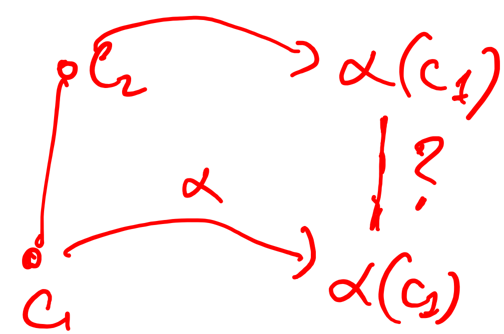
Simply Typed Lambda Calculus, Progress and Preservation

Announcements

- HWS on Submitty
  - Questions?
- Grading HW4
- Check your Rainbow grades



Outline

- Applied lambda calculus
- Introduction to types and type systems

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

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

$\boxed{\textbf{M(M(M...))}}$

- Needed to define recursive functions:

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

$\boxed{\text{x-1}}$ $\quad$ $\boxed{\text{y+1}}$

- Therefore:

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

The Fixed-Point Operator

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

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

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

Then show that

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

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

$(\mathrm{fix}\ M) \to M\,(\mathrm{fix}\ M) =$

$(\lambda f.\ \lambda x.\ \lambda y. \ldots\ )\,(\mathrm{fix}\ M) \to \ldots$

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))

$Y\ M \overset{?}{\longrightarrow} M\ (Y\ M)$

$(\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ M \longrightarrow$

$\underbrace{(\lambda x.\ M\ (x\ x))\ (\lambda x.\ M\ (x\ x))}_{}\longrightarrow \underline{\underline{Y\ M}}$

$\longrightarrow M\ (\underbrace{(\lambda x.\ M\ (x\ x))\ (\lambda x.\ M\ (x\ x))}_{}) = M\ (Y\ M)$

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

plus 2 3 $\longrightarrow$ plus 1 4 $\longrightarrow$ plus 0 5 $\longrightarrow$ 5 ✓

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 True + 5
  - 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

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 $E \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots E_n$
- 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**

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

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

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

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

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

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

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

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"
  - More concretely: well-typed terms never reach a stuck state (a "bad" term) during evaluation
    - We must give a definition of stuck state
    - Each programming language defines its own set of stuck states

True + 5

Stuck States

- Informally, a term is "stuck" if it cannot be further reduced, and it is not a value
  - E.g, **0 x** True + 5

- 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 $E_1\ E_2 \rightarrow_\delta E_1$

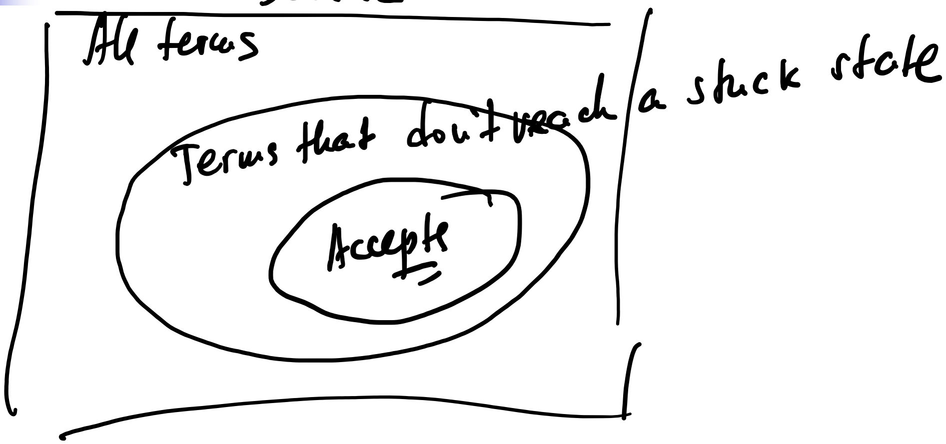
if false $E_1\ E_2 \rightarrow_\delta E_2$

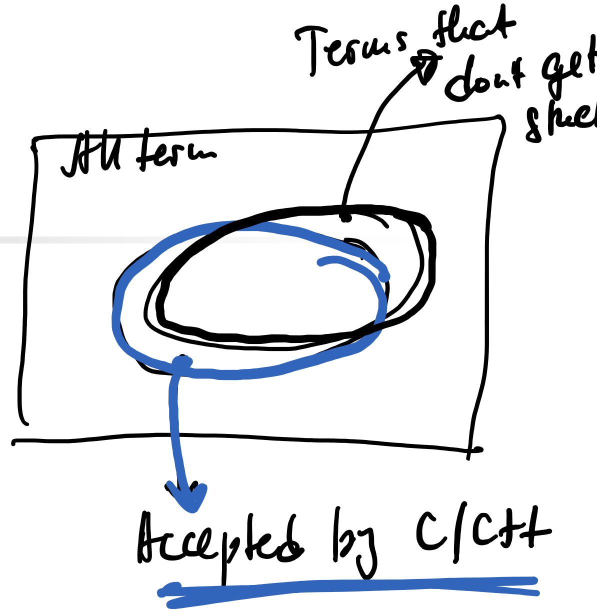
Type Soundness

- Remember, a type system either accepts a term **M** or rejects **M**
- A sound type system never accepts a term that can get stuck $E \to E_1 \to E_2 \to \ldots$
- 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

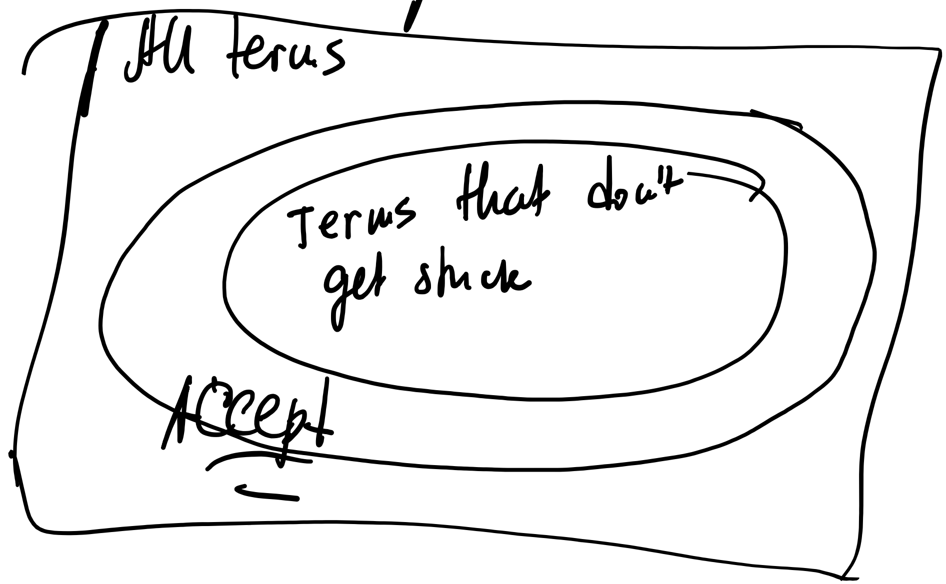
Type Soundness

Sound :





Complete :



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