CS150 APL: Type Systems

Dr. Guannan Wei
guannan.wei@tufts.edu

Sept 9, 2025

Tufts University

Last time

- How to define a programming language?

λ-calculus and operational semantics

Syntax and dynamics

$$
\begin{array}{rcll}
n & \in & \mathbb{N} & \\
t & ::= & n \mid x \mid \lambda x.t \mid t_1\, t_2 \mid t_1 \oplus t_2 & \textbf{terms} \\
E & ::= & \square \mid v\, E \mid E\, t \mid v \oplus E \mid E \oplus t & \textbf{reduction contexts}
\end{array}
$$

$$
\frac{}{(\lambda x.t)\, v \to t[x := v]}\ \beta_v
\qquad
\frac{}{n_1 \oplus n_2 \to n_1 + n_2}\ \textsc{Add}
\qquad
\frac{t_1 \to t_1'}{E[t_1] \to E[t_1']}\ \textsc{Ctx}
$$

λ-calculus and operational semantics

Syntax and dynamics

$$
\begin{array}{rcll}
n & \in & \mathbb{N} & \\
t & ::= & n \mid x \mid \lambda x.t \mid t_1\, t_2 \mid t_1 \oplus t_2 & \textbf{terms} \\
E & ::= & \square \mid v\, E \mid E\, t \mid v \oplus E \mid E \oplus t & \textbf{reduction contexts}
\end{array}
$$

$$
\frac{}{(\lambda x.t)\, v \to t[x := v]}\ \beta_v
\qquad
\frac{}{n_1 \oplus n_2 \to n_1 + n_2}\ \textsc{Add}
\qquad
\frac{t_1 \to t_1'}{E[t_1] \to E[t_1']}\ \textsc{Ctx}
$$

Stuck programs

- Application of a number to another number: $1\ 2$
- Adding a λ-term with a number: $(\lambda x.x) + 3$

Type system: Motivation

Stuck programs

- Application of a number to another number: $1\ 2$
- Adding a λ-term with a number: $(\lambda x.x) + 3$

- There are well-formed (i.e. syntactically valid) programs that do not evaluate to a value according to the dynamics (i.e. they stuck).
- We want to rule out such programs statically (i.e. before running them).

Type system

- Type system: a syntactic discipline to classify the result of terms (i.e. values)
- *Well-typed programs cannot "go wrong".* – Robin Milner, 1978

Simply-typed λ-calculus (STLC)

Syntax

n	$\in$	$\mathbb{N}$	
t	$::=$	$n \mid x \mid \lambda x.t \mid t_1\, t_2 \mid t_1 \oplus t_2$	**terms**
v	$::=$	$n \mid \lambda x.t$	**values**
E	$::=$	$\square \mid v\, E \mid E\, t \mid v \oplus E \mid E \oplus t$	**reduction contexts**
τ	$::=$	$\texttt{nat} \mid \tau_1 \to \tau_2$	**types**
Γ	$::=$	$\cdot \mid \Gamma, x : \tau$	**type environment**

Simply-typed λ-calculus (STLC)

Syntax

$$
\begin{array}{lcll}
\tau & ::= & \texttt{nat} \mid \tau_1 \to \tau_2 & \textbf{types} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau & \textbf{type environment}
\end{array}
$$

Statics

$$
\frac{}{\Gamma \vdash n : \texttt{nat}} \textsc{Num} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \textsc{Var} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \textsc{Abs}
$$

$$
\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2} \textsc{App} \qquad \frac{\Gamma \vdash t_1 : \texttt{nat} \qquad \Gamma \vdash t_2 : \texttt{nat}}{\Gamma \vdash t_1 \oplus t_2 : \texttt{nat}} \textsc{Add}
$$

Simply-typed λ-calculus (STLC)

