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 | x | (λx.E$_1$) | (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 λ terms,

e.g., true=λ**x.**λ**y. x**)

0, iszero, pred, succ

Reduction rules:

if true M N $\rightarrow_\delta$ M

if false M N $\rightarrow_\delta$ N

iszero 0 $\rightarrow_\delta$ true

iszero (succ^k 0) $\rightarrow_\delta$ false, k>0

iszero (pred^k 0) $\rightarrow_\delta$ 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 λ-Calculus	A Language (ML)
Variable	x	x
Constant	c	c
Application	M N	M N
Abstraction	λx.M	fun x => M
Integer	$succ^k$ 0, k>0	k
	$pred^k$ 0, k>0	-k
Conditional	if P M N	if P then M else N
Let	(λx.M) N	let val x = N in M end

The Fixed-Point Operator

- One more constant, and one more rule:

fix $\qquad$ **fix M** $\rightarrow_\delta$ **M (fix M)**

$\boxed{\mathbf{M(M(M\ldots\ ))}}$

- Needed to define recursive functions:

$$\underline{\mathbf{plus}}\ \mathbf{x\ y} = \begin{cases} \mathbf{y} & \text{if } \mathbf{x = 0} \\ \underline{\mathbf{plus}}\ \underbrace{\mathbf{(pred\ x)}}_{\boxed{x-1}}\ \underbrace{\mathbf{(succ\ y)}}_{\boxed{y+1}} & \text{otherwise} \end{cases}$$

- Therefore:

plus = λx.λy. **if** (**iszero** x) y (**plus** (**pred** x) (**succ** y))

The Fixed-Point Operator

- But how do we define **plus**?

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

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

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

fix M $=_{\delta\beta}$

λx.λy. if (iszero x) y ((fix M) (pred x) (succ y))

fix M $=_\delta$ **M (fix M) =**

(λf. λx.λy. if (iszero x) y (f (pred x) (succ y))) (fix M) $=_\beta$

λx.λy. if (iszero x) y ((fix M) (pred x) (succ y))

The Fixed-Point Operator

Define **times** =

fix (λf.λx.λy. if (iszero x) 0 (plus y (f (pred x) y)))

Exercise: define **factorial** = ?

The Y Combinator

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

Y = λf. (λx. f (x x)) (λx. 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** **(λx.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!

looks up the type of **x** in environment **Γ**

$$\frac{x{:}\tau \in \Gamma}{\Gamma \mid\!- \; x : \tau} \qquad \text{(Variable)}$$

$$\frac{\Gamma \mid\!- E_1 : \sigma \rightarrow \tau \quad \Gamma \mid\!- E_2 : \sigma}{\Gamma \mid\!- (E_1\ E_2) : \tau} \qquad \text{(Application)}$$

binding: augments environment **Γ** with binding of **x** to type **σ**

$$\frac{\Gamma, x{:}\sigma \mid\!- E_1 : \tau}{\Gamma \mid\!- (\lambda x{:}\sigma.\ E_1) : \sigma \rightarrow \tau} \qquad \text{(Abstraction)}$$

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

  int

  bool → (int → int) // → is right-associative, thus can write just **bool → int → 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 $\Gamma \mid\!\!- E : \tau$ (|- is the turnstile)
  - Read: environment Γ entails that E has type τ
  - Type judgment

$$\frac{\Gamma \mid\!\!- E_1 : \sigma \rightarrow \tau \qquad \Gamma \mid\!\!- E_2 : \sigma}{\Gamma \mid\!\!- (E_1\ E_2) : \tau}$$

Premises

Conclusion

Semantics

looks up the type of **x** in environment **Γ**

$$\frac{x{:}\tau \in \Gamma}{\Gamma \mid\!- \; x : \tau}$$

(Variable)

$$\frac{\Gamma \mid\!- \; E_1 : \sigma \to \tau \qquad \Gamma \mid\!- \; E_2 : \sigma}{\Gamma \mid\!- \; (E_1\ E_2) : \tau}$$

(Application)

binding: augments environment **Γ** with binding of **x** to type **σ**

$$\frac{\Gamma, x{:}\sigma \;\mid\!- \; E_1 : \tau}{\Gamma \mid\!- \; (\lambda x{:}\sigma.\ E_1) : \sigma \to \tau}$$

(Abstraction)

Examples

- Deduce the type for

  λ**x: int.**λ**y: bool. x** in the **nil** environment

Extensions

$$\frac{}{\Gamma \mid\!- \; c : int}$$

$$\frac{\Gamma \mid\!- E_1 : int \qquad \Gamma \mid\!- E_2 : int}{\Gamma \mid\!- E_1 + E_2 : int}$$

$$\frac{\Gamma \mid\!- E_1 : int \qquad \Gamma \mid\!- E_2 : int}{\Gamma \mid\!- E_1 = E_2 : bool}$$

(Comparison)

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

Examples

- Is this a valid type?

  Nil |- λx: int.λy: bool. x+y : int → bool → 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?

  Nil |- λx: bool.λy: int. if x then y else y+1 :

  bool → int → int

Examples

- Can we deduce the type of this term?

λf. λx. if x=1 then x else (f (f (x-1))) : ?

$$\frac{\Gamma \mid\!- E_1 : \text{int} \qquad \Gamma \mid\!- E_2 : \text{int}}{\Gamma \mid\!- E_1 = E_2 : \text{bool}}$$

$$\frac{\Gamma \mid\!- E_1 : \text{int} \qquad \Gamma \mid\!- E_2 : \text{int}}{\Gamma \mid\!- E_1 + E_2 : \text{int}}$$

$$\frac{\Gamma \mid\!- b : \text{bool} \qquad \Gamma \mid\!- E_1 : \tau \qquad \Gamma \mid\!- E_2 : \tau}{\Gamma \mid\!- \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}$$

Examples

- Can we deduce the type of this term?

foldl =

λf. λx. λy. if x=() then y else (foldl f (cdr x) (f y (car x))):

$$\frac{\Gamma \mid\!- E : \text{list}\ \tau}{\Gamma \mid\!- \text{car}\ E : \tau} \qquad \frac{\Gamma \mid\!- E : \text{list}\ \tau}{\Gamma \mid\!- \text{cdr}\ E : \text{list}\ \tau}$$

Examples

- How about this

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

- **x** cannot have two “different” types
  - **(x** 1**)** demands $\mathbf{int} \rightarrow ?$
  - **(x (**λ**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