CS150 APL: Type Systems and Effects

Guannan Wei
guannan.wei@tufts.edu

Sept 11, 2025

Tufts University

Last time

- Simply typed lambda calculus (STLC)
- Soundness, incompleteness
- Polymorphic lambda calculus (System F)
- Unit type

Today

More expressive type systems:

- Product types
- Sum types
- Existential types
- Effects
- ...

Product type

- Product type (pair type, tuple type): $\tau_1 \times \tau_2$
- E.g. `struct` in C but without field names

Syntax

$$
\begin{array}{rcll}
t & ::= & \cdots \mid (t_1, t_2) \mid \text{fst } t \mid \text{snd } t & \textbf{terms} \\
v & ::= & \cdots \mid (v_1, v_2) & \textbf{values} \\
\tau & ::= & \cdots \mid \tau_1 \times \tau_2 & \textbf{types} \\
E & ::= & \cdots \mid (v, E) \mid (E, t) \mid \text{fst } E \mid \text{snd } E & \textbf{reduction contexts}
\end{array}
$$

Product type

Dynamics

$$\frac{}{\text{fst } (v_1, v_2) \to v_1} \textsc{Fst} \qquad \frac{}{\text{snd } (v_1, v_2) \to v_2} \textsc{Snd}$$

Statics

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2} \textsc{Pair} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } t : \tau_1} \textsc{Fst} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } t : \tau_2} \textsc{Snd}$$

Product type

- Can generalize to n-ary record type:

  $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$

  where l_i are field labels (names).
- Record can be either ordered or unordered. Useful to model objects in OOP languages.
- Example: $\{\text{name} : \text{string}, \text{age} : \text{nat}\}$

Sum type

- Similar to a union of two types, but each the variants are tagged
- Encoding errors with option type, e.g. in SML:

```
datatype 'a option = NONE | SOME of 'a

case some_computation of
    NONE => handle_error ()
  | SOME x => use_value x
```

Sum type

Syntax

$$\begin{array}{rcll} t & ::= & \cdots \mid \textsf{inl}\ t \mid \textsf{inr}\ t \mid \textsf{case}\ t_0\ \textsf{of inl}\ x_1 \Rightarrow t_1; \textsf{inr}\ x_2 \Rightarrow t_2 & \textbf{terms} \\ v & ::= & \cdots \mid \textsf{inl}\ v \mid \textsf{inr}\ v & \textbf{values} \\ \tau & ::= & \cdots \mid \tau_1 + \tau_2 & \textbf{types} \\ E & ::= & \cdots \mid \textsf{inl}\ E \mid \textsf{inr}\ E \mid \textsf{case}\ E\ \textsf{of inl}\ x_1 \Rightarrow t_1; \textsf{inr}\ x_2 \Rightarrow t_2 & \textbf{reduction contexts} \end{array}$$

- `inl` t constructs a value of type $\tau_1 + \tau_2$ from a value of type τ_1
- `inr` t constructs a value of type $\tau_1 + \tau_2$ from a value of type τ_2
- `case` to consume a sum type value, need to provide handler for each variant

Sum type

- Example: option type

```
datatype 'a option = NONE | SOME of 'a
```

$\text{option}(\alpha) \triangleq \text{unit} + \alpha$

$\text{NONE} \triangleq \Lambda\alpha.\text{inl}\ () : \text{option}(\alpha)$

$\text{SOME} \triangleq \Lambda\alpha.\lambda v : \alpha.\text{inr}\ v : \forall\alpha.\alpha \to \text{option}(\alpha)$

$x\ \text{safeDiv}\ y \triangleq \lambda x.\lambda y.\text{if0}\ y\ \text{then NONE[nat] else SOME[nat]}(x/y)$

Sum type

Dynamics

$$\frac{}{\textsf{case}\ (\textsf{inl}\ v)\ \textsf{of inl}\ x_1 \Rightarrow t_1; \textsf{inr}\ x_2 \Rightarrow t_2 \rightarrow t_1[x_1 := v]}\ \textsc{Case-inl}$$

$$\frac{}{\textsf{case}\ (\textsf{inr}\ v)\ \textsf{of inl}\ x_1 \Rightarrow t_1; \textsf{inr}\ x_2 \Rightarrow t_2 \rightarrow t_2[x_2 := v]}\ \textsc{Case-inr}$$

Sum type

Statics

$$\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \mathsf{inl}\ t : \tau_1 + \tau_2}\ \textsc{InL} \qquad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \mathsf{inr}\ t : \tau_1 + \tau_2}\ \textsc{InR}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \mathsf{case}\ t_0\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow t_1; \mathsf{inr}\ x_2 \Rightarrow t_2 : \tau}\ \textsc{Case}$$

Sum type

- Can generalize to n-ary labeled variants:

