Programming Language Theory

Type Safety of the Simply-Typed Lambda Calculus
Outline

▶ Type-safety proof
  ▶ Detailed proof posted on website
▶ Discuss the proof
  ▶ Consider lemma dependencies and what they represent
  ▶ Consider elegance of inverting static and dynamic derivations
▶ Next lecture: Extend STLC
  ▶ Pairs, sums, recursion, …
  ▶ For each, sketch proof additions
  ▶ Discuss the general approach
▶ On homework: References
▶ Further ahead: More flexible typing via polymorphism
Lambda Calculus (with constants) Syntax and Substitution

\[ e ::= c \mid x \mid \lambda x. \, e \mid e \, e \]
\[ v ::= c \mid \lambda x. \, e \]

Work with terms “up to renaming of bound variables” (“up to alpha-conversion”).

\[ FV(c) = \{\} \quad FV(e_f \, e_a) = FV(e_f) \cup FV(e_a) \]
\[ FV(x) = \{x\} \quad FV(\lambda x. \, e_b) = FV(e_b) \setminus \{x\} \]

\[
e_1[e_2/x] = e_3
\]

\[
c[e/x] = c \quad x[e/x] = e \quad y \neq x \quad y \not\in FV(e)
\]
\[
e_b[e/x] = e'_b \quad y \neq x \quad y \not\in FV(e)
\]
\[
(\lambda y. \, e_b)[e/x] = \lambda y. \, e'_b
\]
\[
e_f[e/x] = e'_f \quad e_a[e/x] = e'_a
\]
\[
(e_f \, e_a)[e/x] = e'_f \, e'_a
\]

Substitution usually treated as a metafunction, not a judgement.
Lambda Calculus (with constants) Operational Semantics

Small-step, left-to-right, call-by-value (CBV) operational semantics:

\[ e \rightarrow_{cbv} e' \]

\[
\begin{align*}
\text{E-Apply} & \quad \frac{\left(\lambda x \cdot e_b\right) v_a \rightarrow_{cbv} e_b[v_a/x]}{} \\
\text{E-AppF} & \quad \frac{e_f \rightarrow_{cbv} e'_f}{e_f e_a \rightarrow_{cbv} e'_f e_a} \\
\text{E-AppA} & \quad \frac{e_a \rightarrow_{cbv} e'_a}{v_f e_a \rightarrow_{cbv} v_f e'_a}
\end{align*}
\]

We say that an expression \( e \) is stuck if
- \( e \) is not a value, and there is no \( e' \) such that \( e \rightarrow_{cbv} e' \)

We say that an expression \( e \) gets stuck if
- \( e \rightarrow^{*}_{cbv} e' \), and \( e' \) is stuck.

Avoid stuck expressions to:
- Catch bugs (never want to be stuck)
- Ease implementation (never need to check for being stuck)
Simply-Typed Lambda Calculus (with constants) Type System

Type system to classify (accept or reject) \( \lambda \)-terms.

\[
\tau ::= \text{int} \mid \tau \rightarrow \tau \quad \Gamma ::= \cdot \mid \Gamma, x : \tau
\]

\[
\Gamma \vdash x : \tau
\]

\[
\begin{align*}
\text{C-Hit} & \quad \frac{\Gamma, x : \tau \vdash x \mapsto \tau}{\Gamma, x : \tau, x : \tau \vdash x \mapsto \tau} \\
\text{C-Miss} & \quad \frac{x \neq y}{\Gamma' \vdash x \mapsto \tau} \\
\end{align*}
\]

\[
\begin{align*}
\text{T-Const} & \quad \frac{}{\Gamma \vdash c : \text{int}} \\
\text{T-Var} & \quad \frac{}{\Gamma \vdash x : \tau} \\
\text{T-Lam} & \quad \frac{\Gamma, x : \tau_a \vdash e_b : \tau_r}{\Gamma \vdash \lambda x. e_b : \tau_a \rightarrow \tau_r} \\
\text{T-App} & \quad \frac{\Gamma \vdash e_f : \tau_a \rightarrow \tau_r \quad \Gamma \vdash e_a : \tau_a}{\Gamma \vdash e_f e_a : \tau_r}
\end{align*}
\]
Type Soundness: Main Theorem and Lemmas

A program that type checks does not get stuck.

Theorem (Type Safety): If \( \cdot \vdash e : \tau \) and \( e \rightarrow_{\text{cbv}}^{*} e' \), then either \( e' \) is a value or there exists \( e'' \) such that \( e' \rightarrow_{\text{cbv}} e'' \).

Follows from two key lemmas:

- Lemma (Progress): If \( \cdot \vdash e' : \tau \), then either \( e' \) is a value or there exists an \( e'' \) such that \( e' \rightarrow_{\text{cbv}} e'' \).

