Programming Languages and Types

Klaus Ostermann

based on slides by Benjamin C. Pierce

The Lambda Calculus, formal

The lambda-calculus

- We have already studied the lambda calculus and some of its variants in the first part of the course.
- FAE *is* the lambda calculus plus a little bit of arithmetic.
- We can get rid of the arithmetic without loosing anything "essential".
- What this means is that other programming constructs can be *encoded* in the LC.

Formalities

Syntax

`t` ::=		*terms*
	`x`	*variable*
	`λx.t`	*abstraction*
	`t t`	*application*

*Term*inology:

- terms in the pure λ-calculus are often called λ-*terms*
- terms of the form `λx. t` are called λ-*abstractions* or just *abstractions*

Syntactic conventions

Since λ-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left

  E.g., $t\ u\ v$ *means* $(t\ u)\ v$, *not* $t\ (u\ v)$

- Bodies of λ- abstractions extend as far to the right as possible

  E.g., $\lambda x.\ \lambda y.\ x\ y$ *means* $\lambda x.\ (\lambda y.\ x\ y)$, *not* $\lambda x.\ (\lambda y.\ x)\ y$

Values

$\mathtt{v}$	::=	*values*
	$\lambda \mathtt{x}.\mathtt{t}$	*abstraction value*

Operational Semantics

Computation rule:

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[x \mapsto v_2]\, t_{12}$ *is "the term that results from substituting free occurrences of* x *in* t_{12} *with* v_{12}*."*

Congruence rules:

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

Terminology

A term of the form `(λx.t) v` — that is, a λ-abstraction applied to a *value* — is called a *redex* (short for "reducible expression").

Alternative evaluation strategies

Strictly speaking, the language we have defined is called the *pure, call-by-value lambda-calculus*.

Other evaluation strategies (call by name etc.) can be defined by changing the congruence rules accordingly.

In contrast to the substitution-based interpreter, we can also define full (non-deterministic) beta-reduction in SOS.

Programming in the Lambda-Calculus

Multiple arguments by Currying

Consider the function `double`, which returns a function as an argument.

$$\texttt{double} \quad = \quad \lambda\texttt{f. } \lambda\texttt{y. f (f y)}$$

This idiom — a λ-abstraction that does nothing but immediately yield another abstraction — is very common in the λ-calculus.

In general, $\lambda\texttt{x. } \lambda\texttt{y. t}$ is a function that, given a value `v` for `x`, yields a function that, given a value `u` for `y`, yields `t` with `v` in place of `x` and `u` in place of `y`.

That is, $\lambda\texttt{x. } \lambda\texttt{y. t}$ is a two-argument function.

The "Church Booleans"

```
tru  =  λt. λf. t
fls  =  λt. λf. f
```

```
     tru v w
  =  (λt.λf.t) v w     by definition
⟶   (λf. v) w          reducing the underlined redex
⟶   v                  reducing the underlined redex

     fls v w
  =  (λt.λf.f) v w     by definition
⟶   (λf. f) w          reducing the underlined redex
⟶   w                  reducing the underlined redex
```

Functions on Booleans

$$\texttt{not} \quad = \quad \lambda\texttt{b. b fls tru}$$

That is, `not` is a function that, given a boolean value `v`, returns `fls` if `v` is `tru` and `tru` if `v` is `fls`.

Functions on Booleans

$$\texttt{and} \quad = \quad \lambda\texttt{b}.\ \lambda\texttt{c}.\ \texttt{b}\ \texttt{c}\ \texttt{fls}$$

That is, `and` is a function that, given two boolean values `v` and `w`, returns `w` if `v` is `tru` and `fls` if `v` is `fls`
Thus `and v w` yields `tru` if both `v` and `w` are `tru` and `fls` if either `v` or `w` is `fls`.

Pairs

```
pair = λf.λs.λb. b f s
fst = λp. p tru
snd = λp. p fls
```

That is, `pair v w` is a function that, when applied to a boolean value `b`, applies `b` to `v` and `w`.
By the definition of booleans, this application yields `v` if `b` is `tru` and `w` if `b` is `fls`, so the first and second projection functions `fst` and `snd` can be implemented simply by supplying the appropriate boolean.

Example

	`fst (pair v w)`	
$=$	`fst ((λf. λs. λb. b f s) v w)`	by definition
$\longrightarrow$	`fst ((λs. λb. b v s) w)`	reducing
$\longrightarrow$	`fst (λb. b v w)`	reducing
$=$	`(λp. p tru) (λb. b v w)`	by definition
$\longrightarrow$	`(λb. b v w) tru`	reducing
$\longrightarrow$	`tru v w`	reducing
$\longrightarrow^*$	`v`	as before.

Church numerals

Idea: represent the number n by a function that "repeats some action n times."

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

That is, each number n is represented by a term c_n that takes two arguments, s and z (for "successor" and "zero"), and applies s, n times, to z.

