Simple Types

Concepts of Programming Languages
Lecture 18

CAS CS 320

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~~

Recap

Recall: The Environment Model

$$\langle \mathcal{E}, e \rangle \Downarrow v$$

Recall: The Environment Model

$$\langle \mathscr{E}, e \rangle \Downarrow v$$

<u>Idea.</u> We keep track of their values in an *environment*

Recall: The Environment Model

$$\langle \mathscr{E}, e \rangle \Downarrow v$$

Idea. We keep track of their values in an *environment*

And evaluate *relative* to the environment, *lazily* filling in variable values along the way

Recall: The Environment Model

$$\langle \mathscr{E}, e \rangle \Downarrow v$$

Idea. We keep track of their values in an *environment*

And evaluate *relative* to the environment, *lazily* filling in variable values along the way

Now the **configurations** in our semantics have nonempty state

Recall: Closures

$$(\mathscr{E}, e)$$

Recall: Closures

$$(\mathscr{E}, e)$$

<u>Definition.</u> A **closure** is an expression together with an environment

Recall: Closures

$$(\mathscr{E}, e)$$

Definition. A **closure** is an expression together with an environment

The environment *captures* bindings which a function needs

Recall: Closures

$$(\mathscr{E}, e)$$

Definition. A **closure** is an expression together with an environment

The environment *captures* bindings which a function needs

Functions need to *remember* what the environment looks like in order to behave correctly according to lexical scoping

Recall: Named Closures

$$(\text{name}, \mathscr{E}, \lambda x \,.\, e)$$

Recall: Named Closures

$$(\text{name}, \mathscr{E}, \lambda x \,.\, e)$$

To implement recursion, we need to be able to *name* closures

Recall: Named Closures

$$(\text{name}, \mathscr{E}, \lambda x \,.\, e)$$

To implement recursion, we need to be able to *name* closures

The idea. Named closures will put themselves into their environment *when they're called*

Recall: Lambda Calculus^{++} (Syntax)

```
<expr> ::= λ<var>.<expr>
         | <var>
         | <expr><expr>
         | let <var> = <expr>
           in <expr>
         | let rec <var> <var> = <expr>
           in <expr>
         | <num>
```

Recall: Lambda Calculus^{++} (Semantics)

Recall: Lambda Calculus^{++} (Semantics)

values and variables

$$\frac{}{\langle \mathcal{E}, \lambda x \,.\, e \rangle \Downarrow (\mathcal{E}, \lambda x \,.\, e)} \qquad \frac{}{\langle \mathcal{E}, n \rangle \Downarrow n} \qquad \frac{\mathcal{E}(x) \neq \bot}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

Recall: Lambda Calculus^{++} (Semantics)

values and variables

$$\frac{}{\langle \mathcal{E}, \lambda x\,.\,e\rangle \Downarrow (\mathcal{E}, \lambda x\,.\,e)} \qquad \frac{}{\langle \mathcal{E}, n\rangle \Downarrow n} \qquad \frac{\mathcal{E}(x) \neq \bot}{\langle \mathcal{E}, x\rangle \Downarrow \mathcal{E}(x)}$$

application (unnamed closure)

$[v_2/x]e$

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

Recall: Lambda Calculus^{++} (Semantics)

values and variables

$$\frac{}{\langle \mathscr{E}, \lambda x\,.\,e\rangle \Downarrow (\mathscr{E}, \lambda x\,.\,e)} \qquad \frac{}{\langle \mathscr{E}, n\rangle \Downarrow n} \qquad \frac{\mathscr{E}(x) \neq \bot}{\langle \mathscr{E}, x\rangle \Downarrow \mathscr{E}(x)}$$

application (unnamed closure)

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

application (named closure)

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

Recall: Lambda Calculus^{++} (Semantics)

values and variables

$$\frac{}{\langle \mathscr{E}, \lambda x \,.\, e \rangle \Downarrow (\mathscr{E}, \lambda x \,.\, e)} \qquad \frac{}{\langle \mathscr{E}, n \rangle \Downarrow n} \qquad \frac{\mathscr{E}(x) \neq \bot}{\langle \mathscr{E}, x \rangle \Downarrow \mathscr{E}(x)}$$

application (unnamed closure)

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

application (named closure)

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

let expressions

$$\frac{\langle \mathscr{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathscr{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathscr{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2} \qquad \boxed{\frac{\langle \mathscr{E}[f \mapsto (f, \mathscr{E}, \lambda x \,.\, e_1)], e_2 \rangle \Downarrow v_2}{\langle \mathscr{E}, \text{let rec } f\ x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}}$$

