Foundations of Software Winter Semester 2007

Week 6 October 23

October 23, 2007 - version 1.0

Plan

PREVIOUSLY:

- 1. type safety as *progress* and *preservation*
- 2. typed arithmetic expressions
- 3. simply typed lambda calculus (STLC)

TODAY:

- 1. Equivalence of lambda terms
- 2. Preservation for STLC
- 3. Extensions to STLC

NEXT: state, exceptions NEXT: polymorphic (not so simple) typing

Equivalence of Lambda Terms

Representing Numbers

We have seen how certain terms in the lambda-calculus can be used to represent natural numbers.

 $c_0 = \lambda s. \lambda z. z$ $c_1 = \lambda s. \lambda z. s z$ $c_2 = \lambda s. \lambda z. s (s z)$ $c_3 = \lambda s. \lambda z. s (s (s z))$

Other lambda-terms represent common operations on numbers:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Representing Numbers

We have seen how certain terms in the lambda-calculus can be used to represent natural numbers.

 $c_{0} = \lambda s. \lambda z. z$ $c_{1} = \lambda s. \lambda z. s z$ $c_{2} = \lambda s. \lambda z. s (s z)$ $c_{3} = \lambda s. \lambda z. s (s (s z))$

Other lambda-terms represent common operations on numbers:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

In what sense can we say this representation is "correct"? In particular, on what basis can we argue that scc on church numerals corresponds to ordinary successor on numbers?

The naive approach

One possibility:

For each *n*, the term scc c_n evaluates to c_{n+1} .

The naive approach... doesn't work

One possibility:

For each *n*, the term scc c_n evaluates to c_{n+1} . Unfortunately, this is false. E.g.:

 $scc c_2 = (\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s. \lambda z. s (s z))$ $\longrightarrow \lambda s. \lambda z. s ((\lambda s. \lambda z. s (s z)) s z)$ $\neq \lambda s. \lambda z. s (s (s z))$ $= c_3$

A better approach

Recall the intuition behind the church numeral representation:

- a number n is represented as a term that "does something n times to something else"
- scc takes a term that "does something *n* times to something else" and returns a term that "does something *n* + 1 times to something else"

I.e., what we really care about is that $scc c_2$ behaves the same as c_3 when applied to two arguments.

$$scc c_2 v w = (\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s. \lambda z. s (s z)) v w$$
$$\longrightarrow (\lambda s. \lambda z. s ((\lambda s. \lambda z. s (s z)) s z)) v w$$
$$\longrightarrow (\lambda z. v ((\lambda s. \lambda z. s (s z)) v z)) w$$
$$\longrightarrow v ((\lambda s. \lambda z. s (s z)) v w)$$
$$\longrightarrow v ((\lambda z. v (v z)) w)$$
$$\longrightarrow v (v (v w))$$

$$c_3 \lor w = (\lambda s. \lambda z. s (s (s z))) \lor w$$
$$\longrightarrow (\lambda z. \lor (\lor (\lor z))) w$$
$$\longrightarrow \lor (\lor (\lor w)))$$

A general question

We have argued that, although $scc c_2$ and c_3 do not evaluate to the same thing, they are nevertheless "behaviorally equivalent."

What, precisely, does behavioral equivalence mean?

Intuition

Roughly,

"terms ${\bf s}$ and ${\bf t}$ are behaviorally equivalent"

should mean:

"there is no 'test' that distinguishes ${\tt s}$ and ${\tt t}$ — i.e., no way to put them in the same context and observe different results."

Intuition

Roughly,

"terms ${\bf s}$ and ${\bf t}$ are behaviorally equivalent" should mean:

"there is no 'test' that distinguishes ${\tt s}$ and ${\tt t}$ — i.e., no way to put them in the same context and observe different results."

To make this precise, we need to be clear what we mean by a *testing context* and how we are going to *observe* the results of a test.

Examples

```
tru = \lambda t. \lambda f. t
tru' = \lambda t. \lambda f. (\lambda x.x) t
fls = \lambda t. \lambda f. f
omega = (\lambda x. x x) (\lambda x. x x)
poisonpill = \lambda x. omega
placebo = \lambda x. tru
Y_f = (\lambda x. f (x x)) (\lambda x. f (x x))
```

Which of these are behaviorally equivalent?