- Example

$$\dfrac{\dfrac{\dfrac{\Gamma = f : \tau \to \tau,\ x : \tau \quad \Gamma(f) = \tau \to \tau}{f : \tau \to \tau,\ x : \tau \vdash f : \tau \to \tau}\,\textsc{Var} \quad \dfrac{\Gamma = f : \tau \to \tau,\ x : \tau \quad \Gamma(x) = \tau}{f : \tau \to \tau,\ x : \tau \vdash x : \tau}\,\textsc{Var}}{f : \tau \to \tau,\ x : \tau \vdash f\,x : \tau}\,\textsc{App}}{\dfrac{f : \tau \to \tau \vdash \lambda x.\, f\,x : \tau \to \tau}{\vdash \lambda f.\, \lambda x.\, f\,x : (\tau \to \tau) \to \tau \to \tau}\,\textsc{Abs}}\,\textsc{Abs}$$

- Annotation for argument type: $\lambda x : \tau . t$ (Church-style) vs $\lambda x . t$ (Curry-style)

Simply-typed λ-calculus (STLC)

- Example: $(\lambda x.\lambda y.x \oplus y)\ 42$

Soundness of STLC

Progress

If $\Gamma \vdash t : \tau$, then either t is a value or there exists a term t_2 such that $t \to t_2$.

Preservation

If $\Gamma \vdash t : \tau$ and $t \to t'$, then $\Gamma \vdash t' : \tau$.

Significance

- In real programming languages, it means ruling out certain runtime errors.
- Behaviors that are *not* defined by the dynamic semantics (aka UB, undefined behavior), such as dereferencing a dangling pointer, etc.

Incompleteness

Typically, type systems are incomplete: Not all well-behaved programs are well-typed.

Consider extension to STLC:

Syntax, dynamics, and statics

$$t \quad ::= \quad \cdots \mid \text{if0}\ t_1\ t_2\ t_3 \quad \textbf{terms}$$

$$\ldots \qquad \frac{}{\text{if0}\ 0\ t_2\ t_3 \rightarrow t_2}\ \textsc{If0-then} \qquad \frac{}{\text{if0}\ n\ t_2\ t_3 \rightarrow t_3}\ \textsc{If0-else}$$

$$\frac{\Gamma \vdash t_1 : \texttt{nat} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \text{if0}\ t_1\ t_2\ t_3 : \tau}\ \textsc{If0}$$

- What would be a well-behaved but untypable program?

Simply-typed λ-calculus (STLC) with division

- Can we capture all possible runtime errors with a type system?

Syntax and dynamics

$$t \quad ::= \quad \cdots \mid t_1 \text{ div } t_2 \qquad \textbf{terms}$$

$\ldots$

$$\frac{}{n_1 \text{ div } n_2 \to n_1 / n_2} \text{ Div}$$

Simply-typed λ-calculus (STLC) with division

- Can we capture all possible runtime errors with a type system?

Syntax and dynamics

$$t \quad ::= \quad \cdots \mid t_1 \ \mathsf{div} \ t_2 \qquad \textbf{terms}$$

$$\ldots \qquad \frac{}{n_1 \ \mathsf{div} \ n_2 \rightarrow n_1 / n_2} \ \textsc{Div}$$

Options:

- Enhance the type system to rule out division by zero statically
  - Needs to analyze possible numeric values, possible but would rule out many good programs too
- Change the dynamic semantics to include runtime checks, so that $n/0 \rightarrow err$
  - Add a value representation err checked runtime errors
  - Runtime overhead

Relation to type checking/inference

- Declarative type system
  - Read type judgment as a relation: $(\Gamma, t, \tau) \in \mathfrak{T}$
  - Describe a set of triples (i.e. relation) of type environment, term, and type
  - Can be nondeterministic (i.e. multiple rules can apply)
- Algorithmic type checking/inference
  - Describe an algorithm to decide given a term t if $(\Gamma, t, \tau) \in \mathfrak{T}$
  - Usually syntax-directed, avoiding backtracking
  - Could require type annotations from users to be decidable

Polymorphism

- How to write a generic identity function that works for all types?
- $\vdash \lambda x : \tau . x : \tau \rightarrow \tau$ only defines for a specifc τ (note that τ is a meta variable here).