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

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
```

$\emptyset$

$\{x \mapsto 0\}$

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

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

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

demo

Type Theory

What is a Type?

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

What is a Type?

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

Who knows...

What is a Type?

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

Who knows...

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

What is a Type?

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

Who knows...

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

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

What is a Type?

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

Who knows...

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

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

Types help us delineate "well-behaved" programs

Trade-offs

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

`lambda term called Ω`

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

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

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

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

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

The goal is to balance:

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

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

The goal is to balance:

» Simplicity/Usability

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

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

The goal is to balance:

» Simplicity/Usability
» Expressivity

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

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

<u>The goal is to balance:</u>

- 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
```

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

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 (these are topics for mini-project 3)

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 (these are topics for mini-project 3)

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

Recall: Typing Judgments

$$\Gamma \vdash e : \tau$$

This judgment reads:

e *has type* τ *in the context* Γ

We say that e is **well-typed** if $\cdot \vdash e : \tau$ for some type τ

Recall: Typing Judgments

$$\Gamma \vdash e : \tau$$

This judgment reads:

e has type τ in the context Γ

We say that e is **well-typed** if $\cdot \vdash e : \tau$ for some type τ

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

Recall: Contexts

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

$$x ::= \texttt{vars}$$

$$\tau ::= \texttt{types}$$

Recall: Contexts

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$
$$x ::= \texttt{vars}$$
$$\tau ::= \texttt{types}$$

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

Recall: Contexts

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$
$$x ::= \texttt{vars}$$
$$\tau ::= \texttt{types}$$

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**

Recall: Contexts

empty $\emptyset$

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$
$$x ::= \texttt{vars}$$
$$\tau ::= \texttt{types}$$

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**

(a variable declaration is a variable together with a type)

Recall: Inference Rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

Recall: Inference Rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

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

Recall: Inference Rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

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

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

Recall: Inference Rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

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

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

» How do we know what rules to include?

Recall: Inference Rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

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

The questions we need to answer:

» How do we know what rules to include?
» How do we know if we've chosen *good* rules?

Simply-Typed Lambda Calculus

STLC Syntax

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

STLC Syntax

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

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

STLC Syntax

unit

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

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

» we include a unit expression

STLC Syntax

type annotations

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

fun(x : unit) -> unit

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

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

STLC Syntax

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

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**

Syntax

unit type annot.

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

unit ty

$$\tau ::= \top \mid \tau \to \tau$$

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

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**

STLC Typing

STLC Typing

$$\frac{}{\Gamma \vdash \bullet : \top}\;\textbf{unit}$$

STLC Typing

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

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

STLC Typing

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

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

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

STLC Typing

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

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

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

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

STLC Typing

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

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

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

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

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

Type Annotations?

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

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

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?

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

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

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

Aside: Church vs. Curry Typing

```
fun x -> x

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

Aside: Church vs. Curry Typing

```
fun x -> x

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

: unit → unit ?
: int → int ?

: unit → unit

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

Aside: 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

Aside: 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

Aside: 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

Uniqueness of Types

Uniqueness of Types

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

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)

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)

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

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)

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

STLC Semantics (Review)

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

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

STLC Semantics (Review)

$$\frac{}{\langle \mathcal{E}, \lambda x^{\tau} . e \rangle \Downarrow (\mathcal{E}, \lambda x . e)}\ \textbf{fun} \qquad \frac{}{\langle \mathcal{E}, \bullet \rangle \Downarrow \bullet}\ \textbf{unit} \qquad \frac{}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}\ \textbf{variable}$$

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

The semantics are identical

STLC Semantics (Review)

$$\frac{}{\langle \mathcal{E}, \lambda x^{\tau} . e \rangle \Downarrow (\mathcal{E}, \lambda x . e)}\ \textbf{fun} \qquad \frac{}{\langle \mathcal{E}, \bullet \rangle \Downarrow \bullet}\ \textbf{unit} \qquad \frac{}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}\ \textbf{variable}$$

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

The semantics are <u>identical</u>

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

STLC Semantics (Review)

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

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

The semantics are <u>identical</u>

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 (Church)

$\Omega = (\lambda x. xx)(\lambda x. xx)$

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

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

$\tau = \tau_1 \to \tau_2 \quad \tau_1 = \tau$

$\tau = \tau \to \tau_2$

$\tau = \tau_1 \to \tau_2$

$= (\tau_1 \to \tau_2) \to \tau_2$

$\tau = \tau_1$

$= ((\tau_1 \to \tau_2) \to \tau_2) \to \tau_2 = \dots$

$$\dfrac{\dfrac{\dfrac{}{x : \tau \vdash x : \tau} \qquad \dfrac{}{x : \tau \vdash x : \tau}}{\cancel{x : \tau \vdash x\, x : ?}}}{\emptyset \vdash \lambda x^{\tau} . x\, x : ?}$$

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

Type Safety

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

Type Safety

> **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:

Type Safety

> **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:

> **Theorem.** If $\cdot \vdash e : \tau$, then
>
> » *(progress)* either e is a value or there is an e' such that $e \longrightarrow e'$
>
> » *(preservation)* If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$

