Formal Methods

Lecture 8

(B. Pierce's slides for the book "Types and Programming Languages")

Erasure and Typability

Erasure

We can transform terms in $\lambda_{\to}$ to terms of the untyped lambda-calculus simply by erasing type annotations on lambda-abstractions.

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

Typability

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

This process is called *type reconstruction* or *type inference*.

Typability

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

This process is called *type reconstruction* or *type inference*.

Example: Is the term

λ`x. x x`

typable?

The Curry-Howard Correspondence

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 $\mathrm{P}\rightarrow\mathrm{Q}$
proposition $P \wedge Q$	type $\mathrm{P} \times \mathrm{Q}$
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

On to real programming languages…

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

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

Sequencing

t	$::=$	$\ldots$	*terms*
		$t_1 ; t_2$	

Sequencing

t ::= ... *terms*

$t_1 ; t_2$

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

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

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

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 $x \notin FV(t_2)$

Ascription

New syntactic forms

$t ::= \ldots$ *terms*

$\quad t \text{ as } T$ *ascription*

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

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

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

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

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

Ascription as a derived form

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

Let-bindings

New syntactic forms

t ::= ... *terms*

$\quad \texttt{let } x\texttt{=}t \texttt{ in } t$ *let binding*

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

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

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

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

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

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*
		$T_1 \times T_2$	*product type*

Evaluation rules for pairs

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

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

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

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

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

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

Typing rules for pairs

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

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

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

Tuples

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

Evaluation rules for tuples

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

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

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

Typing rules for tuples

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

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

Records

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

Evaluation rules for records

$$\{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

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

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

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

Sums – motivating example

```
PhysicalAddr = {firstlast:String, addr:String}
VirtualAddr  = {name:String, email:String}
Addr         = PhysicalAddr + VirtualAddr
inl : "PhysicalAddr → PhysicalAddr+VirtualAddr"
inr : "VirtualAddr → PhysicalAddr+VirtualAddr"
```

```
getName = λa:Addr.
  case a of
    inl x ⇒ x.firstlast
  | inr y ⇒ y.name;
```

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*

T_1+T_2 is a *disjoint union* of T_1 and T_2 (the tags inl and inr ensure disjointness)

New evaluation rules

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

$$\begin{array}{ll} \mathtt{case}\ (\mathtt{inl}\ \mathtt{v}_0) & \longrightarrow [\mathtt{x}_1 \mapsto \mathtt{v}_0]\mathtt{t}_1 \\ \mathtt{of}\ \mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathtt{t}_1 \mid \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathtt{t}_2 & \end{array} \qquad \text{(E-CaseInl)}$$

$$\begin{array}{ll} \mathtt{case}\ (\mathtt{inr}\ \mathtt{v}_0) & \longrightarrow [\mathtt{x}_2 \mapsto \mathtt{v}_0]\mathtt{t}_2 \\ \mathtt{of}\ \mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathtt{t}_1 \mid \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathtt{t}_2 & \end{array} \qquad \text{(E-CaseInr)}$$

$$\frac{\mathtt{t}_0 \longrightarrow \mathtt{t}_0'}{\begin{array}{c} \mathtt{case}\ \mathtt{t}_0\ \mathtt{of}\ \mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathtt{t}_1 \mid \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathtt{t}_2 \\ \longrightarrow \mathtt{case}\ \mathtt{t}_0'\ \mathtt{of}\ \mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathtt{t}_1 \mid \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathtt{t}_2 \end{array}} \qquad \text{(E-Case)}$$

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

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

New typing rules

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

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

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

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

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

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

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

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

Evaluation rules ignore annotations:

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

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

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

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

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

Variants

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

New syntactic forms

t ::= ...		*terms*
	$\langle l=t \rangle$ as T	*tagging*
	case t of $\langle l_i = x_i \rangle \Rightarrow t_i \;^{i \in 1..n}$	*case*
T ::= ...		*types*
	$\langle l_i : T_i \;^{i \in 1..n} \rangle$	*type of variants*

New evaluation rules

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

$$\texttt{case}\ (<l_j = v_j>\ \texttt{as}\ T)\ \texttt{of}\ <l_i = x_i> \Rightarrow t_i\ ^{i \in 1..n} \longrightarrow [x_j \to v_j]t_j \qquad \text{(E-CaseVariant)}$$

$$\frac{t_0 \longrightarrow t_0'}{\texttt{case}\ t_0\ \texttt{of}\ <l_i = x_i> \Rightarrow t_i\ ^{i \in 1..n} \longrightarrow \texttt{case}\ t_0'\ \texttt{of}\ <l_i = x_i> \Rightarrow t_i\ ^{i \in 1..n}} \qquad \text{(E-Case)}$$

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

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

Just like in OCaml...

```
OptionalNat = <none:Unit, some:Nat>;

Table = Nat→OptionalNat;

emptyTable = λn:Nat. <none=unit> as OptionalNat;

extendTable =
  λt:Table. λm:Nat. λv:Nat.
    λn:Nat.
      if  equal n m then <some=v> as OptionalNat
      else t n;

x = case t(5) of
      <none=u> ⇒ 999
    | <some=v> ⇒ v;
```

Enumerations

```
Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit,
           thursday:Unit, friday:Unit>;


nextBusinessDay = λw:Weekday.
  case w of <monday=x>     ⇒ <tuesday=unit> as Weekday
          | <tuesday=x>    ⇒ <wednesday=unit> as Weekday
          | <wednesday=x>  ⇒ <thursday=unit> as Weekday
          | <thursday=x>   ⇒ <friday=unit> as Weekday
          | <friday=x>     ⇒ <monday=unit> as Weekday;
```

Recursion

Recursion in $\lambda_{\rightarrow}$

- In $\lambda_{\rightarrow}$, all programs terminate.
- Hence, untyped terms like omega and `fix` 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

$\mathtt{t}$	::=	...	*terms*
		$\mathtt{fix}\ \mathtt{t}$	*fixed point of* t

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

$$\mathtt{fix}\ (\lambda x{:}T_1.\mathtt{t}_2) \longrightarrow [x \mapsto (\mathtt{fix}\ (\lambda x{:}T_1.\mathtt{t}_2))]\mathtt{t}_2 \qquad \text{(E-FixBeta)}$$

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

New typing rules

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

$$\frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \text{fix } t_1 : T_1} \quad \text{(T-Fix)}$$

A more convenient form

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

```
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;
```