  $\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle$

  where l_i are variant labels (names).
- Injection to one of the variants: $\mathsf{inj}_{l_i}\ t : \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle$

Existential types

- Existential types: $\exists\alpha.\tau$
- Dual to universal types ($\forall$-quantification)

Syntax

$$
\begin{array}{rcll}
t & ::= & \cdots \mid \mathsf{pack}\ \tau_1, t\ \mathsf{as}\ \exists\alpha.\tau_2 \mid \mathsf{unpack}\ \alpha, x = t_1\ \mathsf{in}\ t_2 & \textbf{terms} \\
v & ::= & \cdots \mid \mathsf{pack}\ \tau_1, v\ \mathsf{as}\ \exists\alpha.\tau_2 & \textbf{values} \\
\tau & ::= & \cdots \mid \exists\alpha.\tau & \textbf{types} \\
E & ::= & \cdots \mid \mathsf{pack}\ \tau_1, E\ \mathsf{as}\ \exists\alpha.\tau_2 \mid \mathsf{unpack}\ \alpha, x = E\ \mathsf{in}\ t & \textbf{reduction contexts}
\end{array}
$$

Existential types

- Useful to express abstract data types (ADTs) or module systems (e.g. in SML/OCaml) and to hide implementation details
- Example: a counter ADT (using record type)

$$\text{Counter} \triangleq \exists \alpha.\{\text{init} : \alpha, \text{inc} : \alpha \to \alpha, \text{get} : \alpha \to \text{nat}\}$$

An implementation:

$$\text{Impl} \triangleq \text{pack nat}, \{\text{init} : 0, \text{inc} : \lambda x : \text{nat}.x + 1, \text{get} : \lambda x : \text{nat}.x\} \text{ as Counter}$$

A client:

$$\text{unpack } \alpha, x = \text{Impl in}$$

$$\text{let } c = x.\text{inc}(x.\text{init}) \text{ in } x.\text{get } c$$

Existential types

Dynamics

$$\frac{}{\text{unpack } \alpha, x = (\text{pack } \tau_1, v \text{ as } \exists \beta . \tau_2) \text{ in } t_2 \rightarrow t_2[\alpha := \tau_1, x := v]} \textsc{UnpackPack}$$

Existential types

Statics

$$\frac{\Gamma \vdash t : \tau_1[\alpha := \tau_2]}{\Gamma \vdash \text{pack } \tau_2, t \text{ as } \exists\alpha.\tau_1 : \exists\alpha.\tau_1} \textsc{Pack}$$

$$\frac{\Gamma \vdash t_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \text{unpack } \alpha, x = t_1 \text{ in } t_2 : \tau_2} \textsc{Unpack}$$

Existential types

- We can Church-encode existential types using universal types in System F!

  $\exists \alpha.\tau \triangleq \forall \beta.(\forall \alpha.\tau \to \beta) \to \beta$

  See *Types and Programming Languages (TAPL)*, Chapter 24, Pierce

What else?

Important features in real-world languages we haven't covered:

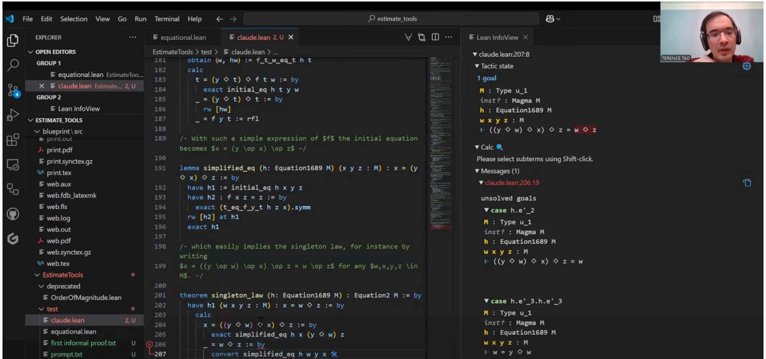
- Recursive type: $\mu\alpha.\tau$
  - Can encode general recursive functions
  - Useful to define inductive data types, e.g. list: $\mu\alpha.\text{unit} + \text{nat} \times \alpha$
- Subtyping: $\tau_1 <: \tau_2$
- Nominal vs structural type systems
- Type checking/inference

