CIS 500

Software Foundations

Fall 2005

26 October, 2005

Simply-Typed Lambda Calculus

Last time

Last time we talked about proving type soundness for the simply-typed lambda calculus with booleans. This involved showing:

- Canonical Forms
- Inversion Lemmas
- Progress Theorem
- Preservation Theorem

Preservation

Theorem: If $\vdash \texttt{t} : \texttt{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\vdash \texttt{t}' : \texttt{T}$.

We got to the point where we needed a substitution lemma to show the preservation theorem:

If $\texttt{x}{:}\texttt{S} \vdash \texttt{t} : \texttt{T}$ and $\vdash \texttt{s} : \texttt{S}$, then $\vdash [\texttt{x} \mapsto \texttt{s}]\texttt{t} : \texttt{T}$.

Typing rules

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-If)}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-App)}$$

Strengthened induction hypothesis

Lemma: Types are preserved under substitition.

If $\Gamma, x{:}S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.

Lemmas about the context

Lemma [Permutation]: If $\Gamma \vdash t : T$ and Δ is a permutation of Γ then $\Delta \vdash t : T$.

Lemma [Weakening]: If $\Gamma \vdash t : T$ and $x \notin dom(\Gamma)$ then $\Gamma, x : S \vdash t : T$.

Important points about type soundness for STLC

- Typing rules need a context to deal with variables.
- Progress lemma must have an empty context to be true.
- Preservation lemma need not (see TAPL).
- Application case of preservation lemma needs substitution lemma.
- Substitution lemma must be strenthened to non-empty contexts.
- Substitution lemma cannot be proved by induction on typing derivation.

On to real programming languages...

The Unit type

t	::=	...	*terms*
		unit	*constant* unit
v	::=	...	*values*
		unit	*constant* *unit*
T	::=	...	*types*
		Unit	*unit type*

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \qquad \text{(T-Unit)}$$

Sequencing

t	::=	...	*terms*
		$t_1 ; t_2$	

Sequencing

t	::=	...	*terms*
		$t_1 ; t_2$	

$$\frac{t_1 \longrightarrow t_1'}{t_1 ; t_2 \longrightarrow t_1' ; t_2} \qquad \text{(E-Seq)}$$

$$\texttt{unit} ; t_2 \longrightarrow t_2 \qquad \text{(E-SeqNext)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 ; t_2 : T_2} \qquad \text{(T-Seq)}$$

Derived forms

One way to extend a programming language is through derived forms.

A derived form is a new syntax, operational semantics $t \longrightarrow_E t'$ and type system $\Gamma \vdash^E t : T$ for a language such that there is a function e that translates the new language into the old language with the following properties:

- $t \longrightarrow_E t'$ iff $e(t) \longrightarrow e(t')$.
- $\Gamma \vdash t : T$ iff $\Gamma \vdash e(t) : T$.

Derived forms are also called syntactic sugar.

Sequencing as a derived form

$$t_1 ; t_2 \stackrel{\text{def}}{=} (\lambda x{:}\text{Unit}.t_2)\ t_1 \quad \text{where } x \notin FV(t_2)$$

Equivalence of the two definitions

[board]

Let-bindings

New syntactic forms

t	::=	...	*terms*
		$\texttt{let } x{=}t \texttt{ in } t$	*let binding*

New evaluation rules

$t \longrightarrow t'$

$$\texttt{let } x{=}v_1 \texttt{ in } t_2 \longrightarrow [x \mapsto v_1]t_2 \qquad \text{(E-LetV)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{let } x{=}t_1 \texttt{ in } t_2 \longrightarrow \texttt{let } x{=}t_1' \texttt{ in } t_2} \qquad \text{(E-Let)}$$

New typing rules

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 : T_2} \qquad \text{(T-Let)}$$

Pairs

t	::=	...	*terms*
		$\{t,t\}$	*pair*
		$t.1$	*first projection*
		$t.2$	*second projection*
v	::=	...	*values*
		$\{v,v\}$	*pair value*
T	::=	...	*types*
		$T_1 \times T_2$	*product type*

Evaluation rules for pairs

$$\{v_1,v_2\}.1 \longrightarrow v_1 \qquad \text{(E-PairBeta1)}$$

$$\{v_1,v_2\}.2 \longrightarrow v_2 \qquad \text{(E-PairBeta2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.1 \longrightarrow t_1'.1} \qquad \text{(E-Proj1)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.2 \longrightarrow t_1'.2} \qquad \text{(E-Proj2)}$$

$$\frac{t_1 \longrightarrow t_1'}{\{t_1,t_2\} \longrightarrow \{t_1',t_2\}} \qquad \text{(E-Pair1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\{v_1,t_2\} \longrightarrow \{v_1,t_2'\}} \qquad \text{(E-Pair2)}$$