Functions on Church Numerals

Successor:

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

Addition:

$$\texttt{plus} = \lambda\texttt{m}.\ \lambda\texttt{n}.\ \lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{m}\ \texttt{s}\ (\texttt{n}\ \texttt{s}\ \texttt{z})$$

Multiplication:

$$\texttt{times} = \lambda\texttt{m}.\ \lambda\texttt{n}.\ \texttt{m}\ (\texttt{plus}\ \texttt{n})\ \texttt{c}_0$$

Zero test:

$$\texttt{iszro} = \lambda\texttt{m}.\ \texttt{m}\ (\lambda\texttt{x}.\ \texttt{fls})\ \texttt{tru}$$

Predecessor is more difficult, but possible. I'll spare you the details.

Normal forms

Recall:

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus?

Does every term evaluate to a normal form?

Recursion and Divergence

$$\mathtt{Y} \;=\; \lambda \mathtt{f}.\ (\lambda \mathtt{x}.\ \mathtt{f}\ \mathtt{x}\ \mathtt{x})\ (\lambda \mathtt{x}.\ \mathtt{f}\ \mathtt{x}\ \mathtt{x})$$

$$\mathtt{Z} \;=\; \lambda \mathtt{f}.\ (\lambda \mathtt{x}.\ \mathtt{f}\ (\lambda \mathtt{y}.\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda \mathtt{x}.\ \mathtt{f}\ (\lambda \mathtt{y}.\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y}$$

We have already seen fixed point combinators at work.

$\mathtt{Y}$ works in a call-by-name setting; $\mathtt{Z}$ also works with call-by-value.

Can be used to write divergent terms, such as $\mathtt{Y}\ \lambda \mathtt{x}.\mathtt{x}$.

Induction on Derivations

Two induction principles

Like before, we have two ways to prove that properties are true of the untyped lambda calculus.

- Structural induction on terms
- Induction on a derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$.

Let's look at an example of each.

Structural induction on terms

To show that a property $\mathcal{P}$ holds for all lambda-terms $\mathtt{t}$, it suffices to show that

- $\mathcal{P}$ holds when $\mathtt{t}$ is a variable;
- $\mathcal{P}$ holds when $\mathtt{t}$ is a lambda-abstraction $\lambda\mathtt{x.\ t_1}$, assuming that $\mathcal{P}$ holds for the immediate subterm $\mathtt{t_1}$; and
- $\mathcal{P}$ holds when $\mathtt{t}$ is an application $\mathtt{t_1\ t_2}$, assuming that $\mathcal{P}$ holds for the immediate subterms $\mathtt{t_1}$ and $\mathtt{t_2}$.

N.b.: The variant of this principle where "immediate subterm" is replaced by "arbitrary subterm" is also valid. (Cf. *ordinary induction* vs. *complete induction* on the natural numbers.)

An example of structural induction on terms

Define the set of *free variables* in a lambda-term as follows:

$$
\begin{aligned}
FV(\mathtt{x}) &= \{\mathtt{x}\} \\
FV(\lambda \mathtt{x}.\mathtt{t}_1) &= FV(\mathtt{t}_1) \setminus \{\mathtt{x}\} \\
FV(\mathtt{t}_1\ \ \mathtt{t}_2) &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2)
\end{aligned}
$$

Define the *size* of a lambda-term as follows:

$$
\begin{aligned}
size(\mathtt{x}) &= 1 \\
size(\lambda \mathtt{x}.\mathtt{t}_1) &= size(\mathtt{t}_1) + 1 \\
size(\mathtt{t}_1\ \ \mathtt{t}_2) &= size(\mathtt{t}_1) + size(\mathtt{t}_2) + 1
\end{aligned}
$$

Theorem: $|FV(\mathtt{t})| \leq size(\mathtt{t})$.

An example of structural induction on terms

Theorem: $|FV(\mathtt{t})| \leq size(\mathtt{t})$.

Proof: By induction on the structure of $\mathtt{t}$.

- If $\mathtt{t}$ is a variable, then $|FV(\mathtt{t})| = 1 = size(\mathtt{t})$.
- If $\mathtt{t}$ is an abstraction $\lambda \mathtt{x}.\ \mathtt{t}_1$, then

$$
\begin{array}{lll}
 & |FV(\mathtt{t})| & \\
= & |FV(\mathtt{t}_1) \setminus \{\mathtt{x}\}| & \text{by defn} \\
\leq & |FV(\mathtt{t}_1)| & \text{by arithmetic} \\
\leq & size(\mathtt{t}_1) & \text{by induction hypothesis} \\
\leq & size(\mathtt{t}_1) + 1 & \text{by arithmetic} \\
= & size(\mathtt{t}) & \text{by defn.}
\end{array}
$$

An example of structural induction on terms

Theorem: $|FV(\mathtt{t})| \leq size(\mathtt{t})$.

Proof: By induction on the structure of $\mathtt{t}$.

