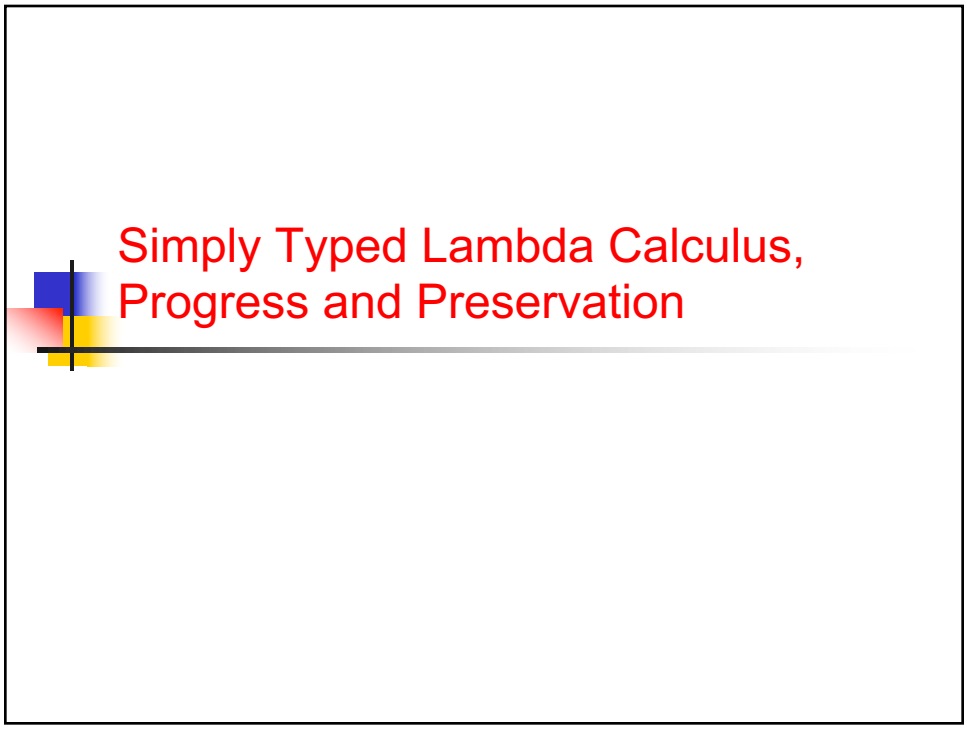
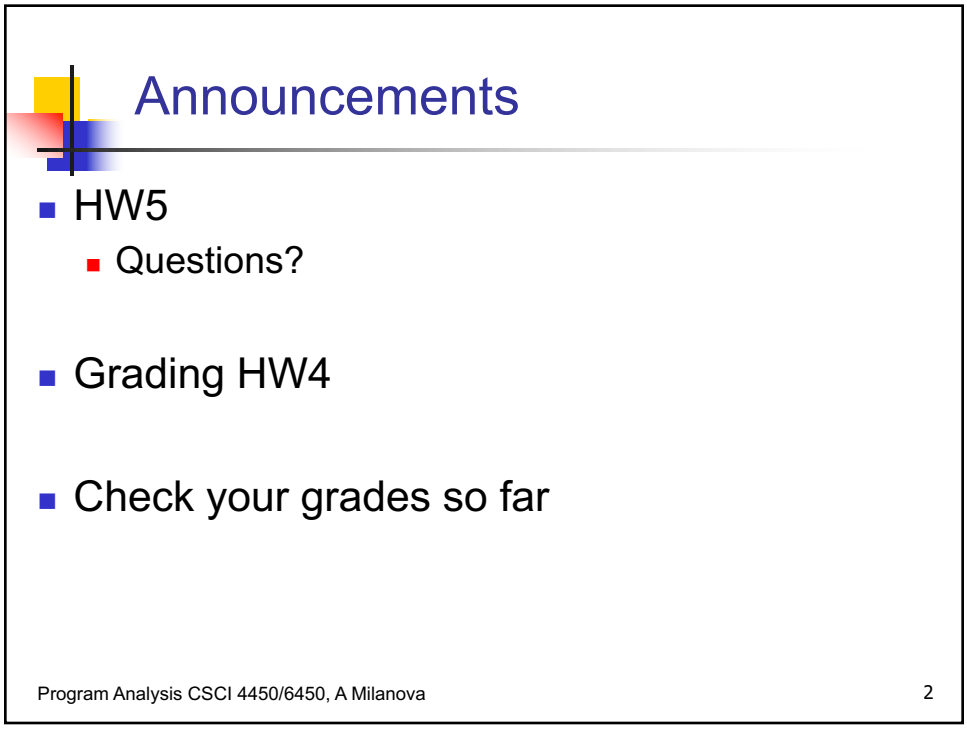

