Foundations of Software
Winter Semester 2007

Week 6

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.

$$
\begin{aligned}
c_0 &= \lambda s.\ \lambda z.\ z \\
c_1 &= \lambda s.\ \lambda z.\ s\ z \\
c_2 &= \lambda s.\ \lambda z.\ s\ (s\ z) \\
c_3 &= \lambda s.\ \lambda z.\ s\ (s\ (s\ z))
\end{aligned}
$$

Other lambda-terms represent common operations on numbers:

$$\mathtt{scc} = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$$

Representing Numbers

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

$$
\begin{aligned}
\mathtt{c}_0 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{z} \\
\mathtt{c}_1 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ z} \\
\mathtt{c}_2 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (s\ z)} \\
\mathtt{c}_3 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (s\ (s\ z))}
\end{aligned}
$$

Other lambda-terms represent common operations on numbers:

$$\mathtt{scc} = \lambda\mathtt{n}.\ \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (n\ s\ z)}$$

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 $\mathtt{scc}\ \mathtt{c}_n$ evaluates to $\mathtt{c}_{n+1}$.

The naive approach... doesn't work

One possibility:

For each n, the term $\mathtt{scc}\ \mathtt{c}_n$ evaluates to $\mathtt{c}_{n+1}$.

Unfortunately, this is false.
E.g.:

$$
\begin{aligned}
\mathtt{scc}\ \mathtt{c}_2 &= (\lambda\mathtt{n}.\ \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (n\ s\ z)})\ (\lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (s\ z)}) \\
&\longrightarrow \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s}\ ((\lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (s\ z)})\ \mathtt{s\ z}) \\
&\neq \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (s\ (s\ z))} \\
&= \mathtt{c}_3
\end{aligned}
$$

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 $\mathtt{scc}\ \mathtt{c}_2$ behaves the same as $\mathtt{c}_3$ when applied to two arguments.

$$
\begin{aligned}
\texttt{scc}\ \texttt{c}_2\ \texttt{v}\ \texttt{w} &= (\lambda\texttt{n}.\ \lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s}\ (\texttt{n}\ \texttt{s}\ \texttt{z}))\ (\lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s}\ (\texttt{s}\ \texttt{z}))\ \texttt{v}\ \texttt{w} \\
&\longrightarrow (\lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s}\ ((\lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s}\ (\texttt{s}\ \texttt{z}))\ \texttt{s}\ \texttt{z}))\ \texttt{v}\ \texttt{w} \\
&\longrightarrow (\lambda\texttt{z}.\ \texttt{v}\ ((\lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s}\ (\texttt{s}\ \texttt{z}))\ \texttt{v}\ \texttt{z}))\ \texttt{w} \\
&\longrightarrow \texttt{v}\ ((\lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s}\ (\texttt{s}\ \texttt{z}))\ \texttt{v}\ \texttt{w}) \\
&\longrightarrow \texttt{v}\ ((\lambda\texttt{z}.\ \texttt{v}\ (\texttt{v}\ \texttt{z}))\ \texttt{w}) \\
&\longrightarrow \texttt{v}\ (\texttt{v}\ (\texttt{v}\ \texttt{w}))
\end{aligned}
$$

$$
\begin{aligned}
\texttt{c}_3\ \texttt{v}\ \texttt{w} &= (\lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s}\ (\texttt{s}\ (\texttt{s}\ \texttt{z})))\ \texttt{v}\ \texttt{w} \\
&\longrightarrow (\lambda\texttt{z}.\ \texttt{v}\ (\texttt{v}\ (\texttt{v}\ \texttt{z})))\ \texttt{w} \\
&\longrightarrow \texttt{v}\ (\texttt{v}\ (\texttt{v}\ \texttt{w}))
\end{aligned}
$$

A general question

We have argued that, although `scc` $\texttt{c}_2$ and $\texttt{c}_3$ do not evaluate to the same thing, they are nevertheless "behaviorally equivalent."

What, precisely, does behavioral equivalence mean?

Intuition

Roughly,

> "terms `s` and `t` are behaviorally equivalent"

should mean:

> "there is no 'test' that distinguishes `s` and `t` — i.e., no way to put them in the same context and observe different results."

Intuition

Roughly,

> "terms `s` and `t` are behaviorally equivalent"

should mean:

> "there is no 'test' that distinguishes `s` and `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 = λt. λf. t
tru' = λt. λf. (λx.x) t
fls = λt. λf. f
omega = (λx. x x) (λx. x x)
poisonpill = λx. omega
placebo = λx. tru
Y_f = (λx. f (x x)) (λx. 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.

I.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.

I.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.