- If $\mathtt{t}$ is an application $\mathtt{t}_1\ \mathtt{t}_2$, then

$$
\begin{array}{lll}
 & |FV(\mathtt{t})| & \\
= & |FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2)| & \text{by defn} \\
\leq & max(|FV(\mathtt{t}_1)|, |FV(\mathtt{t}_2)|) & \text{by arithmetic} \\
\leq & max(size(\mathtt{t}_1), size(\mathtt{t}_2)) & \text{by IH and arithmetic} \\
\leq & |size(\mathtt{t}_1)| + |size(\mathtt{t}_2)| & \text{by arithmetic} \\
\leq & |size(\mathtt{t}_1)| + |size(\mathtt{t}_2)| + 1 & \text{by arithmetic} \\
= & size(\mathtt{t}) & \text{by defn.}
\end{array}
$$

Induction on derivations

Recall that the reduction relation is defined as the smallest binary relation on terms satisfying the following rules:

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

Induction on derivations

Induction principle for the small-step evaluation relation.

To show that a property $\mathcal{P}$ holds for all derivations of $\mathtt{t} \longrightarrow \mathtt{t}'$, it suffices to show that

- $\mathcal{P}$ holds for all derivations that use the rule E-AppAbs;
- $\mathcal{P}$ holds for all derivations that end with a use of E-App1 assuming that $\mathcal{P}$ holds for all subderivations; and
- $\mathcal{P}$ holds for all derivations that end with a use of E-App2 assuming that $\mathcal{P}$ holds for all subderivations.

Example

Theorem: if $\mathtt{t} \longrightarrow \mathtt{t}'$ then $FV(\mathtt{t}) \supseteq FV(\mathtt{t}')$.

Induction on derivations

We must prove, for all derivations of $\mathtt{t} \longrightarrow \mathtt{t}'$, that $FV(\mathtt{t}) \supseteq FV(\mathtt{t}')$.

There are three cases.

- If the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$ is just a use of E-AppAbs, then $\mathtt{t}$ is $(\lambda \mathtt{x}.\mathtt{t}_1)\mathtt{v}$ and $\mathtt{t}'$ is $[\mathtt{x}|\!\rightarrow\!\mathtt{v}]\mathtt{t}_1$. Reason as follows:

$$\begin{aligned} FV(\mathtt{t}) &= FV((\lambda \mathtt{x}.\mathtt{t}_1)\mathtt{v}) \\ &= FV(\mathtt{t}_1)/\{\mathtt{x}\} \cup FV(\mathtt{v}) \\ &\supseteq FV([\mathtt{x}|\!\rightarrow\!\mathtt{v}]\mathtt{t}_1) \\ &= FV(\mathtt{t}') \end{aligned}$$

- If the derivation ends with a use of E-App1, then $\mathtt{t}$ has the form $\mathtt{t}_1\ \mathtt{t}_2$ and $\mathtt{t}'$ has the form $\mathtt{t}'_1\ \mathtt{t}_2$, and we have a subderivation of $\mathtt{t}_1 \longrightarrow \mathtt{t}'_1$

  By the induction hypothesis, $FV(\mathtt{t}_1) \supseteq FV(\mathtt{t}'_1)$. Now calculate:

$$\begin{aligned} FV(t) &= FV(\mathtt{t}_1\ \mathtt{t}_2) \\ &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2) \\ &\supseteq FV(\mathtt{t}'_1) \cup FV(\mathtt{t}_2) \\ &= FV(\mathtt{t}'_1\ \mathtt{t}_2) \\ &= FV(t') \end{aligned}$$

- If the derivation ends with a use of E-App2, the argument is similar to the previous case.

Substitution and α-Equivalence

Substitution

Our definition of evaluation is based on the "substitution" of values for free variables within terms.

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

But what is substitution, exactly? How do we define it?

Answer: It's almost how we defined substitution for FAE and related languages.

One glitch: Substitution of a variable with terms with free variables.

Substitution as defined for FAE et al

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{y}.\mathtt{t}_1) = \lambda \mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{x}.\mathtt{t}_1) = \lambda \mathtt{x}.\ \mathtt{t}_1 & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

What is wrong with this definition?

It suffers from *variable capture*!

$$[\mathtt{x} \mapsto \mathtt{y}](\lambda \mathtt{y}.\mathtt{x}) = \lambda \mathtt{x}.\ \mathtt{x}$$

Only a problem if terms with free variables can occur.

Trying to fix substitution...

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{y}.\mathtt{t}_1) = \lambda \mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \text{if } \mathtt{x} \neq \mathtt{y},\ \mathtt{y} \notin \mathit{FV}(\mathtt{s}) \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{x}.\mathtt{t}_1) = \lambda \mathtt{x}.\ \mathtt{t}_1 & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

What is wrong with this definition?

Now substition is a *partial function!*

E.g., $[\mathtt{x} \mapsto \mathtt{y}](\lambda \mathtt{y}.\mathtt{x})$ is undefined.

But we want an result for every substitution.

Bound variable names shouldn't matter

It's annoying that that the "spelling" of bound variable names is causing trouble with our definition of substitution.

Intuition tells us that there shouldn't be a difference between the functions $\lambda x.x$ and $\lambda y.y$. Both of these functions do exactly the same thing.

Because they differ only in the names of their bound variables, we'd like to think that these *are* the same function.

We call such terms *alpha-equivalent*.

Alpha-equivalence classes

In fact, we can create equivalence classes of terms that differ only in the names of bound variables.