Simply Typed Lambda Calculus,
Progress and Preservation

1


Announcements
HW5
Questions?
Grading HW4
Check your grades so far
Program Analysis CSCI 4450/6450, A Milanova
2

2

Outline

- Applied lambda calculus
- Introduction to types and type systems

- Simply typed lambda calculus (System F_1)
  - Syntax
  - Static semantics
  - Dynamic semantics (next time)
  - Type safety (next time)

Reading

- "Types and Programming Languages" by Benjamin Pierce, Chapters 8 and 9

- Lecture notes based on Pierce and notes by Dan Grossman, UW

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=**λx.λy. 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 **fix M** $\rightarrow_\delta$ **M (fix M)**

  M(M(M…))

- Needed to define recursive functions:

  plus x y = { **y** if **x = 0**; **plus (pred x) (succ y)** otherwise

  x-1 y+1

- 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

M = λf. λx.λy. **if (iszero x) y (f (pred x) (succ y))**

Then show that

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

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))

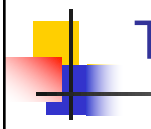
- Show that **Y M** indeed reduces into **M (Y M)**:

$Y\,M \rightarrow^{?} M\,(Y\,M)$

$(\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)))\,M \rightarrow_{\beta}$

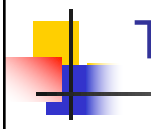
$(\lambda x.\,M\,(x\,x))\,(\lambda x.\,M\,(x\,x)) \rightarrow$

$M\,((\lambda x.\,M\,(x\,x))\,(\lambda x.\,M\,(x\,x))) \equiv M\,(Y\,M)$



Types!

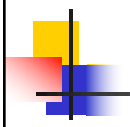
- Constants add power
- But they raise problems because they permit "bad" terms such as
  - **if (λx.x) y z** (arbitrary function values are not permitted as predicates, only true/false values)
  - **(0 x)** (0 does not apply as a function)
  - **succ true** (undefined in our language)
  - **plus true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



Types!

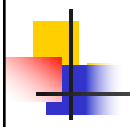
- Why types?
  - Safety. Catch semantic errors early
  - Data abstraction. Simple types and ADTs
  - Documentation (statically-typed languages only)
    - Type signature is a form of specification!
- Statically typed vs. dynamically typed languages
- Type annotations vs. type inference
- Type safe vs.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

True + 5



Types!

- Important subarea of programming languages and program analysis
- Related to abstract interpretation, although…
  - AI is framework of choice for reasoning about **imperative languages**
  - Type systems is framework of choice for reasoning about **functional languages**



Type System

- Syntax PL syntax
- Dynamic semantics (aka concrete semantics!). In type theory, it is
  - A sequence of reductions $E \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots$
- Static semantics (aka abstract semantics!). 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**

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

$\Gamma = [x:\tau_1, y:\tau_2, z:\tau_3]$

$\Gamma = [x:int, y:int \to int]$

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

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

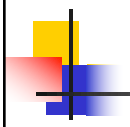
$$\frac{\Gamma, x{:}\sigma \vdash E_1 : \tau}{\Gamma \vdash (\lambda x{:}\sigma.\ E_1) : \sigma \to \tau}$$ (Abstraction)

$[] \vdash \lambda x:int.\lambda y:bool.x$

Type System

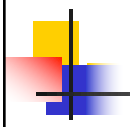
- A type system either accepts a term (i.e., term is well-typed), or rejects it
- Type soundness, also called type safety
  - Well-typed terms never "go wrong" $E \to E_1 \to E_2$
  - More concretely: well-typed terms never reach a stuck state (a "bad" term) during evaluation
    - A programming language defines its own set of stuck states

True + 5



Stuck States

- A term is “stuck” if it cannot be further reduced, and it is not a value
  - E.g., **0 x**
- In real programming languages stuck states correspond to forbidden errors which is execution of operation on illegal arguments
- We will define stuck states formally for the simply typed lambda calculus, in just awhile



Stuck States Examples

- E.g., **c (λx.x),** where **c** is an **int** constant, is a stuck state, i.e., a meaningless state
- E.g., **if c E_1 E_2** where **c** is an **int** constant, is a stuck state
  - Clearly not a value and clearly no rule applies!
  - Because the evaluation rules for **if-then-else** are

  if true E_1 E_2 $\rightarrow_\delta$ E_1

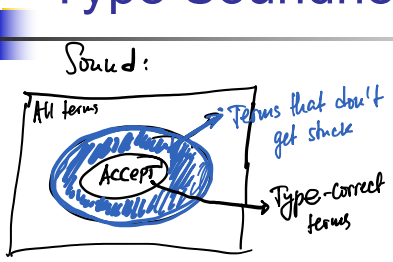
  if false E_1 E_2 $\rightarrow_\delta$ E_2

