



# Simply Typed Lambda Calculus

---





# Lecture Outline

---

- Applied lambda calculus
- Introduction to types and type systems
- The simply typed lambda calculus (**System  $F_1$** )
- Syntax
- Dynamic semantics
- Static semantics
- Type safety



# Applied Lambda Calculus (from Sethi)

- $E ::= c \mid x \mid (\lambda x. E_1) \mid (E_1 E_2)$

Augments the pure lambda calculus with **constants**.  
An applied lambda calculus defines its set of constants and reduction rules. For example:

Constants:

**if, true, false**

(all these are  $\lambda$  terms,

e.g.,  $\text{true} = \lambda x. \lambda y. x$ )

**0, iszero, pred, succ**

Reduction rules:

**if true**  $M N \rightarrow_{\delta} M$

**if false**  $M N \rightarrow_{\delta} N$

**iszero 0**  $\rightarrow_{\delta} \text{true}$

**iszero (succ<sup>k</sup> 0)**  $\rightarrow_{\delta} \text{false}$ ,  $k > 0$

**iszero (pred<sup>k</sup> 0)**  $\rightarrow_{\delta} \text{false}$ ,  $k > 0$

**succ (pred M)**  $\rightarrow_{\delta} M$

**pred (succ M)**  $\rightarrow_{\delta} M$



# From an Applied Lambda Calculus to a Functional Language

Construct	Applied $\lambda$ -Calculus	A Language (ML)
Variable	$x$	$x$
Constant	$c$	$c$
Application	$M\ N$	$M\ N$
Abstraction	$\lambda x.M$	<b>fun</b> $x \Rightarrow M$
Integer	$\text{succ}^k\ 0, k > 0$ $\text{pred}^k\ 0, k > 0$	$k$ $-k$
Conditional	<b>if</b> $P\ M\ N$	<b>if</b> $P$ <b>then</b> $M$ <b>else</b> $N$
Let	$(\lambda x.M)\ N$	<b>let val</b> $x = N$ <b>in</b> $M$ <b>end</b>



# The Fixed-Point Operator

- One more constant, and one more rule:

**fix**

**fix**  $M \rightarrow_{\delta} M$  (**fix**  $M$ )

$M(M(M\ldots))$

- Needed to define recursive functions:

**plus**  $x$   $y$  =  $\begin{cases} y & \text{if } x = 0 \\ \text{plus } (\text{pred } x) (\text{succ } y) & \text{otherwise} \end{cases}$

$x-1$        $y+1$

- Therefore:

**plus** =  $\lambda x. \lambda y. \text{if } (\text{iszero } x) y (\text{plus } (\text{pred } x) (\text{succ } y))$



# The Fixed-Point Operator

---

- But how do we define plus?

Define **plus** = **fix** **M**, where

**M** =  $\lambda \mathbf{f}. \lambda \mathbf{x}. \lambda \mathbf{y}. \mathbf{if} \ (\mathbf{iszero} \ \mathbf{x}) \ \mathbf{y} \ (\mathbf{f} \ (\mathbf{pred} \ \mathbf{x}) \ (\mathbf{succ} \ \mathbf{y}))$

We must show that

$\mathbf{fix} \ \mathbf{M} =_{\delta\beta} \lambda \mathbf{x}. \lambda \mathbf{y}. \mathbf{if} \ (\mathbf{iszero} \ \mathbf{x}) \ \mathbf{y} \ ((\mathbf{fix} \ \mathbf{M}) \ (\mathbf{pred} \ \mathbf{x}) \ (\mathbf{succ} \ \mathbf{y}))$