When working with the lambda calculus, it is convenient to think about these *equivalence classes*, instead of raw terms.

For example, when we write $\lambda \mathtt{x}.\mathtt{x}$ we mean not just this term, but the class of terms that includes $\lambda \mathtt{y}.\mathtt{y}$ and $\lambda \mathtt{z}.\mathtt{z}$.

We can now freely choose a different *representative* from a term's alpha-equivalence class, whenever we need to, to avoid getting stuck.

In Handin-1, you have already seen one way to represent alpha equivalence classes: de Bruijn indices.

Substitution, for alpha-equivalence classes

Now consider substitution as an operation over *alpha-equivalence classes* of terms.

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{y}.\mathtt{t}_1) = \lambda \mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \text{if } \mathtt{x} \neq \mathtt{y},\ \mathtt{y} \notin \mathit{FV}(\mathtt{s}) \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{x}.\mathtt{t}_1) = \lambda \mathtt{x}.\ \mathtt{t}_1 & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

Examples:

- $[\mathtt{x} \mapsto \mathtt{y}](\lambda \mathtt{y}.\mathtt{x})$ must give the same result as $[\mathtt{x} \mapsto \mathtt{y}](\lambda \mathtt{z}.\mathtt{x})$. We know the latter is $\lambda \mathtt{z}.\mathtt{y}$, so that is what we will use for the former.
- $[\mathtt{x} \mapsto \mathtt{y}](\lambda \mathtt{x}.\mathtt{z})$ must give the same result as $[\mathtt{x} \mapsto \mathtt{y}](\lambda \mathtt{w}.\mathtt{z})$. We know the latter is $\lambda \mathtt{w}.\mathtt{z}$ so that is what we use for the former.

Types

Plan

- For now, we'll go back to the simple language of arithmetic and boolean expressions and show how to equip it with a (very simple) type system
- The key property of this type system will be *soundness*: Well-typed programs do not get stuck
- After that, we'll develop a simple type system for the lambda-calculus
- We'll spend a good part of the rest of the semester adding features to this type system

Outline

1. begin with a set of terms, a set of values, and an evaluation relation
2. define a set of *types* classifying values according to their "shapes"
3. define a *typing relation* `t : T` that classifies terms according to the shape of the values that result from evaluating them
4. check that the typing relation is *sound* in the sense that,
   - 4.1 if `t : T` and `t` $\longrightarrow^*$ `v`, then `v : T`
   - 4.2 if `t : T`, then evaluation of `t` will not get stuck

Review: Arithmetic Expressions – Syntax

t ::=		*terms*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
	`0`	*constant zero*
	`succ t`	*successor*
	`pred t`	*predecessor*
	`iszero t`	*zero test*
v ::=		*values*
	`true`	*true value*
	`false`	*false value*
	`nv`	*numeric value*
nv ::=		*numeric values*
	`0`	*zero value*
	`succ nv`	*successor value*

Evaluation Rules

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-If)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'} \qquad \text{(E-Succ)}$$

$$\texttt{pred}\ 0 \longrightarrow 0 \qquad \text{(E-PredZero)}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1 \qquad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'} \qquad \text{(E-Pred)}$$

$$\texttt{iszero}\ 0 \longrightarrow \texttt{true} \qquad \text{(E-IszeroZero)}$$

$$\texttt{iszero}\ (\texttt{succ}\ nv_1) \longrightarrow \texttt{false} \qquad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero}\ t_1 \longrightarrow \texttt{iszero}\ t_1'} \qquad \text{(E-IsZero)}$$

Types

In this language, values have two possible "shapes": they are either booleans or numbers.

`T` ::=		*types*
	`Bool`	*type of booleans*
	`Nat`	*type of numbers*

Typing Rules

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

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

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

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ } t_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred } t_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero } t_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

Typing Derivations

Every pair $(\texttt{t}, \texttt{T})$ in the typing relation can be justified by a *derivation tree* built from instances of the inference rules.

$$\dfrac{\dfrac{\dfrac{}{\texttt{0 : Nat}}\ \textsc{T-Zero}}{\texttt{iszero 0 : Bool}}\ \textsc{T-IsZero} \qquad \dfrac{}{\texttt{0 : Nat}}\ \textsc{T-Zero} \qquad \dfrac{\dfrac{}{\texttt{0 : Nat}}\ \textsc{T-Zero}}{\texttt{pred 0 : Nat}}\ \textsc{T-Pred}}{\texttt{if iszero 0 then 0 else pred 0 : Nat}}\ \textsc{T-If}$$

Proofs of properties about the typing relation often proceed by induction on typing derivations.

Imprecision of Typing

Like other static program analyses, type systems are generally *imprecise*: they do not predict exactly what kind of value will be returned by every program, but just a conservative (safe) approximation.

$$\frac{\texttt{t}_1 : \texttt{Bool} \qquad \texttt{t}_2 : \texttt{T} \qquad \texttt{t}_3 : \texttt{T}}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}} \qquad \text{(T-If)}$$

Using this rule, we cannot assign a type to

```
if true then 0 else false
```

even though this term will certainly evaluate to a number.

