CS 611: Lecture 29

Soundness of Denotational Semantics and Strong Normalization in the Simply-Typed Lambda Calculus

November 5, 1999

Cornell University Computer Science Department
Andrew Myers

Soundness for SOS

- Last time: soundness of typing rules for structural operational semantics
- "e is typable"

$$\vdash e : \tau$$

- "e does not get stuck":

$$\forall e' \,.\, e \rightarrow^* e' \Rightarrow \mathit{value}(e') \vee \exists\, e'' \,.\, e' \rightarrow e''$$

- Soundness: "e is typable" ⇒ "e does not get stuck"
- Three parts to proof:
  - Preservation/Subject reduction $\quad \vdash e : \tau \Rightarrow e \rightarrow^* e' \Rightarrow \vdash e' : \tau$
  - Progress $\quad \vdash e : \tau \Rightarrow \mathit{value}(e') \vee \exists\, e'' \,.\, e' \rightarrow e''$
  - Induction on number of steps (generic)
- New tool: induction on height of type derivation
- Real languages much harder...

Outline

- Soundness of denotational semantics
- Agreement between operational and denotational semantics (briefly)
- Proof of strong normalization

Denotational semantics for TLC

- Semantic function $\mathcal{C}$ is applied to type judgements $\mathcal{C}[\![\Gamma \vdash e : \tau]\!]$ rather than just to expressions
- Function $\mathcal{C}$ is defined only for legal type judgements : only typable expressions have a denotation
- Environment ρ must *satisfy* the type context Γ:

$$\rho \models \Gamma \Leftrightarrow \forall x \in \mathrm{dom}(\Gamma) \,.\, \rho(x) \in \mathcal{T}[\![\Gamma(x)]\!]$$

- Denotational semantics are sound with respect to typing if they map expressions to meta-language values of the right type:

$$\rho \models \Gamma \Rightarrow \mathcal{C}[\![\Gamma \vdash e : \tau]\!]\,\rho \in \mathcal{T}[\![\tau]\!]$$

Denotational Semantics

$\mathcal{C}[\![\Gamma \vdash \mathrm{n} : \mathrm{int}]\!]\,\rho = \mathrm{n}$

$\mathcal{C}[\![\Gamma \vdash x : \Gamma(x)]\!]\,\rho = \rho(x)$

$\mathcal{C}[\![\Gamma \vdash e_0\ e_1 : \tau]\!]\,\rho = \mathcal{C}[\![\Gamma \vdash e_0 : \tau_1 \rightarrow \tau]\!]\rho\ (\mathcal{C}[\![\Gamma \vdash e_1 : \tau_1]\!]\,\rho)$

$\mathcal{C}[\![\Gamma \vdash (\lambda\, x : \tau \,.\, e) : \tau \rightarrow \tau']\!]\,\rho =$

$\quad \lambda\, v \in \mathcal{T}[\![\tau]\!] \,.\, \mathcal{C}[\![\Gamma[x \mapsto \tau] \vdash e : \tau']\!]\,\rho\,[x \mapsto v]$

Example:

$\mathcal{C}[\![\Gamma \vdash (\lambda\, \mathrm{x} : \mathrm{int} \,.\, \mathrm{x}) : \mathrm{int} \rightarrow \mathrm{int}]\!]\,\rho$

$= \lambda\, v \in \mathcal{T}[\![\mathrm{int}]\!] \,.\, \mathcal{C}[\![\Gamma[x \mapsto \mathrm{int}] \vdash \mathrm{x} : \mathrm{int}]\!]\,\rho\,[x \mapsto v]$

$= \lambda\, v \in \mathrm{Z} \,.\, v$

= identity function on integers

Type soundness

- To show: $\Gamma \vdash e : \tau \wedge \rho \models \Gamma \Rightarrow \mathcal{C}[\![\Gamma \vdash e : \tau]\!]\,\rho \in \mathcal{T}[\![\tau]\!]$
- Use induction on height of type derivation for type judgement
- Axioms:

$$\mathcal{C}[\![\Gamma \vdash \mathrm{n} : \mathrm{int}]\!]\,\rho = \mathrm{n} \quad \mathrm{n} \in \mathrm{Z}$$

