CS 611
Advanced Programming Languages
Andrew Myers
Cornell University
Lecture 28
Strong Normalization, Logical relations
1 Nov 00

Strong normalization
• Every program in $\lambda^\rightarrow$ terminates. Obvious?
  – Reduction can increase size of an expression
  – Reduction can increase number of contained lambda expressions
    \[(\lambda f :: \text{int} \to \text{int}. f (\text{int} (1)) \land \lambda y :: \text{int} \to \text{int} (* y 2))\]
  – Untyped lambda calculus is not strongly normalizing
• Idea: size of types decreases
• Proof strategy: define set of strongly normalizing expressions $SN_\tau$ for every type $\tau$, show by induction on type derivation that expression of type $\tau$ is a member of $SN_\tau$
• Method of logical relations: relations on expressions indexed by types

Soundness for SOS
• Last time: soundness of typing rules for structural operational semantics
  – “$e$ is well-typed” $\vdash e : \tau$
  – “$e$ does not get stuck” $\forall \delta', e \rightarrow^* e' \Rightarrow e e' : \tau$
  – “$e$ is typable” $\vdash e : \tau$ (since $T \vdash e : \tau$)
• Soundness: “$e$ is typable” $\Rightarrow$ “$e$ does not get stuck”
• Three parts to proof:
  – Preservation/Subject reduction $\vdash e : \tau \land e \rightarrow^* e' \Rightarrow \vdash e' : \tau$
  – Progress $\vdash e : \tau \Rightarrow (e e' \lor \exists e'' e' \rightarrow e'')$
  – Induction on number of steps (generic)
• New tool: induction on type derivation
• Real languages much harder...

Stable expressions
• Problem: induction hypothesis is not strong enough to handle application expressions.
• Strengthen induction hypothesis: define subset of strongly normalizing expressions (the stable expressions); show all expressions in $\lambda^\rightarrow$ are stable.
• Stable expressions are strongly normalizing and result in strongly normalizing expressions when applied to other strongly normalizing expressions.
• $T_\tau$ is the set of stable expressions of type $\tau$.
• Define inductively (note $e \Uparrow v \equiv e \rightarrow^* v$)
  \[T_{\text{int}} = \{ e | e : \text{int} \land e \Uparrow n \}\]
  \[T_{\tau \rightarrow} = \{ e | e : \tau \rightarrow \tau' \land (e e') \in T_{\tau'} \}\]
• Goal: $\vdash e : \tau \Rightarrow e \in T_{\tau}$

Strategy
\[
T_{\text{int}} = \{ e | e : \text{int} \land e \Uparrow n \}
\]
\[
T_{\tau \rightarrow} = \{ e | e : \tau \rightarrow \tau' \land e \Uparrow v \land (\forall \nu \nu \in T_{\tau} (e e') \in T_{\tau'}) \}
\]
Goal: $\vdash e : \tau \Rightarrow e \in T_{\tau}$ (since $T_{\tau} \subseteq SN_\tau$)
• Will use induction on type derivation for $e$
• Problem: rule for typing $\lambda$ expr adds to type context $\Gamma$. Need to extend goal to allow it to be proved inductively: use substitution operators
• Introduce function $\gamma$ mapping variables to expressions. $\gamma : \text{Var} \rightarrow \text{Exp}$
• $\gamma$ only substitutes stable expressions of the right type: $\gamma : \Gamma \Rightarrow \forall x \in \text{dom}(\Gamma) \cdot \gamma(x) \in T_{\Gamma(x)}$

Substitution function
• Given any function $\gamma$, we can define a related function $\gamma'$ mapping $\text{Expr} \rightarrow \text{Expr}$ and performing all the substitutions specified by $\gamma$:
  \[\gamma'[x] = \gamma(x) \text{ if } x \in \text{dom}(\gamma)\]
  \[\gamma'[x] = x \text{ if } x \notin \text{dom}(\gamma)\]
  \[\gamma'[n] = n\]
  \[\gamma'[e_0 e_1] = \gamma'[e_0] \cdot \gamma'[e_1]\]
  \[\gamma'[\lambda x :: \tau. e] = \lambda x :: \tau. \gamma'[e]\]

$\gamma'$ is identical to $\gamma$ except that it does not map $x$. 
Refined goal

- Original goal: show all expressions are stable
  \[ \vdash e : \tau \Rightarrow e \in T_\tau \]
- Suppose we can prove the following goal:
  \[ \Gamma \vdash e : \tau \Rightarrow \forall \gamma \in \Gamma, \gamma[e] \in T_\tau \]

- Now consider \( \Gamma = \emptyset \). The only \( \gamma \) satisfying this type context is the identity mapping. Therefore, our refined goal becomes our original goal.
- Substitution Lemma: \( \Gamma \vdash e : \tau \Rightarrow \forall \gamma \in \Gamma, \gamma[e] : \tau \)
  - Generalization of proof from last class
- Now we turn the inductive crank.