Properties of the Typing Relation

Type Safety

The safety (or soundness) of this type system can be expressed by two properties:

1. *Progress:* A well-typed term is not stuck

   If $t : T$, *then either* t *is a value or else* $t \longrightarrow t'$ *for some* t'.

2. *Preservation:* Types are preserved by one-step evaluation

   If $t : T$ *and* $t \longrightarrow t'$, *then* $t' : T$.

Inversion

Lemma:

1. If $\texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : \texttt{R}$, then $t_1 : \texttt{Bool}$, $t_2 : \texttt{R}$, and $t_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
6. If $\texttt{pred } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
7. If $\texttt{iszero } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $t_1 : \texttt{Nat}$.

Proof: ...

This leads directly to a recursive algorithm for calculating the type of a term...

Typechecking Algorithm

```
typeof(t) = if t = true then Bool
            else if t = false then Bool
            else if t = if t1 then t2 else t3 then
              let T1 = typeof(t1) in
              let T2 = typeof(t2) in
              let T3 = typeof(t3) in
              if T1 = Bool and T2=T3 then T2
              else "not typable"
            else if t = 0 then Nat
            else if t = succ t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Nat else "not typable"
            else if t = pred t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Nat else "not typable"
            else if t = iszero t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Bool else "not typable"
```

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof: Recall the syntax of values:

`v`	::=	*values*
	`true`	*true value*
	`false`	*false value*
	`nv`	*numeric value*
`nv`	::=	*numeric values*
	`0`	*zero value*
	`succ nv`	*successor value*

For part 1, if `v` is `true` or `false`, the result is immediate. But `v` cannot be `0` or `succ nv`, since the inversion lemma tells us that `v` would then have type `Nat`, not `Bool`. Part 2 is similar.

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof:
By induction on a derivation of $\mathtt{t} : \mathtt{T}$.

The T-True, T-False, and T-Zero cases are immediate, since $\mathtt{t}$ in these cases is a value.

Case T-If: $\quad \mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$
$\mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$

By the induction hypothesis, either $\mathtt{t}_1$ is a value or else there is some $\mathtt{t}_1'$ such that $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$. If $\mathtt{t}_1$ is a value, then the canonical forms lemma tells us that it must be either $\mathtt{true}$ or $\mathtt{false}$, in which case either E-IfTrue or E-IfFalse applies to $\mathtt{t}$. On the other hand, if $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$, then, by E-If, $\mathtt{t} \longrightarrow \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$.

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof: By induction on a derivation of $\mathtt{t} : \mathtt{T}$.

The cases for rules T-Zero, T-Succ, T-Pred, and T-IsZero are similar.

(Recommended: Try to reconstruct them.)

Preservation

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

Proof: By induction on the given typing derivation.

Preservation

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

Proof: By induction on the given typing derivation.

Preservation

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

Proof: By induction on the given typing derivation.

Case T-TRUE: $\mathtt{t} = \mathtt{true}$ $\mathtt{T} = \mathtt{Bool}$

Then $\mathtt{t}$ is a value, so it cannot be that $\mathtt{t} \longrightarrow \mathtt{t}'$ for any $\mathtt{t}'$, and the theorem is vacuously true.

Preservation

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

Proof: By induction on the given typing derivation.

Case T-If:

$$\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \quad \mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$$

There are three evaluation rules by which $\mathtt{t} \longrightarrow \mathtt{t}'$ can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.

Preservation

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

Proof: By induction on the given typing derivation.

Case T-If:

$\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \quad \mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$

There are three evaluation rules by which $\mathtt{t} \longrightarrow \mathtt{t}'$ can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.

Subcase E-IfTrue: $\mathtt{t}_1 = \mathtt{true} \qquad \mathtt{t}' = \mathtt{t}_2$

Immediate, by the assumption $\mathtt{t}_2 : \mathtt{T}$.

(E-IfFalse subcase: Similar.)

Preservation

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

Proof: By induction on the given typing derivation.

Case T-If:

$$\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \quad \mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$$

There are three evaluation rules by which $\mathtt{t} \longrightarrow \mathtt{t}'$ can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.

Subcase E-If: $\mathtt{t}_1 \longrightarrow \mathtt{t}_1' \qquad \mathtt{t}' = \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$

Applying the IH to the subderivation of $\mathtt{t}_1 : \mathtt{Bool}$ yields $\mathtt{t}_1' : \mathtt{Bool}$. Combining this with the assumptions that $\mathtt{t}_2 : \mathtt{T}$ and $\mathtt{t}_3 : \mathtt{T}$, we can apply rule T-If to conclude that $\mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}$, that is, $\mathtt{t}' : \mathtt{T}$.

The Simply Typed Lambda-Calculus

The simply typed lambda-calculus

The system we are about to define is commonly called the *simply typed lambda-calculus*, or $\lambda_{\to}$ for short.

Unlike the untyped lambda-calculus, the "pure" form of $\lambda_{\to}$ (with no primitive values or operations) is not very interesting; to talk about $\lambda_{\to}$, we always begin with some set of "base types."

- So, strictly speaking, there are *many* variants of $\lambda_{\to}$, depending on the choice of base types.
- For now, we'll work with a variant constructed over the booleans.