Typing rules for pairs

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1,t_2\} : T_1 \times T_2} \qquad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \qquad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \qquad \text{(T-Proj2)}$$

Tuples

t	::=	...	*terms*
		$\{t_i \ ^{i \in 1..n}\}$	*tuple*
		$t.i$	*projection*
v	::=	...	*values*
		$\{v_i \ ^{i \in 1..n}\}$	*tuple value*
T	::=	...	*types*
		$\{T_i \ ^{i \in 1..n}\}$	*tuple type*

Evaluation rules for tuples

$$\{v_i \ ^{i\in 1..n}\}.j \longrightarrow v_j \qquad \text{(E-ProjTuple)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.i \longrightarrow t_1'.i} \qquad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\{v_i \ ^{i\in 1..j-1}, t_j, t_k \ ^{k\in j+1..n}\} \longrightarrow \{v_i \ ^{i\in 1..j-1}, t_j', t_k \ ^{k\in j+1..n}\}} \qquad \text{(E-Tuple)}$$

Typing rules for tuples

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i \ ^{i\in 1..n}\} : \{T_i \ ^{i\in 1..n}\}} \qquad \text{(T-Tuple)}$$

$$\frac{\Gamma \vdash t_1 : \{T_i \ ^{i\in 1..n}\}}{\Gamma \vdash t_1.j : T_j} \qquad \text{(T-Proj)}$$

Records

t	::=	...	*terms*
		$\{l_i=t_i \ ^{i\in 1..n}\}$	*record*
		$t.l$	*projection*
v	::=	...	*values*
		$\{l_i=v_i \ ^{i\in 1..n}\}$	*record value*
T	::=	...	*types*
		$\{l_i:T_i \ ^{i\in 1..n}\}$	*type of records*

Evaluation rules for records

$$\{l_i=v_i \ ^{i\in 1..n}\}.l_j \longrightarrow v_j \qquad \text{(E-ProjRcd)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.l \longrightarrow t_1'.l} \qquad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\{l_i=v_i \ ^{i\in 1..j-1}, l_j=t_j, l_k=t_k \ ^{k\in j+1..n}\} \longrightarrow \{l_i=v_i \ ^{i\in 1..j-1}, l_j=t_j', l_k=t_k \ ^{k\in j+1..n}\}} \qquad \text{(E-Rcd)}$$

Typing rules for records

$$\frac{\text{for each } \mathtt{i} \quad \Gamma \vdash \mathtt{t_i} : \mathtt{T_i}}{\Gamma \vdash \{\mathtt{l_i}=\mathtt{t_i}\ ^{\mathtt{i}\in 1..n}\} : \{\mathtt{l_i}:\mathtt{T_i}\ ^{\mathtt{i}\in 1..n}\}} \quad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \{\mathtt{l_i}:\mathtt{T_i}\ ^{\mathtt{i}\in 1..n}\}}{\Gamma \vdash \mathtt{t_1}.\mathtt{l_j} : \mathtt{T_j}} \quad \text{(T-Proj)}$$

Discussion

Intro vs. elim forms

An introduction form for a given type gives us a way of constructing elements of this type.

An elimination form for a type gives us a way of using elements of this type.

What typing rules are introduction forms? What are elimination forms?

Erasure

$$\begin{aligned} \mathit{erase}(\mathtt{x}) &= \mathtt{x} \\ \mathit{erase}(\lambda\mathtt{x}{:}\mathtt{T_1}.\ \mathtt{t_2}) &= \lambda\mathtt{x}.\ \mathit{erase}(\mathtt{t_2}) \\ \mathit{erase}(\mathtt{t_1}\ \mathtt{t_2}) &= \mathit{erase}(\mathtt{t_1})\ \mathit{erase}(\mathtt{t_2}) \end{aligned}$$

Typability

An untyped λ-term $\mathtt{m}$ is said to be typable if there is some term $\mathtt{t}$ in the simply typed lambda-calculus, some type $\mathtt{T}$, and some context Γ such that $\mathit{erase}(\mathtt{t}) = \mathtt{m}$ and $\Gamma \vdash \mathtt{t} : \mathtt{T}$.

Cf. type reconstruction in OCaml.