \vdash e : \tau \), then \( e \) is a value or \( \exists e' \) such that \( e \rightarrow e' \).

Let bindings (CBV)

\[ e ::= \ldots \mid \text{let } x = e_1 \text{ in } e_2 \]

\[
\begin{array}{c}
e_1 \rightarrow e'_1 \\
\end{array}
\]

\[
\begin{array}{c}
\text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e'_1 \text{ in } e_2 \\
\end{array}
\]

\[
\begin{array}{c}
\text{let } x = v \text{ in } e \rightarrow e[v/x] \\
\end{array}
\]

(Also need to extend definition of substitution...)

Progress: If \( e \) is a let, 1 of the 2 new rules apply (using induction)

Preservation: Uses Substitution Lemma

Substitution Lemma: Uses Weakening and Exchange
Derived forms

- `let` seems just like `λ`, so can make it a derived form
  - `let x = e1 in e2` "a macro" / "desugars to" `(λx. e2) e1`
  - A "derived form"

  (Harder if `λ` needs explicit type)

Or just define the semantics to replace `let` with `λ`:

```
let x = e1 in e2 → (λx. e2) e1
```

These 3 semantics are different in the state-sequence sense

```
(e1 → e2 → ... → en)
```

- But (totally) equivalent and you could prove it (not hard)

Note: ML type-checks `let` and `λ` differently (later topic)

Note: Don’t desugar early if it hurts error messages!

---

Booleans and Conditionals

```
e ::= ... | true | false | if e1 e2 e3
v ::= ... | true | false
τ ::= ... | bool
```

```
e1 → e′

if e1 e2 e3 → if e′ e2 e3
```

```
if true e2 e3 → e2
if false e2 e3 → e3
```

```
Γ ⊢ e1 : bool  Γ ⊢ e2 : τ  Γ ⊢ e3 : τ
Γ ⊢ if e1 e2 e3 : τ
```

```
Γ ⊢ true : bool  Γ ⊢ false : bool
```

Also extend definition of substitution (will stop writing that)... Notes: CBN, new Canonical Forms case, all lemma cases easy