Untyped lambda-calculus with booleans

`t` ::=		*terms*
	`x`	*variable*
	`λx.t`	*abstraction*
	`t t`	*application*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
`v` ::=		*values*
	`λx.t`	*abstraction value*
	`true`	*true value*
	`false`	*false value*

"Simple Types"

T ::=		*types*
	Bool	*type of booleans*
	T→T	*types of functions*

Type Annotations

We now have a choice to make. Do we...

- annotate lambda-abstractions with the expected type of the argument

  $$\lambda x{:}T_1.\ t_2$$

  (as in most mainstream programming languages), or
- continue to write lambda-abstractions as before

  $$\lambda x.\ t_2$$

  and ask the typing rules to "guess" an appropriate annotation (as in OCaml)?

Both are reasonable choices, but the first makes the job of defining the typin rules simpler. Let's take this choice for now.

Typing rules

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

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

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

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

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

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

Typing rules

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

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

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

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

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

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

Typing rules

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

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

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

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

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

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

Typing rules

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

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

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

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

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

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

Typing Derivations

What derivations justify the following typing statements?

- $\vdash$ (λx:Bool.x) true : Bool
- f:Bool→Bool $\vdash$ f (if false then true else false) : Bool
- f:Bool→Bool $\vdash$
  λx:Bool. f (if x then false else x) : Bool→Bool

Properties of $\lambda_{\to}$

The fundamental property of the type system we have just defined is *soundness* with respect to the operational semantics.

1. *Progress:* A closed, well-typed term is not stuck

   If $\vdash t : T$, *then either* t *is a value or else* $t \longrightarrow t'$ *for some* t'.

2. *Preservation:* Types are preserved by one-step evaluation

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

Proving progress

Same steps as before...

- inversion lemma for typing relation
- canonical forms lemma
- progress theorem

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\Gamma \vdash \texttt{t}_1 : \texttt{Bool}$ and $\Gamma \vdash \texttt{t}_2, \texttt{t}_3 : \texttt{R}$.
4. If $\Gamma \vdash \texttt{x} : \texttt{R}$, then $\texttt{x:R} \in \Gamma$.
5. If $\Gamma \vdash \lambda \texttt{x:T}_1 . \texttt{t}_2 : \texttt{R}$, then $\texttt{R} = \texttt{T}_1 \to \texttt{R}_2$ for some $\texttt{R}_2$ with $\Gamma, \texttt{x:T}_1 \vdash \texttt{t}_2 : \texttt{R}_2$.
6. If $\Gamma \vdash \texttt{t}_1 \ \texttt{t}_2 : \texttt{R}$, then there is some type $\texttt{T}_{11}$ such that $\Gamma \vdash \texttt{t}_1 : \texttt{T}_{11} \to \texttt{R}$ and $\Gamma \vdash \texttt{t}_2 : \texttt{T}_{11}$.

Canonical Forms

Lemma:

1. If $\mathtt{v}$ is a value of type $\mathtt{Bool}$, then $\mathtt{v}$ is either $\mathtt{true}$ or $\mathtt{false}$.
2. If $\mathtt{v}$ is a value of type $\mathtt{T}_1 \to \mathtt{T}_2$, then $\mathtt{v}$ has the form $\lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2$.

Progress

Theorem: Suppose $\mathtt{t}$ is a closed, well-typed term (that is, $\vdash \mathtt{t} : \mathtt{T}$ for some $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because $\mathtt{t}$ is closed). The abstraction case is immediate, since abstractions are values.
Consider the case for application, where $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ with $\vdash \mathtt{t}_1 : \mathtt{T}_{11}{\to}\mathtt{T}_{12}$ and $\vdash \mathtt{t}_2 : \mathtt{T}_{11}$. By the induction hypothesis, either $\mathtt{t}_1$ is a value or else it can make a step of evaluation, and likewise $\mathtt{t}_2$. If $\mathtt{t}_1$ can take a step, then rule E-App1 applies to $\mathtt{t}$. If $\mathtt{t}_1$ is a value and $\mathtt{t}_2$ can take a step, then rule E-App2 applies. Finally, if both $\mathtt{t}_1$ and $\mathtt{t}_2$ are values, then the canonical forms lemma tells us that $\mathtt{t}_1$ has the form $\lambda\mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12}$, and so rule E-AppAbs applies to $\mathtt{t}$.

Preservation

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

Proof: ...

