Simply-Typed Lambda Calculus

Principles of Programming Languages

CAS CS 320

Lecture 19

Practice Problem

```
let x = 0 in
let g = fun y -> x + 1 in
let x = 1 in
let f = fun y -> g x in
let x = 2 in
f
```

What (closure) does the following expression evaluate to? You don't need to give the derivation.

$$\langle \varnothing , \ldots \rangle \Downarrow \boxed{(\!(\mathcal{E} , \ldots)\!)}$$

Answer

```
let x = 0 in
let g = fun y -> x + 1 in
let x = 1 in
let f = fun y -> g x in
let x = 2 in
f x
```

$$\left(\{ x \mapsto 1, g \mapsto ? \}, \lambda y. g\, x \right)$$

$$\left(\{ x \mapsto 0 \}, \lambda y. x + 1 \right)$$

$$\boxed{\left(\{ x \mapsto 1, g \mapsto \left(\{ x \mapsto 0 \}, \lambda y. x+1 \right) \}, \lambda y. g\, x \right)}$$

Outline

Have a high-level discussion of type theory in general

Introduce and analyze the simply-typed lambda calculus (STLC)

~~Demo an implementation of the STLC~~

Learning Objectives

Give a derivation of a typing judgment in the STLC, both with Curry-style typing and Church-style typing

Given an example of expression that cannot be typed the STLC, but which can still be evaluated to a value

Implement the STLC

Type Theory

What is a Type?

```
let f : int -> int = ...
```

(unknown)

Who knows...

A **type** is an syntactic object that we give to an expression which describes something about its behavior

as opposed to semantics

This description can be used to *restrict* the use of the expression *within* a program

Types help us delineate "well-behaved" programs

ex.

let sort : int list → ~~int sort~~ $\underbrace{\text{int sorted list}}_{\text{dependent type theory}}$

Trade-offs

$$(\lambda x\,.\,xx)(\lambda x\,.\,xx)$$

`lambda term called Ω`

Types are *restrictive*. They tells us what we *can't* do in our programs

Types are *safe*. They make sure we don't do dumb things in our program

The goal is to balance:

» Simplicity/Usability
» Expressivity
» Safety/Theoretical Guarantees

OCaml

```
# let big_omega =
    let little_omega x = x x in
    little_omega little_omega;;
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

The type system of OCaml tells us when we're trying to define an ill-behaved program

But OCaml also has strong *type inference* and *polymorphism* to balance these benefits with better ergonomics

The more expressive, the more complex the the type system, designing programming languages is finding the balance that works for you

Typing Judgments

$$\underbrace{\Gamma}_{\substack{\text{context} \\ \text{static environment}}} \vdash \overbrace{\underbrace{e}_{\text{subject}} : \underbrace{\tau}_{\text{predicate}}}^{\text{typing statement}}$$

<u>This judgment reads:</u>

e has type τ in the context Γ

We say that e is **well-typed** if $\underbrace{\cdot}_{\text{empty context}} \vdash e : \tau$ for some type τ

Most of what type theorists do is come up with rules for deriving typing judgments

ex. type inf.

$$\Gamma \vdash e : \tau \dashv \underbrace{C}_{\text{constraints}}$$

ex. System F

$$\underbrace{\Delta \mid \Gamma}_{\text{multiple contexts}} \vdash e : \tau$$

ex. bidirectional typing

$$\Gamma \vdash e \Rightarrow \tau$$

$$\Gamma \vdash e \Leftarrow \tau$$

What is a Context?

variable decl. / bindings.

BNF grammar

$$\left.\begin{aligned} \Gamma &::= \cdot \mid \Gamma, \overbrace{x : \tau}^{\text{variable decl.}} \\ x &::= \texttt{vars} \\ \tau &::= \texttt{types} \end{aligned}\right\} \text{defined inductively}$$

This depends...

In Theory: A context is an inductively-defined syntactic object, just like a type or a expression

In Practice: A context is a set (or ordered list, in some cases) of **variable declarations** } ex. env in projects or assoc. lists.

(a variable declaration a variable together with a type)

ex. of set-contexts being "weird"

$$\dfrac{\dfrac{}{\{x : int\} \vdash x : int}}{\{x : bool\} \vdash \texttt{fun } x \to x : int \to int}$$

Inference Rules

$$\frac{\overbrace{\Gamma_1 \vdash e_1 : \tau_1}^{\text{premise}} \qquad \cdots \qquad \overbrace{\Gamma_k \vdash e_k : \tau_k}^{\text{premise}}}{\underbrace{\Gamma \vdash e : \tau}_{\text{conclusion}}}$$

Inference rules then tell us when we derive a new typing judgment from old typing judgments

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}$$

An inference rule with no premises is called an **axiom**

axiom:

$$\frac{}{\Gamma \vdash 2 : int}$$

<u>The questions we need to answer:</u>

» How do we know what rules to include? one for each construct.
» How do we know if we've chosen *good* rules?

Simply-Typed Lambda Calculus

Syntax

unit expr

type annotations