- Lemma (Preservation): If \( \cdot \vdash e : \tau \) and \( e \rightarrow_{\text{cbv}} e^\dagger \), then \( \cdot \vdash e^\dagger : \tau \).

Proof of Type Safety given Progress and Preservation:

- By induction on (the derivation) \( e \rightarrow_{\text{cbv}}^{*} e' \).

  - \( e \rightarrow_{\text{cbv}}^{*} e' \equiv e \rightarrow_{\text{cbv}}^{*} e \rightarrow_{\text{cbv}} e \equiv e \rightarrow_{\text{cbv}}^{*} e : \) By Progress.

  - \( e \rightarrow_{\text{cbv}}^{*} e' \equiv e \rightarrow_{\text{cbv}}^{*} e \rightarrow_{\text{cbv}} e^\dagger \rightarrow_{\text{cbv}} e' \equiv e \rightarrow_{\text{cbv}}^{*} e' : \) By Preservation and IH.
Type Soundness: Auxiliary Lemmas

Lemma (Canonical Forms): If $\cdot \vdash v : \tau$, then
1. if $\tau = \text{int}$, then $v = c$ (for some $c$)
2. if $\tau = \tau_1 \rightarrow \tau_2$, then $v = \lambda x. e$ (for some $\lambda x. e$)

Lemma (Substitution): If $\Gamma, x : \tau_x \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau_x$, then $\Gamma \vdash e_1[e_2/x] : \tau$.

Lemma (Exchange): If $\Gamma, x : \tau_x, y : \tau_y \vdash e : \tau$ and $x \neq y$, then $\Gamma, y : \tau_y, x : \tau_x \vdash e : \tau$.

Lemma (Weakening): If $\Gamma \vdash e : \tau$ and $x \not\in \text{Dom}(\Gamma)$, then $\Gamma, x : \tau_x \vdash e : \tau$. 
Lemma dependencies

Safety (evaluation never gets stuck)
- Progress (well-typed not stuck yet)
  - Canonical Forms (primitive reductions apply)
- Preservation (to stay well-typed)
  - Substitution ($\beta$-reduction stays well-typed)
    - Weakening (substituting under nested $\lambda$s well-typed)
    - Exchange (technical point)

Comments:
- Substitution strengthened to open terms for the proof
- When we (and by “we”, I mean “you”) add heaps, Preservation will use Weakening directly
Summary

What may seem a weird lemma pile is a powerful recipe:

Soundness: We don’t get stuck because our induction hypothesis (well typedness) holds (Preservation) and stuck terms aren’t well typed (contrapositive of Progress).

Preservation holds by induction on typing because we replace a subterm with another subterm of same type; for the tricky subterm replacement ($\beta$-reduction), we use Substitution.

- Substitution must work over open terms and requires Weakening and Exchange.

Progress holds by induction on typing because either a subexpression progresses or we can make a primitive reduction (using Canonical Forms).
Induction on derivations – Another Look

The app cases are really elegant and worth mastering: \( e = e_f \ e_a \).
For Preservation, lemma assumes \( \Gamma \vdash e_f \ e_a : \tau \) and \( e_f \ e_a \rightarrow_{\text{cbv}} e' \).

Inverting the typing derivation ensures that it has the form:

\[
\frac{D_f}{\Gamma \vdash e_f : \tau_a \rightarrow \tau} \quad \frac{D_a}{\Gamma \vdash e_a : \tau_a} \quad \frac{\Gamma \vdash e_f \ e_a : \tau}
\]

One subcase: If \( e_f \ e_a \rightarrow_{\text{cbv}} e'_f \ e_a \), then inverting that derivation ensures:

\[
\frac{D_s}{e_f \rightarrow_{\text{cbv}} e'_f} \quad \frac{\ e_f \ e_a \rightarrow_{\text{cbv}} e'_f \ e_a
\]
The inductive hypothesis (applied to $\Gamma \vdash e_f : \tau_a \rightarrow \tau$ and $e_f \rightarrow_{cbv} e'_f$) gives a derivation of this form:

$$
\begin{array}{c}
\frac{D_{f'}}{
\Gamma \vdash e'_f : \tau_a \rightarrow \tau
}
\end{array}
$$

Therefore, a derivation of this form exists:

$$
\begin{array}{c}
\frac{D_{f'} \quad D_a}{
\frac{\Gamma \vdash e'_f : \tau_a \rightarrow \tau}{
\Gamma \vdash e'_f \ e_a : \tau}
\end{array}
$$

(The app case of the Substitution Lemma is similar but we apply induction twice to get the new derivation.)