I.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 $v_1, v_2, \ldots, v_n$, 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 = λt. λf. t
tru' = λt. λf. (λx.x) t
```

So are these:

```
omega = (λx. x x) (λx. x x)
Y_f = (λx. f (x x)) (λx. f (x x))
```

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

```
fls = λt. λf. f
poisonpill = λx. omega
placebo = λx. tru
```

Proving behavioral equivalence

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

Proving behavioral inequivalence

To prove that `s` and `t` are *not* behaviorally equivalent, it suffices to find a sequence of values $\mathtt{v}_1 \ldots \mathtt{v}_n$ such that one of

$$\mathtt{s}\ \mathtt{v}_1\ \mathtt{v}_2\ \ldots\ \mathtt{v}_n$$

and

$$\mathtt{t}\ \mathtt{v}_1\ \mathtt{v}_2\ \ldots\ \mathtt{v}_n$$

diverges, while the other reaches a normal form.

Proving behavioral inequivalence

Example:

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

$$\begin{aligned} &\mathtt{fls\ unit} \\ =\ &(\lambda\mathtt{t}.\ \lambda\mathtt{f}.\ \mathtt{f})\ \mathtt{unit} \\ \longrightarrow^*\ &\lambda\mathtt{f}.\ \mathtt{f} \end{aligned}$$

$$\mathtt{poisonpill\ unit}$$

diverges

Proving behavioral inequivalence

Example:

- the argument sequence $(\lambda\mathtt{x}.\ \mathtt{x})$ `poisonpill` $(\lambda\mathtt{x}.\ \mathtt{x})$ demonstrate that `tru` is not behaviorally equivalent to `fls`:

$$\begin{aligned} &\mathtt{tru}\ (\lambda\mathtt{x}.\ \mathtt{x})\ \mathtt{poisonpill}\ (\lambda\mathtt{x}.\ \mathtt{x}) \\ \longrightarrow^*\ &(\lambda\mathtt{x}.\ \mathtt{x})(\lambda\mathtt{x}.\ \mathtt{x}) \\ \longrightarrow^*\ &\lambda\mathtt{x}.\ \mathtt{x} \end{aligned}$$

$$\begin{aligned} &\mathtt{fls}\ (\lambda\mathtt{x}.\ \mathtt{x})\ \mathtt{poisonpill}\ (\lambda\mathtt{x}.\ \mathtt{x}) \\ \longrightarrow^*\ &\mathtt{poisonpill}\ (\lambda\mathtt{x}.\ \mathtt{x}), \text{ which diverges} \end{aligned}$$

Proving behavioral equivalence

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

Proving behavioral equivalence

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 = λt. λf. t
tru' = λt. λf. (λx.x) t
```

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

- For the case where the sequence has just one element (i.e., $n = 1$), note that both tru v_1 and tru′ v_1 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 v_1 v_2 v_3 ... v_n and tru′ v_1 v_2 v_3 ... v_n reduce (in two steps) to v_1 v_3 ... v_n. So either both normalize or both diverge.

Proving behavioral equivalence

Theorem: These terms are behaviorally equivalent:

```
omega = (λx. x x) (λx. x x)
Yf = (λx. f (x x)) (λx. f (x x))
```

Proof: Both

$$\text{omega}\ v_1 \ldots v_n$$

and

$$Y_f\ v_1 \ldots v_n$$

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

Preservation for STLC

Preservation for STLC

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

Proof: By induction

Preservation for STLC

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

Proof: By induction on typing derivations.

Which case is the hard one??

Preservation for STLC

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

Proof: By induction on typing derivations.

Case T-APP: Given $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$

$\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$

$\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$

$\mathtt{T} = \mathtt{T}_{12}$

Show $\Gamma \vdash \mathtt{t}' : \mathtt{T}_{12}$

Preservation for STLC

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

Proof: By induction on typing derivations.

Case T-APP: Given $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$

$\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$

$\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$

$\mathtt{T} = \mathtt{T}_{12}$

Show $\Gamma \vdash \mathtt{t}' : \mathtt{T}_{12}$

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

Preservation for STLC

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 $t = t_1\ t_2$
$\Gamma \vdash t_1 : T_{11} \to T_{12}$
$\Gamma \vdash t_2 : T_{11}$
$T = T_{12}$
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
$t' = [x \mapsto v_2]t_{12}$

Preservation for STLC

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 $t = t_1\ t_2$
$\Gamma \vdash t_1 : T_{11} \to T_{12}$
$\Gamma \vdash t_2 : T_{11}$
$T = T_{12}$
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
$t' = [x \mapsto v_2]t_{12}$

Uh oh.

The "Substitution Lemma"

Lemma: Types are preserved under substitition.

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

The "Substitution Lemma"

Lemma: Types are preserved under substitition.

That is, if $\Gamma, x{:}S \vdash t : T$ 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 $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.

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 $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.

Permutation tells us that the order of assumptions in (the list) Γ does not matter.

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

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 $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.

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

Permutation tells us that the order of assumptions in (the list) Γ does not matter.

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

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

The “Substitution Lemma”

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

I.e., “Types are preserved under substitition.”