Propositions as types

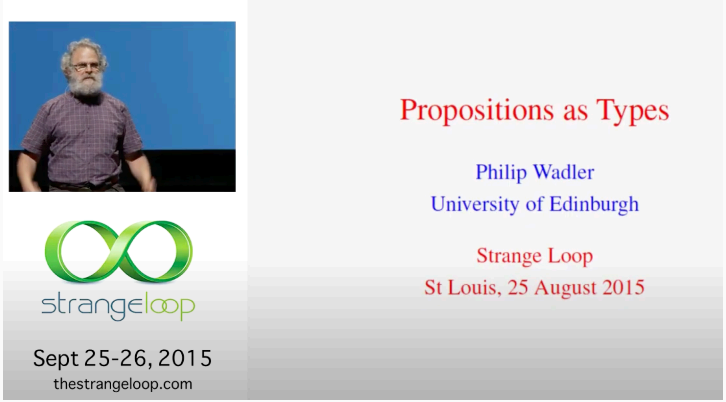
- Curry-Howard Correspondence
- Propositions as types
  - $\top$ as unit type
  - $\bot$ as empty type
  - $A \land B$ as $A \times B$
  - $A \lor B$ as $A + B$
  - $A \implies B$ as $A \to B$
  - $\forall x.P(x)$ as Π-type
  - $\exists x.P(x)$ as Σ-type
- Proofs as programs; proof normalization as program evaluation

Propositions as types

- Very expressive system to formalize mathematics
- Languages (or proof assistants) based on dependent type theory: Coq/Rocq, Lean, Agda, etc.
- Proper mathematicians are using these PLs to write and verify their proofs nowadays!



Propositions as types



Talk: *Propositions as Types* by Philip Wadler

https://www.youtube.com/watch?v=IOiZatlZtGU

Effects

- What is an effect?
- Generally: anything happening during program execution that is not computing a value from its input

Effects

- What is an effect?
- Generally: anything happening during program execution that is not computing a value from its input
- Examples:
  - non-termination
  - read a state, update a state, I/O
  - exceptions/continuations
  - non-determinism
  - etc.

Mutable State

- SML-style mutable references: `ref`, `set`, `get`

Syntax

$$
\begin{array}{lcll}
t & ::= & \cdots \mid \mathsf{ref}\ t \mid \mathsf{set}\ t_1\ t_2 \mid \mathsf{get}\ t \mid l & \textbf{terms} \\
l & \in & \mathsf{Loc} \triangleq \mathbb{N} & \textbf{locations} \\
v & ::= & \cdots \mid l & \textbf{values} \\
\tau & ::= & \cdots \mid \mathsf{ref}\ \tau & \textbf{types} \\
E & ::= & \cdots \mid \mathsf{ref}\ E \mid \mathsf{set}\ E\ t \mid \mathsf{set}\ v\ E \mid \mathsf{get}\ E & \textbf{reduction contexts}
\end{array}
$$

Mutable State

- Need to extend the configuration to include a store (or heap) $\sigma : \mathsf{Loc} \rightharpoonup \mathsf{Val}$ that maps locations to values

Dynamics $(t, \sigma) \to (t', \sigma')$

$$\frac{l \notin \mathsf{dom}(\sigma)}{(\mathsf{ref}\ v, \sigma) \to (l, \sigma[l \mapsto v])}\ \textsc{Ref} \qquad \frac{}{(\mathsf{set}\ l\ v, \sigma) \to ((), \sigma[l \mapsto v])}\ \textsc{Set}$$

$$\frac{\sigma(l) = v}{(\mathsf{get}\ l, \sigma) \to (v, \sigma)}\ \textsc{Get} \qquad \frac{(t, \sigma) \to (t', \sigma')}{(E[t], \sigma) \to (E[t'], \sigma')}\ \textsc{Context}$$

Mutable State

- Need to statically type the store $\Sigma : \mathsf{Loc} \rightharpoonup \mathsf{Type}$ that maps locations to types

Statics $\Sigma, \Gamma \vdash t : \tau$

$$\frac{\Sigma, \Gamma \vdash t : \tau}{\Sigma, \Gamma \vdash \mathsf{ref}\ t : \mathsf{ref}\ \tau}\ \textsc{Ref} \qquad \frac{\Sigma, \Gamma \vdash t_1 : \mathsf{ref}\ \tau \qquad \Sigma, \Gamma \vdash t_2 : \tau}{\Sigma, \Gamma \vdash \mathsf{set}\ t_1\ t_2 : \mathsf{unit}}\ \textsc{Set}$$

$$\frac{\Sigma, \Gamma \vdash t : \mathsf{ref}\ \tau}{\Sigma, \Gamma \vdash \mathsf{get}\ t : \tau}\ \textsc{Get} \qquad \frac{\Sigma(l) = \tau}{\Sigma, \Gamma \vdash l : \mathsf{ref}\ \tau}\ \textsc{Loc}$$

- Question: why do we need to type locations even they cannot appear in the surface syntax?

Next week

- Next Tuesday (Sept 16):

  A Functional Correspondence between Evaluators and Abstract Machines

- Be sure reading the paper before class! Submit a summary on Canvas before the class.

- Next Thursday (Sept 18): paper discussion or lecture?