Observational equivalence

As a first step toward defining behavioral equivalence, we can use the notion of *normalizability* to define a simple notion of *test*.

Two terms s and t are said to be *observationally equivalent* if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both diverge.

l.e., we "observe" a term's behavior simply by running it and seeing if it halts.

Observational equivalence

As a first step toward defining behavioral equivalence, we can use the notion of *normalizability* to define a simple notion of *test*.

Two terms s and t are said to be *observationally equivalent* if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both diverge.

l.e., we "observe" a term's behavior simply by running it and seeing if it halts.

Aside:

Is observational equivalence a decidable property?

Observational equivalence

As a first step toward defining behavioral equivalence, we can use the notion of *normalizability* to define a simple notion of *test*.

Two terms s and t are said to be *observationally equivalent* if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both diverge.

l.e., we "observe" a term's behavior simply by running it and seeing if it halts.

Aside:

- Is observational equivalence a decidable property?
- Does this mean the definition is ill-formed?

Examples

omega and tru are not observationally equivalent

Examples

- omega and tru are not observationally equivalent
- tru and fls are observationally equivalent

Behavioral Equivalence

This primitive notion of observation now gives us a way of "testing" terms for behavioral equivalence

Terms s and t are said to be *behaviorally equivalent* if, for every finite sequence of values v_1 , v_2 , ..., v_n , the applications

 $s v_1 v_2 \ldots v_n$

and

```
t v_1 v_2 \ldots v_n
```

are observationally equivalent.

Examples

These terms are behaviorally equivalent:

tru = $\lambda t. \lambda f. t$ tru' = $\lambda t. \lambda f. (\lambda x.x) t$

So are these:

omega = $(\lambda x. x x) (\lambda x. x x)$ $Y_f = (\lambda x. f (x x)) (\lambda x. f (x x))$

These are not behaviorally equivalent (to each other, or to any of the terms above):

```
fls = \lambda t. \lambda f. f
poisonpill = \lambda x. omega
placebo = \lambda x. tru
```

Given terms s and t, how do we *prove* that they are (or are not) behaviorally equivalent?

To prove that s and t are *not* behaviorally equivalent, it suffices to find a sequence of values $v_1 \dots v_n$ such that one of

 $s v_1 v_2 \ldots v_n$

and

t $v_1 v_2 \ldots v_n$

diverges, while the other reaches a normal form.

Example:

the single argument unit demonstrates that fls is not behaviorally equivalent to poisonpill:

 $\begin{array}{c} \text{fls unit} \\ = (\lambda t. \ \lambda f. \ f) \ \text{unit} \\ \xrightarrow{}^* \lambda f. \ f \end{array}$

poisonpill unit diverges

Example:

the argument sequence (λx. x) poisonpill (λx. x) demonstrate that tru is not behaviorally equivalent to fls:

$$ext{tru} (\lambda \mathbf{x}. \mathbf{x}) ext{ poisonpill } (\lambda \mathbf{x}. \mathbf{x}) \ \longrightarrow^* (\lambda \mathbf{x}. \mathbf{x}) (\lambda \mathbf{x}. \mathbf{x}) \ \longrightarrow^* \lambda \mathbf{x}. \mathbf{x}$$

fls $(\lambda x. x)$ poisonpill $(\lambda x. x)$ \longrightarrow^* poisonpill $(\lambda x. x)$, which diverges

To prove that s and t *are* behaviorally equivalent, we have to work harder: we must show that, for *every* sequence of values $v_1 \dots v_n$, either both

 $s v_1 v_2 \ldots v_n$

and

t $v_1 v_2 \ldots v_n$

diverge, or else both reach a normal form.

How can we do this?

In general, such proofs require some additional machinery that we will not have time to get into in this course (so-called *applicative bisimulation*). But, in some cases, we can find simple proofs. *Theorem:* These terms are behaviorally equivalent:

tru = $\lambda t. \lambda f. t$ tru' = $\lambda t. \lambda f. (\lambda x.x) t$

Proof: Consider an arbitrary sequence of values $v_1 \dots v_n$.