$$\mathcal{C}[\![\Gamma \vdash x : \Gamma(x)]\!]\,\rho = \rho(x) \qquad \rho \models \Gamma \Rightarrow \rho(x) \in \mathcal{T}[\![\Gamma(x)]\!]$$

$$\mathcal{C}[\![\Gamma \vdash e_0\ e_1 : \tau]\!]\,\rho = \underbrace{\mathcal{C}[\![\Gamma \vdash e_0 : \tau_2 \rightarrow \tau]\!]\rho}_{\mathcal{T}[\![\tau_2 \rightarrow \tau]\!] = \mathcal{T}[\![\tau_2]\!] \rightarrow \mathcal{T}[\![\tau]\!]}\ \underbrace{(\mathcal{C}[\![\Gamma \vdash e_1 : \tau_2]\!]\,\rho)}_{\mathcal{T}[\![\tau_2]\!]}$$

Type soundness, cont.

$$\mathcal{C}[\![\Gamma \vdash \lambda x:\tau\,.\,e:\tau\to\tau']\!]\,\rho =$$
$$\lambda v \in \underbrace{\mathcal{T}[\![\tau]\!]}_{\mathcal{T}[\![\tau\to\tau']\!]}\,.\,\underbrace{\mathcal{C}[\![\Gamma[x\mapsto\tau] \vdash e:\tau']\!]\,\rho\,[x\mapsto v]}_{\mathcal{T}[\![\tau']\!]}$$

$$\mathcal{T}[\![\tau]\!]\to\mathcal{T}[\![\tau']\!] = \mathcal{T}[\![\tau\to\tau']\!]$$

$$\frac{\Gamma[x\mapsto\tau] \vdash e:\tau'}{\Gamma \vdash \lambda x:\tau\,.\,e:\tau\to\tau'}$$

$$\rho\,[x\mapsto v] \models \Gamma[x\mapsto\tau] \text{ since } v \in \mathcal{T}[\![\tau]\!]$$

Agreement

- Would like to know that denotational semantics and operational semantics agree

Operational

$$e \to e' \to e'' \to \ldots \to v \quad : \tau$$
$$\mathcal{C}[\![\vdash v:\tau]\!]\,\rho_0 \quad \in \mathcal{T}[\![\tau]\!]$$

Denotational

Adequacy

- Denotational semantics are *adequate* with respect to operational semantics:
- Operational evaluation produces one of the values allowed by denotational semantics

$$e \to^* v \;\wedge\; \vdash e:\tau \;\Rightarrow \mathcal{C}[\![\vdash e:\tau]\!]\,\rho_0 = \mathcal{C}[\![\vdash v:\tau]\!]\,\rho_0$$

- They agree on observable results: divergence

$$\exists v\,.\,e \to^* v \;\wedge\; \vdash e:\tau \;\Leftrightarrow\; \mathcal{C}[\![\vdash e:\tau]\!]\,\rho_0 \neq \bot$$

- and also on ground types (e.g. int)

$$e \to^* v \;\wedge\; \vdash e:\text{int} \;\Leftarrow\; \mathcal{C}[\![\vdash e:\text{int}]\!]\,\rho_0 = v$$

- Proof: last year's CS 611 lecture notes

Strong normalization

- Every program in $\lambda^{\to}$ terminates. Is this obvious?
  - Reduction can increase size of an expression
  - Reduction can increase number of contained lambda expressions

    $((\lambda f:\text{int}\to\text{int}\,.\,(+\,(f\,0)\,(f\,1)))\,(\lambda y:\text{int}\,.\,(*\,y\,2)))$
  - Untyped lambda calculus is *not* strongly normalizing
- Idea: size of *types* decreases
- Proof strategy: define set of strongly normalizing expressions SN_τ for every type τ, show by induction on type derivation that expression of type t is a member of SN_τ.
- Problem: induction hypothesis is not strong enough to handle application expressions.