Preservation

Recommended: Complete the proof of preservation

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* $\mathtt{unit}$
`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

$\mathtt{t}$	::=	...	*terms*
		$\mathtt{t}_1\,;\mathtt{t}_2$	

Sequencing

$$\begin{array}{lll} \mathtt{t} & ::= & \ldots \qquad\qquad\qquad\qquad\qquad\qquad \textit{terms} \\ & & \mathtt{t}_1\mathtt{;}\mathtt{t}_2 \end{array}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{;}\mathtt{t}_2 \longrightarrow \mathtt{t}_1'\mathtt{;}\mathtt{t}_2} \qquad \text{(E-Seq)}$$

$$\mathtt{unit;t}_2 \longrightarrow \mathtt{t}_2 \qquad \text{(E-SeqNext)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Unit} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\mathtt{;}\mathtt{t}_2 : \mathtt{T}_2} \qquad \text{(T-Seq)}$$

Derived forms

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

Sequencing as a derived form

$$\texttt{t}_1\texttt{;t}_2 \quad \stackrel{\text{def}}{=} \quad (\lambda\texttt{x:Unit.t}_2)\ \texttt{t}_1$$
$$\text{where } \texttt{x} \notin \mathit{FV}(\texttt{t}_2)$$

Ascription

New syntactic forms

$$\begin{array}{lll} \texttt{t} & ::= & \ldots \\ & & \texttt{t as T} \end{array}$$

terms

ascription

New evaluation rules

$$\boxed{\texttt{t} \longrightarrow \texttt{t}'}$$

$$\texttt{v}_1 \texttt{ as T} \longrightarrow \texttt{v}_1 \qquad \text{(E-Ascribe)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{t}_1 \texttt{ as T} \longrightarrow \texttt{t}_1' \texttt{ as T}} \qquad \text{(E-Ascribe1)}$$

New typing rules

$$\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}}{\Gamma \vdash \texttt{t}_1 \texttt{ as T} : \texttt{T}} \qquad \text{(T-Ascribe)}$$

Ascription as a derived form

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

Let-bindings

New syntactic forms

$\mathtt{t} \;::=\; \ldots$ *terms*

$\quad\quad \mathtt{let\ x{=}t\ in\ t}$ *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, tuples, and records

Pairs

`t`	::=	...	*terms*
		`{t,t}`	*pair*
		`t.1`	*first projection*
		`t.2`	*second projection*
`v`	::=	...	*values*
		`{v,v}`	*pair value*
`T`	::=	...	*types*
		$\mathtt{T}_1 \times \mathtt{T}_2$	*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

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

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

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

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

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

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

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

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

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

Evaluation rules for records

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

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

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

Typing rules for records

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

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

Sums and variants

New syntactic forms

`t`	::=	...	*terms*
		`inl t`	*tagging (left)*
		`inr t`	*tagging (right)*
		`case t of inl x⇒t | inr x⇒t`	*case*
`v`	::=	...	*values*
		`inl v`	*tagged value (left)*
		`inr v`	*tagged value (right)*
`T`	::=	...	*types*
		`T+T`	*sum type*

$\texttt{T}_1\texttt{+}\texttt{T}_2$ is a *disjoint union* of $\texttt{T}_1$ and $\texttt{T}_2$ (the tags `inl` and `inr` ensure disjointness)

New evaluation rules

$\boxed{t \longrightarrow t'}$

$$\texttt{case (inl } v_0\texttt{) of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow [x_1 \mapsto v_0]t_1 \qquad \text{(E-CaseInl)}$$

$$\texttt{case (inr } v_0\texttt{) of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow [x_2 \mapsto v_0]t_2 \qquad \text{(E-CaseInr)}$$

$$\frac{t_0 \longrightarrow t_0'}{\begin{array}{l}\texttt{case } t_0 \texttt{ of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \\ \longrightarrow \texttt{case } t_0' \texttt{ of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2\end{array}} \qquad \text{(E-Case)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inl } t_1 \longrightarrow \texttt{inl } t_1'} \qquad \text{(E-Inl)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inr } t_1 \longrightarrow \texttt{inr } t_1'} \qquad \text{(E-Inr)}$$

New typing rules

$\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl}\ t_1 : T_1 + T_2} \quad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \texttt{inr}\ t_1 : T_1 + T_2} \quad \text{(T-Inr)}$$

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma, x_1 : T_1 \vdash t_1 : T \qquad \Gamma, x_2 : T_2 \vdash t_2 : T}{\Gamma \vdash \texttt{case}\ t_0\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1\ \mid\ \texttt{inr}\ x_2 \Rightarrow t_2 : T} \quad \text{(T-Case)}$$

Sums and Uniqueness of Types

Problem:

> *If* `t` *has type* `T`*, then* `inl t` *has type* `T+U` *for* every `U`*.*

I.e., we've lost uniqueness of types.

Possible solutions:

- "Infer" `U` as needed during typechecking
- Give constructors different names and only allow each name to appear in one sum type (requires generalization to "variants," which we'll see next) — OCaml's solution
- Annotate each `inl` and `inr` with the intended sum type.

For simplicity, let's choose the third.

New syntactic forms

`t`	::=	...	*terms*
		`inl t as T`	*tagging (left)*
		`inr t as T`	*tagging (right)*
`v`	::=	...	*values*
		`inl v as T`	*tagged value (left)*
		`inr v as T`	*tagged value (right)*

Note that `as T` here is not the ascription operator that we saw before — i.e., not a separate syntactic form: in essence, there is an ascription "built into" every use of `inl` or `inr`.

