CS152: Programming Languages

Lecture 9 — Simply Typed Lambda Calculus

Dan Grossman
Spring 2011

Types

Major new topic worthy of several lectures: Type systems

- Continue to use (CBV) Lambda Caluclus as our core model
- But will soon enrich with other common primitives

This lecture:

- Motivation for type systems
- What a type system is designed to do and not do
  - Definition of stuckness, soundness, completeness, etc.
- The Simply-Typed Lambda Calculus
  - A basic and natural type system
  - Starting point for more expressiveness later

Next lecture:

- Prove Simply-Typed Lambda Calculus is sound

Review: L-R CBV Lambda Calculus

$$e ::= \lambda x.\ e \mid x \mid e\ e$$
$$v ::= \lambda x.\ e$$

Implicit systematic renaming of bound variables

- α-equivalence on expressions ("the same term")

$\boxed{e \to e'}$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

$\boxed{e_1[e_2/x] = e_3}$

$$\frac{}{x[e/x] = e} \qquad \frac{y \neq x}{y[e/x] = y} \qquad \frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{(e_1\ e_2)[e/x] = e_1'\ e_2'}$$

$$\frac{e_1[e/x] = e_1' \qquad y \neq x \qquad y \notin FV(e)}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'}$$

Introduction to Types

Naive thought: More powerful PLs are *always* better

- Be Turing Complete (e.g., Lambda Calculus or x86 Assembly)
- Have really flexible features (e.g., lambdas)
- Have conveniences to keep programs short

If this is the *only* metric, types are a step backward

- Whole point is to allow fewer programs
- A "filter" between abstract syntax and compiler/interpreter
  - Fewer programs in language means less for a correct implementation
- So if types are a great idea, they must help with other desirable properties for a PL...

Why types? (Part 1)

1. Catch "simple" mistakes early, even for untested code
    - Example: "if" applied to "mkpair"
    - Even if some too-clever programmer meant to do it
    - Even though decidable type systems must be conservative

Why types? (Part 1)

1. Catch "simple" mistakes early, even for untested code
   - Example: "if" applied to "mkpair"
   - Even if some too-clever programmer meant to do it
   - Even though decidable type systems must be conservative

2. (Safety) Prevent getting stuck (e.g., $x\ v$)
   - Ensure execution never gets to a "meaningless" state
   - But "meaningless" depends on the semantics
   - Each PL typically makes some things type errors (again being conservative) and others run-time errors

Why types? (Part 1)

1. Catch "simple" mistakes early, even for untested code
    - Example: "if" applied to "mkpair"
    - Even if some too-clever programmer meant to do it
    - Even though decidable type systems must be conservative

2. (Safety) Prevent getting stuck (e.g., $x\ v$)
    - Ensure execution never gets to a "meaningless" state
    - But "meaningless" depends on the semantics
    - Each PL typically makes some things type errors (again being conservative) and others run-time errors

3. Enforce encapsulation (an *abstract type*)
    - Clients can't break invariants
    - Clients can't assume an implementation
    - requires safety, meaning no "stuck" states that corrupt run-time (e.g., C/C++)
    - Can enforce encapsulation without static types, but types are a particularly nice way

Why types? (Part 2)

4. Assuming well-typedness allows faster implementations
   - Smaller interfaces enable optimizations
   - Don't have to check for impossible states
   - Orthogonal to safety (e.g., C/C++)

Why types? (Part 2)

4. Assuming well-typedness allows faster implementations
    - Smaller interfaces enable optimizations
    - Don't have to check for impossible states
    - Orthogonal to safety (e.g., C/C++)

5. Syntactic overloading
    - Have symbol lookup depend on operands' types
    - Only modestly interesting semantically
    - Late binding (lookup via *run-time* types) more interesting

Why types? (Part 2)

4. Assuming well-typedness allows faster implementations
   - Smaller interfaces enable optimizations
   - Don't have to check for impossible states
   - Orthogonal to safety (e.g., C/C++)

5. Syntactic overloading
   - Have symbol lookup depend on operands' types
   - Only modestly interesting semantically
   - Late binding (lookup via *run-time* types) more interesting

6. Detect other errors via extensions
   - Often via a "type-and-effect" system
   - Deep similarities in analyses suggest type systems a good way to think-about/define/prove what you're checking
   - Uncaught exceptions, tainted data, non-termination, IO performed, data races, dangling pointers, ...