Stable expressions

- Strengthen induction hypothesis: define subset of strongly normalizing expressions (the *stable* expressions); show all expressions in $\lambda^{\to}$ are stable.
- Stable expressions are strongly normalizing and result in strongly normalizing expressions when applied to other strongly normalizing expressions.
- T_τ is the set of stable expressions of type τ.
- Define inductively (note $e \Downarrow v \;\Leftrightarrow\; e \to^* v$)

$$T_{\text{int}} = \{\, e \mid \vdash e:\text{int} \;\wedge e \Downarrow n \,\}$$
$$T_{\tau\to\tau'} = \{\, e \mid \vdash e:\tau\to\tau' \wedge e \Downarrow v \;\wedge (\forall e' \in T_\tau\,.\,(e\,e') \in T_{\tau'})\}$$

- Goal: $\vdash e:\tau \Rightarrow e \in T_\tau$

Strategy

$T_{\text{int}} = \{\, e \mid \vdash e:\text{int} \;\wedge e \Downarrow n \,\}$

$T_{\tau\to\tau'} = \{\, e \mid \vdash e:\tau\to\tau' \wedge e \Downarrow v \;\wedge (\forall e' \in T_\tau\,.\,(e\,e') \in T_{\tau'})\,\}$

Goal: $\vdash e:\tau \Rightarrow e \in T_\tau$

- Will use induction on type derivation for e
- Problem: rule for typing λ exprs adds to type context Γ. Need to extend form of goal to allow it to be proved inductively: use substitution operators.
- Introduce function γ mapping variables to expressions. γ : Var $\to$ Exp
- γ only substitutes stable expressions of the right type: $\gamma \models \Gamma \Leftrightarrow \forall x \in \text{dom}(\gamma)\,.\,\gamma(x) \in T_{\Gamma(x)}$

Substitution function

- Given any function γ, we can define a related function $\hat{\gamma}$ mapping *Exp*→*Exp* and performing all the substitutions specified by γ:

$$\hat{\gamma}[\![x]\!] = \gamma(x) \quad \text{if } x \in \text{dom}(\gamma)$$
$$\hat{\gamma}[\![x]\!] = x \quad \text{if } x \notin \text{dom}(\gamma)$$
$$\hat{\gamma}[\![n]\!] = n$$
$$\hat{\gamma}[\![e_0\ e_1]\!] = \hat{\gamma}[\![e_0]\!]\ \hat{\gamma}[\![e_1]\!]$$
$$\hat{\gamma}[\![\lambda\, x : \tau\ .\ e]\!] = \lambda\, x : \tau\ .\ \hat{\gamma'}[\![e]\!]$$

γ' is identical to γ 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:

$$\Gamma \vdash e : \tau \Rightarrow \forall \gamma \models \Gamma\ .\ \hat{\gamma}[\![e]\!] \in T_\tau$$

- Now consider $\Gamma = \varnothing$. The only γ satisfying this type context is the identity mapping. Therefore, our refined goal becomes our original goal.
- Now we turn the inductive crank.

Part I

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

Part II

- To show: $\Gamma \vdash (\lambda\, x : \tau\ .\ e) : \tau \rightarrow \tau'$
  $\Rightarrow \forall \gamma \models \Gamma\ .\ \hat{\gamma}[\![(\lambda\, x : \tau\ .\ e)\ ]\!] \in T_{\tau \rightarrow \tau'}$
- Assume LHS, consider arbitrary $\gamma \models \Gamma$
- Recall $T_{\tau \rightarrow \tau'} = \{\, e \mid \vdash e : \tau \rightarrow \tau' \wedge e \Downarrow v \wedge (\forall e' \in T_\tau\ .\ (e\ e') \in T_{\tau'})\}$
- $\hat{\gamma}[\![(\lambda\, x : \tau\ .\ e)\ ]\!]$ is already a value so $e \Downarrow v$
- Need $(\forall e' \in T_\tau\ .\ (\hat{\gamma}[\![(\lambda\, x : \tau\ .\ e)\ ]\!]\ e') \in T_{\tau'})\}$
- $\hat{\gamma}[\![(\lambda\, x : \tau\ .\ e)\ ]\!]\ e' = \hat{\gamma'}[\![e]\!]\ [e'/x] = \hat{\gamma''}[\![e]\!]$
  where $\gamma'' = \gamma\,[x \mapsto e']$
- From proof rule: $\Gamma[x \mapsto \tau] \vdash e : \tau'$
- Applying induction hypothesis:
  $\gamma'' \models \Gamma[x \mapsto \tau] \Rightarrow \hat{\gamma''}(e) \in T_{\tau'}$ $\quad \gamma'' \models \Gamma[x \mapsto \tau]$ $\ (e' \in T_\tau)$

QED

Coming soon: richer types

- Recursive types
- Polymorphic types
- Subtyping
- Objects