The "Substitution Lemma"

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

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

The "Substitution Lemma"

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

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

The "Substitution Lemma"

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

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

Case T-APP:

$$\begin{aligned} &\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2 \\ &\Gamma, \mathtt{x{:}S} \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1 \\ &\Gamma, \mathtt{x{:}S} \vdash \mathtt{t}_2 : \mathtt{T}_2 \\ &\mathtt{T} = \mathtt{T}_1 \end{aligned}$$

By the induction hypothesis, $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1$ and $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 : \mathtt{T}_2$. By T-APP, $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 : \mathtt{T}$, i.e., $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) : \mathtt{T}$.

The "Substitution Lemma"

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

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

Case T-VAR:

$$\begin{aligned} &\mathtt{t} = \mathtt{z} \\ &\text{with } \mathtt{z{:}T} \in (\Gamma, \mathtt{x{:}S}) \end{aligned}$$

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

The "Substitution Lemma"

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

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

Case T-Abs: $\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T}_2.\mathtt{t}_1 \qquad \mathtt{T} = \mathtt{T}_2 {\to} \mathtt{T}_1$

$\Gamma, \mathtt{x}{:}\mathtt{S}, \mathtt{y}{:}\mathtt{T}_2 \vdash \mathtt{t}_1 : \mathtt{T}_1$

By our conventions on choice of bound variable names, we may assume $\mathtt{x} \neq \mathtt{y}$ and $\mathtt{y} \notin FV(\mathtt{s})$. Using *permutation* on the given subderivation, we obtain $\Gamma, \mathtt{y}{:}\mathtt{T}_2, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_1 : \mathtt{T}_1$. Using *weakening* on the other given derivation ($\Gamma \vdash \mathtt{s} : \mathtt{S}$), we obtain $\Gamma, \mathtt{y}{:}\mathtt{T}_2 \vdash \mathtt{s} : \mathtt{S}$. Now, by the induction hypothesis, $\Gamma, \mathtt{y}{:}\mathtt{T}_2 \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_1$. By T-Abs, $\Gamma \vdash \lambda\mathtt{y}{:}\mathtt{T}_2.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_2 {\to} \mathtt{T}_1$, i.e. (by the definition of substitution), $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\lambda\mathtt{y}{:}\mathtt{T}_2.\ \mathtt{t}_1 : \mathtt{T}_2 {\to} \mathtt{T}_1$.

Summary: Preservation

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

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

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

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" — $P \vee \neg P$ — not recognized.

A proof of $P \wedge Q$ 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 $\mathtt{P} \rightarrow \mathtt{Q}$
proposition $P \wedge Q$	type $\mathtt{P} \times \mathtt{Q}$
proof of proposition P	term $\mathtt{t}$ of type $\mathtt{P}$
proposition P is provable	type $\mathtt{P}$ is inhabited (by some term)
	evaluation

Propositions as Types

Logic	Programming languages
propositions	types
proposition $P \supset Q$	type `P→Q`
proposition $P \wedge Q$	type `P × Q`
proof of proposition P	term `t` of type `P`
proposition P is provable	type `P` is inhabited (by some term)
proof simplification (a.k.a. "cut elimination")	evaluation

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

```
(λf:S. λg:T. f g) (λx:B. x)
```

is well typed.

The `Unit` type

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

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

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

Sequencing

$$\mathtt{t} ::= \ldots \qquad \textit{terms}$$
$$\quad \mathtt{t_1;t_2}$$

Sequencing

$$\mathtt{t} ::= \ldots \qquad \textit{terms}$$
$$\quad \mathtt{t_1;t_2}$$

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

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

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

Derived forms

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

Sequencing as a derived form

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

Equivalence of the two definitions

[board]

Ascription

New syntactic forms

$\mathtt{t}$	::=	...	*terms*
		$\mathtt{t\ as\ T}$	*ascription*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{v_1\ as\ T} \longrightarrow \mathtt{v_1} \qquad \text{(E-Ascribe)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{t_1\ as\ T} \longrightarrow \mathtt{t_1'\ as\ T}} \qquad \text{(E-Ascribe1)}$$

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

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T}}{\Gamma \vdash \mathtt{t_1\ as\ T} : \mathtt{T}} \qquad \text{(T-Ascribe)}$$

Ascription as a derived form

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

Let-bindings

New syntactic forms

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

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

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

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

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

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

Pairs

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

Evaluation rules for pairs

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

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

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

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

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

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

Typing rules for pairs

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

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

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

Tuples

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

Evaluation rules for tuples

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

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

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{v}_i{}^{\,i\in 1..j-1},\mathtt{t}_j,\mathtt{t}_k{}^{\,k\in j+1..n}\}\\ \longrightarrow \{\mathtt{v}_i{}^{\,i\in 1..j-1},\mathtt{t}_j',\mathtt{t}_k{}^{\,k\in j+1..n}\}\end{array}} \qquad \text{(E-Tuple)}$$

Typing rules for tuples

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{T}_i{}^{\,i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{j} : \mathtt{T}_j} \qquad \text{(T-Proj)}$$