- For the case where the sequence has just one element (i.e., n = 1), note that both tru v₁ and tru' v₁ reach normal forms after one reduction step.
- ► For the case where the sequence has more than one element (i.e., n > 1), note that both tru v₁ v₂ v₃ ... v_n and tru' v₁ v₂ v₃ ... v_n reduce (in two steps) to v₁ v₃ ... v_n. So either both normalize or both diverge.

Theorem: These terms are behaviorally equivalent:

omega = $(\lambda x. x x) (\lambda x. x x)$ $Y_f = (\lambda x. f (x x)) (\lambda x. f (x x))$

Proof: Both

omega $v_1 \ldots v_n$

and

 $Y_f v_1 \dots v_n$

diverge, for every sequence of arguments $v_1 \dots v_n$.

Theorem: If $\Gamma \vdash t$: T and t \longrightarrow t', then $\Gamma \vdash t'$: T.

Proof: By induction

Theorem: If $\Gamma \vdash t$: T and t \longrightarrow t', then $\Gamma \vdash t'$: T.

Proof: By induction on typing derivations.

Which case is the hard one??

Theorem: If $\Gamma \vdash t$: T and t \longrightarrow t', then $\Gamma \vdash t'$: T.

Theorem: If $\Gamma \vdash t$: T and t \longrightarrow t', then $\Gamma \vdash t'$: T.

By the inversion lemma for evaluation, there are three subcases...

Theorem: If $\Gamma \vdash t$: T and t \longrightarrow t', then $\Gamma \vdash t'$: T.

Theorem: If $\Gamma \vdash t$: T and t \longrightarrow t', then $\Gamma \vdash t'$: T.

Proof: By induction on typing derivations. Case T-APP: Given $t = t_1 t_2$ $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$ $\Gamma \vdash t_2 : T_{11}$ $T = T_{12}$ Show $\Gamma \vdash t' : T_{12}$ By the inversion lemma for evaluation, there are three subcases... Subcase: $t_1 = \lambda x: T_{11}$. t_{12} t_2 a value v_2 $\mathsf{t}' = [\mathsf{x} \mapsto \mathsf{v}_2] \mathsf{t}_{12}$

Uh oh.

The "Substitution Lemma"

Lemma: Types are preserved under substitution.

That is, if Γ , $x: S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.

The "Substitution Lemma"

Lemma: Types are preserved under substitution.

That is, if Γ , $x: S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.

Proof: ...

Weakening and Permutation

Two other lemmas will be useful.

Weakening tells us that we can *add assumptions* to the context without losing any true typing statements.

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

Weakening and Permutation

Two other lemmas will be useful.

Weakening tells us that we can *add assumptions* to the context without losing any true typing statements.

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

Permutation tells us that the order of assumptions in (the list) $\mbox{\sc F}$ does not matter.

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

Weakening and Permutation

Two other lemmas will be useful.

Weakening tells us that we can *add assumptions* to the context without losing any true typing statements.

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

Moreover, the latter derivation has the same depth as the former.

Permutation tells us that the order of assumptions in (the list) $\mbox{\sc F}$ does not matter.

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

Moreover, the latter derivation has the same depth as the former.

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

I.e., "Types are preserved under substitition."

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

Proof: By induction on the derivation of Γ , $x:S \vdash t : T$. Proceed by cases on the final typing rule used in the derivation.

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

Proof: By induction on the derivation of Γ , $x:S \vdash t : T$. Proceed by cases on the final typing rule used in the derivation.

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

Proof: By induction on the derivation of Γ , $x:S \vdash t : T$. Proceed by cases on the final typing rule used in the derivation.

Case T-APP:
$$t = t_1 t_2$$

 $\Gamma, x: S \vdash t_1 : T_2 \rightarrow T_1$
 $\Gamma, x: S \vdash t_2 : T_2$
 $T = T_1$

By the induction hypothesis, $\Gamma \vdash [x \mapsto s]t_1 : T_2 \rightarrow T_1$ and $\Gamma \vdash [x \mapsto s]t_2 : T_2$. By T-APP, $\Gamma \vdash [x \mapsto s]t_1 \ [x \mapsto s]t_2 : T$, i.e., $\Gamma \vdash [x \mapsto s](t_1 \ t_2) : T$.