# The Fixed-Point Operator

---

We have to show

$$\mathbf{fix\ M} =_{\delta\beta} \lambda\mathbf{x}.\lambda\mathbf{y}.\mathbf{if\ (iszero\ x)\ y\ ((fix\ M)\ (pred\ x)\ (succ\ y))}$$

$$\begin{aligned} \mathbf{fix\ M} &=_{\delta} \mathbf{M\ (fix\ M)} = \\ &(\lambda\mathbf{f}.\lambda\mathbf{x}.\lambda\mathbf{y}.\mathbf{if\ (iszero\ x)\ y\ (f\ (pred\ x)\ (succ\ y))})\ (\mathbf{fix\ M}) =_{\beta} \\ &\lambda\mathbf{x}.\lambda\mathbf{y}.\mathbf{if\ (iszero\ x)\ y\ ((fix\ M)\ (pred\ x)\ (succ\ y))} \end{aligned}$$





# The Fixed-Point Operator

---

Define **times** =

```
fix ( $\lambda f. \lambda x. \lambda y. \text{if } (\text{iszero } x) \text{ 0 } (\text{plus } y \text{ (f (pred } x) \text{ y))})$ )
```

Exercise: define **factorial** = ?





# The Y Combinator

---

- **fix** is, of course, a lambda expression!
- One possibility, the famous Y combinator:

$$Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Show that **Y M** indeed reduces to **M (Y M)**



# Types!

---

- Constants add power
- But they raise problems because they permit “bad” terms such as
  - **if**  $(\lambda x.x)$  **y** **z** (arbitrary function values are not permitted as first argument, only true/false values)
  - **(0 x)** (0 does not apply as a function)
  - **succ true** (undefined in our language)
  - **plus true 0** etc.



# Types!

---

- Why types?
  - Safety. Catch semantic errors early
  - Data abstraction. Simple types and ADTs
  - Documentation (statically-typed languages only)
    - Type signature is a form of specification!
- Statically typed vs. dynamically typed languages
- Type annotations vs. type inference
- Type safe vs. type unsafe

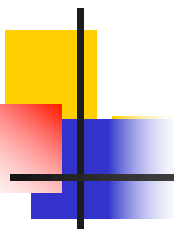


# Type System

---

- Syntax
- Dynamic semantics (i.e., how the program works). In type theory, it is
  - A sequence of reductions
- Static semantics (i.e., typing rules). In type theory, it is defined in terms of
  - Type environment
  - Typing rules, also called type judgments
  - This is typically referred to as the type system

# Example, The Static Semantics. More On This Later!


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

looks up the type of  $x$  in environment  $\Gamma$

(Variable)

$$\frac{\Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1 E_2) : \tau}$$

(Application)

$$\frac{\Gamma, x:\sigma \vdash E_1 : \tau}{\Gamma \vdash (\lambda x:\sigma. E_1) : \sigma \rightarrow \tau}$$

**binding:** augments environment  $\Gamma$  with binding of  $x$  to type  $\sigma$

(Abstraction)



# Type System

---

- A type system either accepts a term (i.e., term is “**well-typed**”), or rejects it
- **Type soundness**, also called **type safety**
  - Well-typed terms never “go wrong”
  - A **sound type system** never accepts a term that can “go wrong”
  - A **complete type system** never rejects a term that cannot “go wrong”
  - Whether a term can “go wrong” is undecidable
    - Type systems choose **type soundness (i.e., safety)**





# Putting It All Together, Formally

---

- Simply typed lambda calculus (**System  $F_1$** )
- Syntax of the simply typed lambda calculus
- The type system: type expressions, environment, and type judgments
- The dynamic semantics
  - **Stuck states**
- Type soundness theorem: progress and preservation theorem

# Type Expressions

- Introducing type expressions

- $\tau ::= \mathbf{b} \mid \tau \rightarrow \tau$
- A type is a basic type  $\mathbf{b}$  (we will only consider **int** and **bool**, for simplicity), or a function type

- Examples

**int**

**bool**  $\rightarrow$  (**int**  $\rightarrow$  **int**) //  $\rightarrow$  is right-associative, thus  
can write just **bool**  $\rightarrow$  **int**  $\rightarrow$  **int**

- Syntax of simply typed lambda calculus:

- $E ::= x \mid ( \lambda x:\tau. E_1 ) \mid ( E_1 E_2 )$



# Type Environment and Type Judgments

- A term in the simply typed lambda calculus is
  - Type correct i.e., well-typed, or
  - Type incorrect
- The rules that judge type correctness are given in the form of **type judgments** in an **environment**
  - Environment  $\Gamma \vdash E : \tau$  ( $\vdash$  is the turnstile)
  - Read: environment  $\Gamma$  entails that  $E$  has type  $\tau$

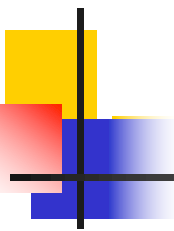
■ Type judgment

$$\frac{\Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1 E_2) : \tau}$$

Premises

Conclusion

# Semantics

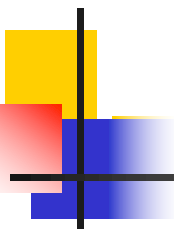

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau}$$

looks up the type of  $x$  in environment  $\Gamma$

(Variable)

$$\frac{\Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1 E_2) : \tau}$$

(Application)


$$\frac{\Gamma, x:\sigma \vdash E_1 : \tau}{\Gamma \vdash (\lambda x:\sigma. E_1) : \sigma \rightarrow \tau}$$

**binding:** augments environment  $\Gamma$  with binding of  $x$  to type  $\sigma$

(Abstraction)



# Examples

---

- Deduce the type for  
 $\lambda x: \text{int}. \lambda y: \text{bool}. x$  in the **nil** environment



# Extensions

---

$$\frac{}{\Gamma \vdash \mathbf{c} : \text{int}}$$
$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 + E_2 : \text{int}}$$
$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 = E_2 : \text{bool}}$$

(Comparison)

$$\frac{\Gamma \vdash b : \text{bool} \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}$$



# Examples

---

- Is this a valid type?

**$\text{Nil} \vdash \lambda x: \text{int}. \lambda y: \text{bool}. x+y : \text{int} \rightarrow \text{bool} \rightarrow \text{int}$**

- No. It gets rightfully rejected. Term reaches a state that goes wrong as it applies **+** on a value of the wrong type (**y** is **bool**, **+** is defined on **ints**)

- Is this a valid type?

**$\text{Nil} \vdash \lambda x: \text{bool}. \lambda y: \text{int}. \text{if } x \text{ then } y \text{ else } y+1 :$   
 **$\text{bool} \rightarrow \text{int} \rightarrow \text{int}$****



# Examples

---

- Can we deduce the type of this term?

**$\lambda f. \lambda x. \text{if } x=1 \text{ then } x \text{ else } (f (f (x-1))) : ?$**

$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 = E_2 : \text{bool}}$$
$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 + E_2 : \text{int}}$$
$$\frac{\Gamma \vdash b : \text{bool} \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}$$



# Examples

---

- Can we deduce the type of this term?

**foldl =**

**$\lambda f. \lambda x. \lambda y. \text{if } x=() \text{ then } y \text{ else } (\text{foldl } f \text{ (cdr } x) (f \ y \ (\text{car } x)))$ :**

$$\frac{\Gamma \vdash E : \text{list } \tau}{\Gamma \vdash \text{car } E : \tau}$$
$$\frac{\Gamma \vdash E : \text{list } \tau}{\Gamma \vdash \text{cdr } E : \text{list } \tau}$$



# Examples

---

- How about this

$(\lambda x. x (\lambda y. y) (x \text{ 1})) (\lambda z. z) : ?$

- $x$  cannot have two “different” types
  - $(x \text{ 1})$  demands  $\text{int} \rightarrow ?$
  - $(x (\lambda y. y))$  demands  $(\tau \rightarrow \tau) \rightarrow ?$
- Program does not reach a “stuck state” but is nevertheless rejected. A sound type system typically rejects some correct programs





# The End

---