Part I

To show: \( \Gamma \vdash e : \tau \Rightarrow \forall \gamma \in \Gamma, \gamma[e] \in T_\tau \)
- Integers: \( \Gamma \vdash n : \text{int} \Rightarrow \forall \gamma \in \Gamma, n \in T_{\text{int}} \)
- Variables: \( \Gamma \vdash x : \Gamma(x) \Rightarrow \forall \gamma \in \Gamma, \gamma[x] \in T_{\Gamma(x)} \)
  - if \( \gamma \in \Gamma \) then \( \gamma[x] = \gamma(x) \in T_{\Gamma(x)} \) by definition.
- Application: \( \Gamma \vdash (e_0, e_1) : \tau \)
  - Consider a \( \gamma \) such that \( \gamma \in \Gamma \)
  - \( \gamma[e_0, e_1] = \gamma[e_0] \gamma[e_1] \)
  - From type judgement: \( \Gamma \vdash e_0 : \tau_0, \Gamma \vdash e_1 : \tau_1 \)
  - Inductive hypothesis gives us \( \gamma[e_0] \) and \( \gamma[e_1] \) are stable; therefore their application is too.

\[ T_\tau = \{ e \mid e : \tau \Rightarrow 0 \} \in T_\tau \]

Part II

To show: \( \Gamma \vdash e : \tau \Rightarrow \forall \gamma \in \Gamma, \gamma[e] \in T_\tau \)

\[ \Gamma \vdash (\lambda x : \tau . e) : \tau \Rightarrow \forall \gamma \in \Gamma, \gamma[\lambda x : \tau . e] \in T_{\tau \Rightarrow \tau} \]
- Assume LHS, consider arbitrary \( \gamma \vdash \Gamma \)

Recall \( T_{\tau \Rightarrow \tau} = \{ e \mid e : \tau \Rightarrow e' \land e' \in \nu \land (\forall \gamma \in \Gamma, (e' \in T_\tau)) \}
- Substitution Lemma: \( \gamma[\lambda x : \tau . e] : \tau \Rightarrow \tau' \)
- \( \gamma[\lambda x : \tau . e] \) is already a value so \( \nu \)
- Need \( \forall \nu \in T_\tau \). \( \gamma[\lambda x : \tau . e] : \tau \Rightarrow \tau' \)
- \( \gamma[\lambda x : \tau . e] \in T_\tau \)
- From typing rule: \( \Gamma \vdash x : \tau \Rightarrow e : \tau' \)
- Apply induction hypothesis, instantiate on \( \gamma' \):
  \( \gamma' \vdash \Gamma(x \Rightarrow \tau) \Rightarrow \gamma'[e] \in T_{\tau} \)
  \( \gamma' \vdash \Gamma(x \Rightarrow \tau) (e' \in T_\tau) \)

QED

Adequacy

- Denotational semantics are adeqate with respect to operational semantics if:

- Operational evaluation produces one of the values allowed by denotational semantics
  \[ e \Rightarrow v \land e : \tau \Rightarrow C \cdot (\vdash e : \tau) \rho_0 = C \cdot (\vdash v : \tau) \rho_0 \]
- They agree on observable divergence: \( \exists v . e \Rightarrow v \land e : \tau \Rightarrow C \cdot (\vdash e : \tau) \rho_0 \neq \perp \)
- and also on ground types (e.g. int)
  \[ e \Rightarrow v \land e : \text{int} \Rightarrow C \cdot (\vdash e : \text{int}) \rho_0 = v \]
  \[ e \Rightarrow v \land e : \tau \Rightarrow C \cdot (\vdash e : \tau) \rho_0 = C \cdot (\vdash v : \tau) \rho_0 \]

Coming soon: richer types

- Recursive types
- Polymorphic types
- Subtyping
- Objects