New typing rules

$\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl}\ t_1\ \texttt{as}\ T_1+T_2 : T_1+T_2} \quad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \texttt{inr}\ t_1\ \texttt{as}\ T_1+T_2 : T_1+T_2} \quad \text{(T-Inr)}$$

Evaluation rules ignore annotations:

$\boxed{t \longrightarrow t'}$

$$\texttt{case (inl } v_0 \texttt{ as } T_0\texttt{)}\ \texttt{of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow [x_1 \mapsto v_0]t_1 \qquad \text{(E-CaseInl)}$$

$$\texttt{case (inr } v_0 \texttt{ as } T_0\texttt{)}\ \texttt{of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow [x_2 \mapsto v_0]t_2 \qquad \text{(E-CaseInr)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inl } t_1 \texttt{ as } T_2 \longrightarrow \texttt{inl } t_1' \texttt{ as } T_2} \qquad \text{(E-Inl)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inr } t_1 \texttt{ as } T_2 \longrightarrow \texttt{inr } t_1' \texttt{ as } T_2} \qquad \text{(E-Inr)}$$

Variants

Just as we generalized binary products to labeled records, we can generalize binary sums to labeled *variants*.

New syntactic forms

`t`	::=	...	*terms*
		`<l=t> as T`	*tagging*
		`case t of` $\texttt{<l}_i\texttt{=x}_i\texttt{>}\Rightarrow\texttt{t}_i\ ^{i\in 1..n}$	*case*
`T`	::=	...	*types*
		$\texttt{<l}_i\texttt{:T}_i\ ^{i\in 1..n}\texttt{>}$	*type of variants*

New evaluation rules

$\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{case (<l}_j\texttt{=v}_j\texttt{> as T) of <l}_i\texttt{=x}_i\texttt{>}\Rightarrow\texttt{t}_i\ ^{i\in 1..n} \longrightarrow [\texttt{x}_j \mapsto \texttt{v}_j]\texttt{t}_j \qquad \text{(E-CaseVariant)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}'_0}{\texttt{case t}_0\texttt{ of <l}_i\texttt{=x}_i\texttt{>}\Rightarrow\texttt{t}_i\ ^{i\in 1..n} \longrightarrow \texttt{case t}'_0\texttt{ of <l}_i\texttt{=x}_i\texttt{>}\Rightarrow\texttt{t}_i\ ^{i\in 1..n}} \qquad \text{(E-Case)}$$

$$\frac{\texttt{t}_i \longrightarrow \texttt{t}'_i}{\texttt{<l}_i\texttt{=t}_i\texttt{> as T} \longrightarrow \texttt{<l}_i\texttt{=t}'_i\texttt{> as T}} \qquad \text{(E-Variant)}$$

New typing rules

$\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \langle l_j = t_j \rangle \text{ as } \langle l_i : T_i \,^{i \in 1..n} \rangle : \langle l_i : T_i \,^{i \in 1..n} \rangle} \quad \text{(T-Variant)}$$

$$\frac{\Gamma \vdash t_0 : \langle l_i : T_i \,^{i \in 1..n} \rangle \qquad \text{for each } i \quad \Gamma, x_i : T_i \vdash t_i : T}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i \,^{i \in 1..n} : T} \quad \text{(T-Case)}$$

Example

```
Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;

a = <physical=pa> as Addr;

getName = λa:Addr.
  case a of
    <physical=x> ⇒ x.firstlast
  | <virtual=y> ⇒ y.name;
```

Options and Enumerations

can be encoded using sum and product types, just like in Haskell.

Recursion

Recursion in $\lambda_{\to}$

- In $\lambda_{\to}$, all programs terminate. (Cf. Chapter 12.)
- Hence, untyped terms like $\mathtt{Y}$ and $\mathtt{Z}$ are not typable.
- But we can *extend* the system with a (typed) fixed-point operator...

Example

```
ff = λie:Nat→Bool.
       λx:Nat.
         if iszero x then true
         else if iszero (pred x) then false
         else ie (pred (pred x));

iseven = fix ff;

iseven 7;
```

New syntactic forms

$$\begin{array}{lll} \texttt{t} & ::= & \ldots \qquad \textit{terms} \\ & & \texttt{fix t} \qquad \textit{fixed point of } \texttt{t} \end{array}$$

New evaluation rules

$$\boxed{\texttt{t} \longrightarrow \texttt{t}'}$$

$$\texttt{fix } (\lambda \texttt{x:T}_1.\texttt{t}_2) \longrightarrow [\texttt{x} \mapsto (\texttt{fix } (\lambda \texttt{x:T}_1.\texttt{t}_2))]\texttt{t}_2 \qquad \text{(E-FixBeta)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{fix t}_1 \longrightarrow \texttt{fix t}_1'} \qquad \text{(E-Fix)}$$

New typing rules

$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \to \mathtt{T}_1}{\Gamma \vdash \mathtt{fix}\ \mathtt{t}_1 : \mathtt{T}_1} \qquad \text{(T-Fix)}$$

A more convenient form

$$\texttt{letrec } x{:}T_1{=}t_1 \texttt{ in } t_2 \stackrel{\text{def}}{=} \texttt{let } x = \texttt{fix } (\lambda x{:}T_1.t_1) \texttt{ in } t_2$$

```
letrec iseven : Nat→Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;
```