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

Proof: By induction on the derivation of Γ , $x:S \vdash t : T$. Proceed by cases on the final typing rule used in the derivation.

Case T-VAR: t = z

with $z:T \in (\Gamma, x:S)$

There are two sub-cases to consider, depending on whether z is x or another variable. If z = x, then $[x \mapsto s]z = s$. The required result is then $\Gamma \vdash s : S$, which is among the assumptions of the lemma. Otherwise, $[x \mapsto s]z = z$, and the desired result is immediate.

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

Proof: By induction on the derivation of Γ , $x:S \vdash t : T$. Proceed by cases on the final typing rule used in the derivation.

By our conventions on choice of bound variable names, we may assume $x \neq y$ and $y \notin FV(s)$. Using *permutation* on the given subderivation, we obtain Γ , $y:T_2, x:S \vdash t_1 : T_1$. Using *weakening* on the other given derivation ($\Gamma \vdash s : S$), we obtain Γ , $y:T_2 \vdash s : S$. Now, by the induction hypothesis, Γ , $y:T_2 \vdash [x \mapsto s]t_1 : T_1$. By T-ABS, $\Gamma \vdash \lambda y:T_2$. $[x \mapsto s]t_1 : T_2 \rightarrow T_1$, i.e. (by the definition of substitution), $\Gamma \vdash [x \mapsto s]\lambda y:T_2$. $t_1 : T_2 \rightarrow T_1$.

Summary: Preservation

Theorem: If $\Gamma \vdash t$: T and t \longrightarrow t', then $\Gamma \vdash t'$: T.

Lemmas to prove:

- Weakening
- Permutation
- Substitution preserves types
- Reduction preserves types (i.e., preservation)

Review: Type Systems

To define and verify a type system, you must:

- 1. Define types
- 2. Specify typing rules
- 3. Prove soundness: progress and preservation

Two Typing Topics

Erasure

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.

The Curry-Howard Correspondence

In constructive logics, a proof of P must provide evidence for P.

• "law of the excluded middle" — $P \lor \neg P$ — not recognized.

A proof of $P \land Q$ is a *pair* of evidence for P and evidence for Q.

A proof of $P \supset Q$ is a *procedure* for transforming evidence for P into evidence for Q.

Propositions as Types

Logic	Programming languages
propositions	types
proposition $P \supset Q$	type P→Q
proposition $P \land Q$	type $\mathbb{P} imes \mathbb{Q}$
proof of proposition <i>P</i>	term t of type P
proposition <i>P</i> is provable	type P is inhabited (by some term)
	evaluation

Propositions as Types

Logic	Programming languages
propositions	types
proposition $P \supset Q$	type P→Q
proposition $P \land Q$	type $P imes Q$
proof of proposition <i>P</i>	term t of type P
proposition <i>P</i> is provable	type P is inhabited (by some term)
proof simplification	evaluation
(a.k.a. "cut elimination")	

Extensions to STLC

Base types

Up to now, we've formulated "base types" (e.g. Nat) by adding them to the syntax of types, extending the syntax of terms with associated constants (zero) and operators (succ, etc.) and adding appropriate typing and evaluation rules. We can do this for as many base types as we like.

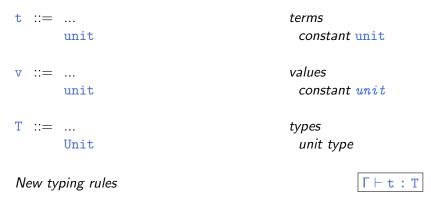
For more theoretical discussions (as opposed to programming) we can often ignore the term-level inhabitants of base types, and just treat these types as uninterpreted constants.

E.g., suppose B and C are some base types. Then we can ask (without knowing anything more about B or C) whether there are any types S and T such that the term

 $(\lambda f:S. \lambda g:T. f g) (\lambda x:B. x)$

is well typed.