```
<e>  ::= () | <v> | <e> <e>
       | fun ( <v> : <ty> ) -> <e>
<ty> ::= unit | <ty> -> <ty>
<v>  ::= a | ... | z
```

types

The syntax is the same as that of the lambda calculus except:

» we include a unit expression

» we have types, which annotate arguments

This is the first time that **types are a part of our syntax**

(later we'll add more things like numbers)

Syntax

unit exps

$$e ::= \bullet \mid x \mid \lambda x^{\tau} . e \mid ee$$

$$\tau ::= \boxed{\top} \mid \tau \rightarrow \tau$$

unit type

$$x ::= \textit{variables}$$

The syntax is the same as that of the lambda calculus except:

- we include a unit expression
- we have types, which annotate arguments

This is the first time that **types are a part of our syntax**

(later we'll add more things like numbers)

if you're interested: Curry-Howard Isom.

Typing

$$\frac{}{\Gamma \vdash \bullet : \top}$$

"unit has type unit"

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

Γ is wellformed

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \underbrace{\lambda x^{\tau} . e}_{\text{function}} : \tau \to \tau'}$$

(abstraction)

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

(application)

These rules enforce that a function can only be applied if we *know* that it's a function

In theory: We need to be careful that are contexts are well-formed...

Γ is WF $\approx$ no repeated vars.

In practice: We will think of our context as as set

Type Annotations?

```
<e>  ::= () | <v> | <e> <e>
       | fun <v> -> <e>
<ty> ::= unit | <ty> -> <ty>
<v>  ::= a | ... | z
```

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x \,.\, e : \tau \to \tau'}$$

Do we have to include the type annotation on function arguments?

No, but it does change the way typing works

Roughly speaking, if we include annotations we're using **Church-style typing.** If we drop annotations, we're using **Curry-style typing**

Church vs. Curry Typing