Why types? (Part 2)

4. Assuming well-typedness allows faster implementations
   - Smaller interfaces enable optimizations
   - Don't have to check for impossible states
   - Orthogonal to safety (e.g., C/C++)

5. Syntactic overloading
   - Have symbol lookup depend on operands' types
   - Only modestly interesting semantically
   - Late binding (lookup via *run-time* types) more interesting

6. Detect other errors via extensions
   - Often via a "type-and-effect" system
   - Deep similarities in analyses suggest type systems a good way to think-about/define/prove what you're checking
   - Uncaught exceptions, tainted data, non-termination, IO performed, data races, dangling pointers, ...

We'll focus on (1), (2), and (3) with a later lecture on (6)

What is a type system?

Er, uh, you know it when you see it. Some clues:

- A decidable (?) judgment for classifying programs
  - E.g., $e_1 + e_2$ has type int if e_1, e_2 have type int (else *no type*)
- A sound (?) abstraction of computation
  - E.g., if $e_1 + e_2$ has type int, then evaluation produces an int (with caveats!))
- Fairly syntax directed
  - Non-example (?): e terminates within 100 steps
- Particularly fuzzy distinctions with *abstract interpretation*
  - Possible topic for a later lecture
  - Often a more natural framework for *flow-sensitive* properties
  - Types often more natural for *higher-order programs*

This is a CS-centric, PL-centric view. Foundational type theory has more rigorous answers

- Later lecture: Typed PLs are like proof systems for logics

Plan for 3ish weeks

- Simply typed $\boldsymbol{\lambda}$ calculus
- (Syntactic) Type Soundness (i.e., safety)
- Extensions (pairs, sums, lists, recursion)

Break for the Curry-Howard isomorphism; continuations; midterm

- Subtyping
- Polymorphic types (generics)
- Recursive types
- Abstract types
- Effect systems

Homework: Adding back mutation
Omitted: Type inference

Adding constants

Enrich the Lambda Calculus with integer constants:

- Not stricly necessary, but makes types seem more natural

$$\begin{aligned} e &::= \lambda x.\ e \mid x \mid e\ e \mid c \\ v &::= \lambda x.\ e \mid c \end{aligned}$$

No new operational-semantics rules since constants are values

We could add $+$ and other *primitives*

- Then we would need new rules (e.g., 3 small-step for $+$)
- Alternately, parameterize "programs" by primitives: $\lambda plus.\ \lambda times.\ \ldots\ e$
  - Like Pervasives in Caml
  - A great way to keep language definitions small

Stuck

Key issue: can a program "get stuck" (reach a "bad" state)?

- Definition: e *is stuck* if e is not a value and there is no e' such that $e \rightarrow e'$
- Definition: e *can get stuck* if there exists an e' such that $e \rightarrow^* e'$ and e' is stuck
  - In a deterministic language, e "gets stuck"

Most people don't appreciate that stuckness depends on the operational semantics

- Inherent given the definitions above

What's stuck?

Given our language, what are the set of stuck expressions?

- Note: Explicitly defining the stuck states is unusual

$$\begin{array}{rcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \\ v & ::= & \lambda x.\ e \mid c \end{array}$$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \quad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \quad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

(Hint: The full set is recursively defined.)

$$S ::=$$

What's stuck?

Given our language, what are the set of stuck expressions?

- Note: Explicitly defining the stuck states is unusual

$$\begin{array}{rcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \\ v & ::= & \lambda x.\ e \mid c \end{array}$$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \quad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \quad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

(Hint: The full set is recursively defined.)

$$S ::= x \mid c\ v \mid S\ e \mid v\ S$$

What's stuck?