The Unit type



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

Sequencing

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

terms

Sequencing

 $\begin{array}{c} \begin{array}{c} t_1 \longrightarrow t_1' \\ \hline t_1; t_2 \longrightarrow t_1'; t_2 \end{array} & (E-SEQ) \\ \\ \begin{array}{c} \text{unit}; t_2 \longrightarrow t_2 \end{array} & (E-SEQNEXT) \end{array} \\ \\ \end{array} \\ \\ \hline \hline \Gamma \vdash t_1: \begin{array}{c} \text{Unit} \quad \Gamma \vdash t_2: T_2 \\ \hline \Gamma \vdash t_1; t_2: T_2 \end{array} & (T-SEQ) \end{array}$

terms

Derived forms

- Syntatic sugar
- Internal language vs. external (surface) language

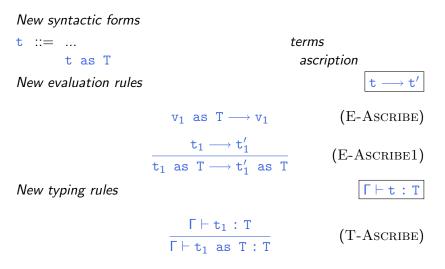
Sequencing as a derived form

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

Equivalence of the two definitions

[board]

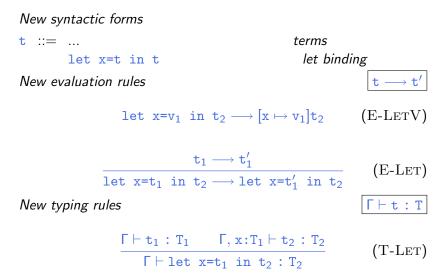
Ascription



Ascription as a derived form

t as $T \stackrel{\text{def}}{=} (\lambda x:T. x)$ t

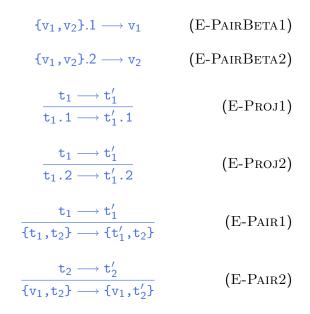
Let-bindings



Pairs

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

Evaluation rules for pairs



Typing rules for pairs

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

 $\frac{\Gamma \vdash \mathtt{t}_1 \,:\, \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1 . 1 \,:\, \mathtt{T}_{11}}$

 $\frac{\Gamma \vdash \mathtt{t}_1 \, : \, \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.2 \, : \, \mathtt{T}_{12}}$

(T-PROJ2)

Tuples

t ::= $\{t_i^{i \in 1..n}\}$ t.i

 $\mathbf{v} ::= \dots \\ \{\mathbf{v}_i \ ^{i \in 1 \dots n}\}$

 $T ::= \dots \\ \{T_i^{i \in 1 \dots n}\}$

terms tuple projection

values tuple value

types tuple type

Evaluation rules for tuples

$$\{\mathbf{v}_i^{i\in 1..n}\}, \mathbf{j} \longrightarrow \mathbf{v}_j \qquad (\text{E-ProjTuple})$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}'_1}{\mathtt{t}_1.\mathtt{i} \longrightarrow \mathtt{t}'_1.\mathtt{i}}$$
(E-Proj)

$$\frac{ \mathsf{t}_{j} \longrightarrow \mathsf{t}'_{j} }{ \{ \mathsf{v}_{i} \stackrel{i \in 1..j-1}{,} \mathsf{t}_{j}, \mathsf{t}_{k} \stackrel{k \in j+1..n}{,} } \\ \longrightarrow \{ \mathsf{v}_{i} \stackrel{i \in 1..j-1}{,} \mathsf{t}'_{j}, \mathsf{t}_{k} \stackrel{k \in j+1..n}{,} \}$$

Typing rules for tuples

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{T}_i \ ^{i \in 1..n}\}}{\Gamma \vdash \mathtt{t}_1 . \mathtt{j} : \mathtt{T}_j}$$