```
fun x -> x
```

```
fun (x : unit) -> x
```

What is the type of the first expression? How about the second?

In Curry-style typing, the type of an expression is *extrinsic*, the expression is just an expression in the lambda calculus

In Church-style typing, it's *intrinsic*, built into the expression and the semantics

Using Curry-style typing is not the same as having polymorphism

Curry:

$$\cdot \vdash \lambda x.x : T \to T$$

$$\cdot \vdash \lambda x.x : (T \to T) \to (T \to T)$$

both are possible

Church:

$$\cdot \vdash \lambda x^{T}.x : T \to T$$

$$\cdot \vdash \lambda x^{T \to T}.x : (T \to T) \to (T \to T)$$

$$\nvdash \lambda f.(f \bullet, f (\lambda x.x))$$

Uniqueness of Types

Lemma. If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$ then $\tau_1 = \tau_2$

Proof. The rough idea is to do induction *on the derivations themselves* (whoa)

this is why we want inductively def. contexts.

In the simply typed lambda calculus with Church-style typing, every expression has a *unique type*

In particular, the function `type_of` is well-defined

Semantics (Review)

$$\frac{}{\langle \mathscr{E}, \lambda x^{\tau} . e \rangle \Downarrow (\mathscr{E}, \lambda x . e)} \qquad \frac{}{\langle \mathscr{E}, \bullet \rangle \Downarrow \bullet} \qquad \frac{}{\langle \mathscr{E}, x \rangle \Downarrow \mathscr{E}(x)}$$

annotated　　　not annotated

$$\frac{\langle \mathscr{E}, e_1 \rangle \Downarrow (\mathscr{E}', \lambda x . e) \qquad \langle \mathscr{E}, e_2 \rangle \Downarrow v_2 \qquad \langle \mathscr{E}'[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathscr{E}, e_1 e_2 \rangle \Downarrow v}$$

The semantics are basically identical

(we can also consider small-step, or big-step with substitution)

This is part of the point. Type-checking only determines *whether* we go on to evaluate the program (whether it makes sense to)

It doesn't determine **how** we evaluate the program

Example (Curry)

$$\lambda x \,.\, xx$$

$\lambda x . x x$ is <u>not</u> well-typed in STLC

What happens if we try to give a type to the above expression?

$\tau'' = \tau'' \to \tau$

IMPOSSIBLE

$\tau = \tau'' \to \tau$

$\tau' = \tau''$

$$\dfrac{\dfrac{\{x : \tau'\} \vdash x : \tau'' \to \tau \qquad \{x : \tau'\} \vdash x : \tau''}{\{x : \tau'\} \vdash x\,x : \tau}}{\cdot \vdash \lambda x . x\,x : \tau}$$

Example (Church)

$$\lambda x^{\tau} . xx$$

What happens if we try to give a type to the above expression? What should τ be?

$$\dfrac{\times \quad \vdots}{\cdot \vdash \lambda x^{\tau} . \, x x : \tau}$$

$\lambda x^{\tau} . xx$

is <u>not</u> welltyped

in STLC

$$\dfrac{\times \quad \vdots}{\cdot \vdash \lambda x^{\tau \to \tau} . \, x x : \tau}$$

Practice Problem

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . \, e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

$$\cdot \vdash \lambda f^{\top \to \top} . \, \lambda x^{\top} . \, f x : (\, \top \to \top \,) \to \top \to \top$$

Give a derivation for the above judgment.

Answer

$$\cdot \vdash \lambda f^{\top \to \top}.\, \lambda x^{\top}.\, f x : (\top \to \top) \to \top \to \top$$

How do we know if we've defined a "good" programming language?

Type Safety

Big Step for STLC

Theorem. If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

With small-step semantics, we can give a finer-grained analysis:

Small step for STLC

Theorem. If $\cdot \vdash e : \tau$, then

» *(progress)* either e is a value or there is an e' such that $e \longrightarrow e'$ } no stuck, eg. if 2 then 2 else

single step relation

» *(preservation)* If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$

multistep relation

» *(normalization)* there is a value v such that $e \twoheadrightarrow v$

not a part of type saftey

These results are *fundamental*. They tell us that our programming language is well-behaved (it's a "good" programming language)

We will eventually drop normalization *(why?)*

want non-termination

$\langle \phi, (\lambda x^{1}. x)\, \bullet \rangle \Downarrow \bullet$ $\quad r = \bullet$

$\cdot \vdash (\lambda x^{1}. x)\, \bullet : T$

$\cdot \vdash \bullet : T$

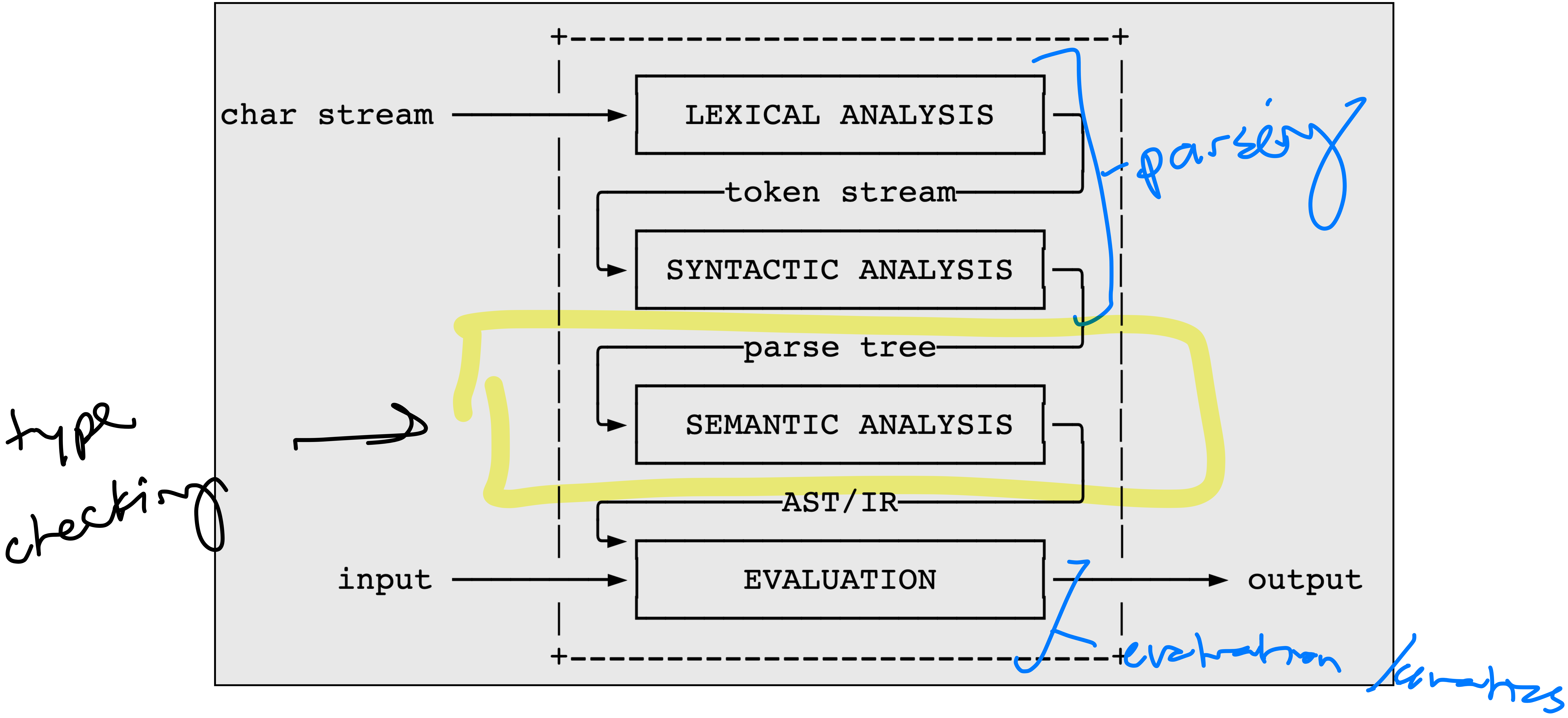
if true then 2 else 3 $\longrightarrow$ 2

int $\qquad$ int

$e \rightarrow e_1 \rightarrow e_2 \rightarrow e \ldots \rightarrow v$

Type Checking

The Picture



char stream
LEXICAL ANALYSIS
token stream
SYNTACTIC ANALYSIS
parse tree
SEMANTIC ANALYSIS
AST/IR
input
EVALUATION
output
parsing
type checking
evaluation
semantics

Type Checking vs. Type Inference