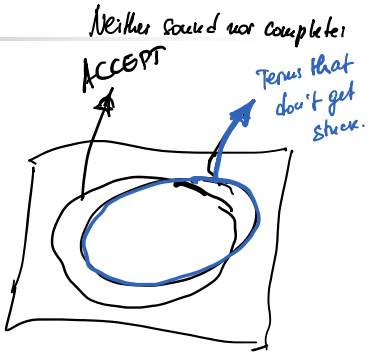
Type Soundness

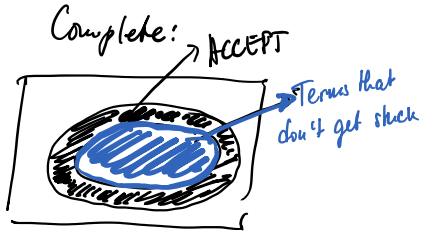
- Remember, a type system either accepts a term **M** or rejects **M**
- A sound type system never accepts a term that can get stuck
- A complete type system never rejects a term that cannot get stuck
- Typically, whether a term gets stuck is undecidable
  - Type systems choose type soundness

19

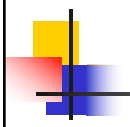
Type Soundness





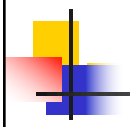


20



Safety = Progress + Preservation

- Progress: A well-typed term is not stuck (i.e., either it is a value, or there is an evaluation step that applies)
- Preservation: If a well-typed term takes a step of evaluation, then the resulting term is well-typed
- Soundness follows:
  - Each state reached by program is well-typed (by Preservation)
  - A well-typed state is not stuck (by Progress)
  - Thus, each state reached by the program is not stuck



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
  - Progress and preservation theorem

Type Expressions

- Syntax of simply typed lambda calculus:
  - **E ::= x | (λx: τ. E_1) | (E_1 E_2) | c**
- Introducing type expressions
  - **τ ::= b | τ → τ**
  - A type is a basic type **b** (we will only consider **int**, for simplicity), or a function type
- Examples

  int, int → int, int → (int → int)
  (int → int) → int

  int

  int → (int → int) // → is right-associative, thus can write just **int → int → int**

23

23

Type Environment and Type Judgments

- A term in the simply typed lambda calculus is
  - Type correct i.e., well-typed, or
  - Type incorrect

  $\Gamma = [x:\tau_1, y:\tau_2, z:\tau_3]$
- The rules that judge type correctness are given in the form of type judgments in an environment
  - Environment **Γ ⊢ E : τ** (⊢ is the turnstile)
  - Read: environment **Γ** entails that **E** has type **τ**
  - Type judgment

$$\frac{\Gamma \vdash E_1 : \sigma \to \tau \qquad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1\ E_2) : \tau}$$

Premises → above the fraction; Conclusion → below the fraction

24

Semantics

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

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

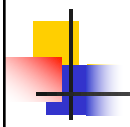
$$\frac{\Gamma \vdash E_1 : \sigma \to \tau \quad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1\ E_2) : \tau}$$ (Application)

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

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

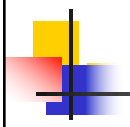
Examples

- Deduce the type for
  λx: int.λy: bool. x in the **nil** environment



Examples

- Deduce the type for
  λ**x: int.**λ**y: bool. x** in the **nil** environment

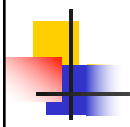


Extensions (of Language and Static Semantics)

$$\frac{}{\Gamma \vdash c : int} \qquad \frac{\Gamma \vdash E_1 : int \quad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 + E_2 : int}$$

$$\frac{\Gamma \vdash E_1 : int \quad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 = E_2 : bool}$$ (Comparison)

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



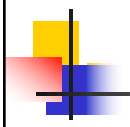
Examples

- Is this a valid type?

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

- Is this a valid type?

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



Examples

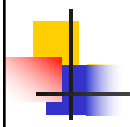
- Can we deduce the type of this term?

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

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 = E_2 : bool}$$

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 + E_2 : int}$$

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



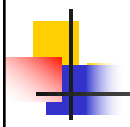
Examples

- How about this **(λx. x (λy. y) (x 1)) (λz. z) : ?**

- **x** cannot have two "different" types
  - **(x 1)** demands **int → ?**
  - **(x (λy. y))** demands **(τ → τ) → ?**
- Program does not reach a "stuck state" but is nevertheless rejected. A sound type system typically rejects some correct programs

31

31



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
  - Progress and preservation theorem

32