Polymorphism

- How to write a generic identity function that works for all types?
- $\vdash \lambda x : \tau . x : \tau \to \tau$ only defines for a specifc τ (note that τ is a meta variable here).
- Polymorphism: enables writing generic code that works for values of different types
  - Example: traverse a list but you don't care about the type of elements in the list, such as `map`/`fold` function in functional programming languages
  - Generics in Java, C#, etc.

Polymorphism

- System F (also called the polymorphic λ-calculus) extends STLC with universal types (aka. parametric polymorphism).
- Idea: introduce universal quantification over types.
- Just as we have λ-terms that abstract over terms, we have Λ-terms that abstract over types.

System F

Syntax

$$
\begin{array}{rcll}
n & \in & \mathbb{N} & \\
t & ::= & n \mid x \mid \lambda x : \tau . t \mid t_1\, t_2 \mid t_1 \oplus t_2 \mid \Lambda \alpha . t \mid t\ \tau & \textbf{terms} \\
v & ::= & n \mid \lambda x : \tau . t \mid \Lambda \alpha . t & \textbf{values} \\
E & ::= & \square \mid v\, E \mid E\, t \mid v \oplus E \mid E \oplus t \mid E\ \tau & \textbf{reduction contexts} \\
\tau & ::= & \texttt{nat} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha . \tau & \textbf{types} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha & \textbf{type environment}
\end{array}
$$

System F

Dynamics

$$\ldots \qquad \frac{}{(\lambda x : \tau . t)\, v \to t[x := v]}\ \beta_v \qquad \frac{}{(\Lambda \alpha . t)\, \tau \to t[\alpha := \tau]}\ \beta_\Lambda$$

- Example:

  type application $(\Lambda \alpha . \lambda x : \alpha . x)$ nat $\to (\lambda x : \alpha . x)[\alpha :=$ nat$] \equiv \lambda x :$ nat$.x$

System F

Statics

$$\ldots \qquad \frac{\Gamma, \alpha \vdash t : \tau}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.\tau}\ \textsc{TAbs} \qquad \frac{\Gamma \vdash t : \forall\alpha.\tau_1}{\Gamma \vdash t\ \tau_2 : \tau_1[\alpha := \tau_2]}\ \textsc{TApp}$$

System F

Example: Church-encoded Booleans

- Type of Booleans: $\forall \alpha . \alpha \to \alpha \to \alpha$
- True: $\Lambda \alpha . \lambda t : \alpha . \lambda f : \alpha . t$
- False: $\Lambda \alpha . \lambda t : \alpha . \lambda f : \alpha . f$
- if t_1 then t_2 else $t_3 : X \triangleq t_1 \ X \ t_2 \ t_3$

System F

Example: Church-encoded Booleans

- Example: if true then 1 else 2

  $= (\Lambda\alpha.\lambda t : \alpha.\lambda f : \alpha.t)$ nat 1 2

  $\rightarrow (\lambda t : \text{nat}.\lambda f : \text{nat}.t)\ 1\ 2$

  $\rightarrow (\lambda f : \text{nat}.1)\ 2$

  $\rightarrow 1$

Datatypes

- Unit type
- Product type
- Sum type

Unit type

- Syntax

$$\begin{array}{llll} t & ::= & \cdots \mid () & \textbf{terms} \\ v & ::= & \cdots \mid () & \textbf{values} \\ \tau & ::= & \cdots \mid \texttt{unit} & \textbf{types} \end{array}$$

- Dynamics: no reduction
- Statics:

$$\frac{}{\Gamma \vdash () : \texttt{unit}} \textsc{Unit}$$