Type Safety

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

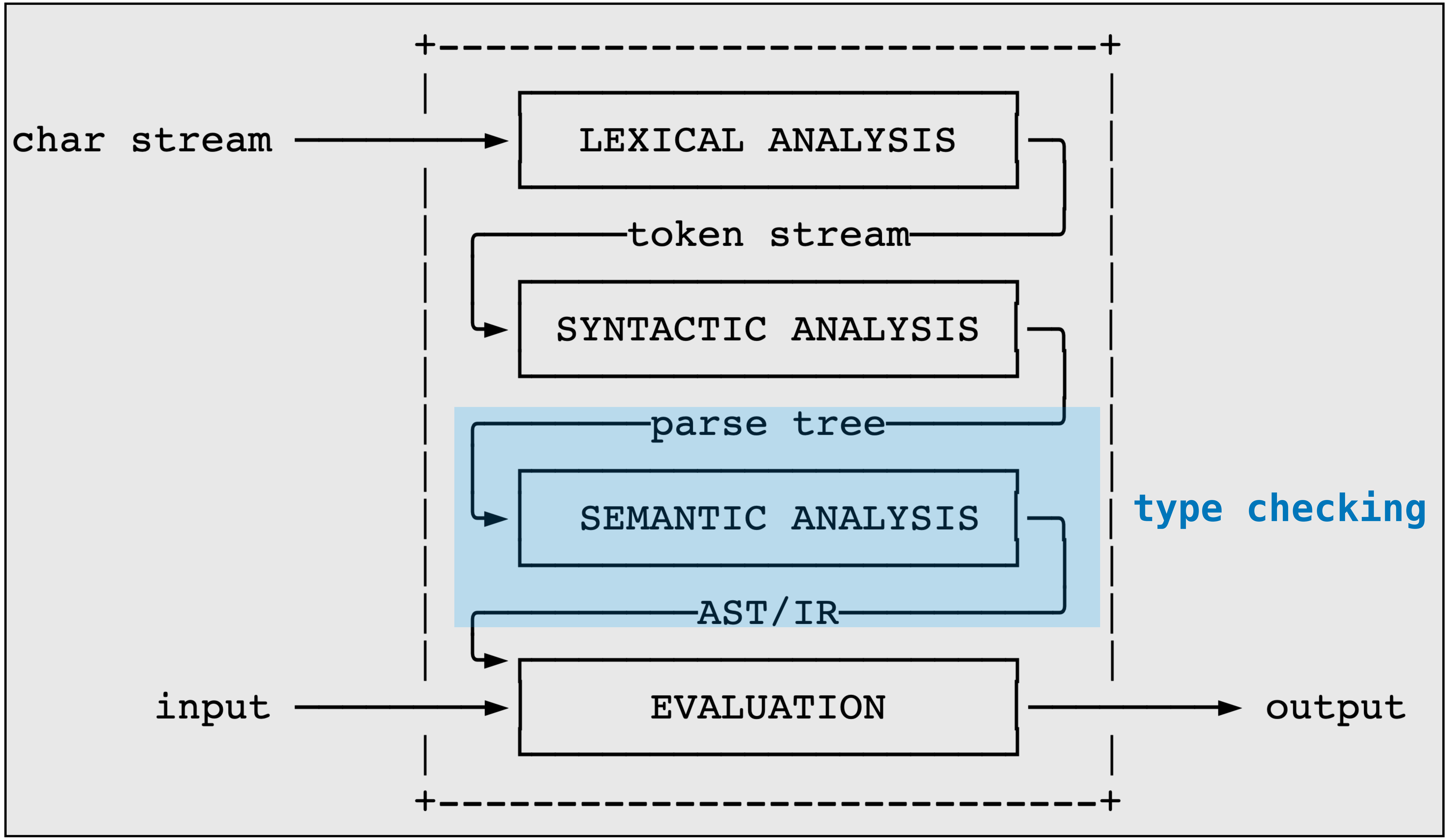
With small-step semantics, we can give a finer-grained analysis:

> **Theorem.** If $\cdot \vdash e : \tau$, then
>
> (never gets stuck)
>
> » *(progress)* either e is a value or there is an e' such that $e \longrightarrow e'$
>
> » *(preservation)* If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$

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

Type Checking

The Picture


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

Type Checking vs. Type Inference

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

type_of : expr -> ty option
```

Type Checking vs. Type Inference

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

type_of : expr -> ty option
```

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

Type Checking vs. Type Inference

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

type_of : expr -> ty option
```

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

Type Checking vs. Type Inference

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

type_of : expr -> ty option
```

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

Theoretically, these two problems can be very different

Type Checking vs. Type Inference

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

type_of : expr -> ty option
```

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

Theoretically, these two problems can be very different

For STLC, they are both easy

The One Issue

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

The 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?

The 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

The 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

The 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 <u>bidirectional type checking</u>

Our solution: We'll just use type inference demo

Summary

Type systems delineate well-behaved expressions

Type inference can sometimes be easier to implement