```
type_check : expr -> ty -> bool
```

```
type_of : expr -> ty option
```

over-simplications

Type checking the problem of determining whether a given expression is a given type

Type inference is the problem of *synthesizing* a type for a given expression, if possible

system F → type checking ∈ P
system F → type inf : undecidable

Theoretically, these two problems can be very different

For STLC, they are both easy

One Issue

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

How do we turn this into a type-checking procedure?

It seems like we need to do *some* amount of inference because it's not immediately clear what type we should check e_1 to be

Aside: If you're interested there is a way of *combining* checking and inference in what's called bidirectional type checking

Our solution: We'll just use type inference

```
let rec type_check e =
  match e with
  |
  ⋮
  | App (e1, e2) ->
                                ? -> τ'
    let t1 = type_check e1        ?     in
                                ?
    let t2 = type_check e2        ?     in
    ...
```

General Recursion

```
let rec f x = f x
```

In the mini-projects, we will be implementing *unrestricted recursion*

If we have unrestricted recursion in our language, **it's no longer normalizing** *(why?)*

Again, it's a trade-off

Demo

Demo (Syntax)

```
<e>  ::= () | <v> | <e> <e> | fun ( <v> : <ty> ) -> <e>
       | let <v> : <ty> = <e> in <e>
       | let rec <v> ( <v> : <ty> ) : <ty> = <e> in <e>
       | if <e> then <e> else <e>
       | <e> + <e> | <e> - <e> | <e> * <e> | <e> = <e>
<ty> ::= unit | int | bool | <ty> -> <ty>
<v>  ::= ...
```

This is an extension of our demo from last lecture

(It would be good practice to write down the typing rules for this language)

Practice Problem

```
let rec f (x : t1) : t2 = e1 in e2
```

Write down (to the best of your ability) the typing rule for recursive let-expressions.