Given our language, what are the set of stuck expressions?

- Note: Explicitly defining the stuck states is unusual

$$\begin{array}{lcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \\ v & ::= & \lambda x.\ e \mid c \end{array}$$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

(Hint: The full set is recursively defined.)

$$S ::= x \mid c\ v \mid S\ e \mid v\ S$$

Note: Can have fewer stuck states if we add more rules

- Example: Javascript
- Example: $\frac{}{c\ v \to v}$
- In *unsafe* languages, stuck states can set the computer on fire

Soundness and Completeness

A *type system* is a judgment for classifying programs

- "accepts" a program if some complete derivation gives it a type, else "rejects"

A *sound* type system never accepts a program that can get stuck

- No false negatives

A *complete* type system never rejects a program that can't get stuck

- No false positives

It is typically *undecidable* whether a stuck state can be reachable

- Corollary: If we want an *algorithm* for deciding if a type system accepts a program, then the type system cannot be sound and complete
- We'll choose soundness, try to reduce false positives in practice

Wrong Attempt

$$\tau \quad ::= \quad \textbf{int} \mid \textbf{fn}$$

$$\boxed{\vdash e : \tau}$$

$$\frac{}{\vdash \lambda x.\ e : \textbf{fn}} \qquad \frac{}{\vdash c : \textbf{int}} \qquad \frac{\vdash e_1 : \textbf{fn} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1\ e_2 : \textbf{int}}$$

Wrong Attempt

$$\tau \quad ::= \quad \textsf{int} \mid \textsf{fn}$$

$\boxed{\vdash e : \tau}$

$$\frac{}{\vdash \lambda x.\ e : \textsf{fn}} \qquad \frac{}{\vdash c : \textsf{int}} \qquad \frac{\vdash e_1 : \textsf{fn} \qquad \vdash e_2 : \textsf{int}}{\vdash e_1\ e_2 : \textsf{int}}$$

1. NO: can get stuck, e.g., $(\lambda x.\ y)\ 3$
2. NO: too restrictive, e.g., $(\lambda x.\ x\ 3)\ (\lambda y.\ y)$
3. NO: types not preserved, e.g., $(\lambda x.\ \lambda y.\ y)\ 3$

Getting it right

1. Need to type-check function bodies, which have free variables
2. Need to classify functions using argument and result types

For (1): $\Gamma ::= \cdot \mid \Gamma, x : \tau$ and $\Gamma \vdash e : \tau$

- Require whole program to type-check under empty context $\cdot$

For (2): $\tau ::= \textbf{int} \mid \tau \rightarrow \tau$

- An infinite number of types:
  $\textbf{int} \rightarrow \textbf{int}$, $(\textbf{int} \rightarrow \textbf{int}) \rightarrow \textbf{int}$, $\textbf{int} \rightarrow (\textbf{int} \rightarrow \textbf{int})$, ...

Concrete syntax note: $\rightarrow$ is right-associative, so
$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ is $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$

STLC Type System

$$\begin{array}{rcl} \tau & ::= & \textsf{int} \mid \tau \to \tau \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash c : \textsf{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

The *function-introduction* rule is the interesting one...

A closer look

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

Where did τ_1 come from?

- Our rule "inferred" or "guessed" it
- To be syntax directed, change $\lambda x.\ e$ to $\lambda x : \tau.\ e$ and use that τ

Can think of "adding x" as shadowing or requiring $x \notin \mathrm{Dom}(\Gamma)$

- Systematic renaming (α-conversion) ensures $x \notin \mathrm{Dom}(\Gamma)$ is not a problem

A closer look

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

Is our type system too restrictive?

- That's a matter of opinion
- But it does reject programs that don't get stuck

Example: $(\lambda x.\ (x\ (\lambda y.\ y))\ (x\ 3))\ \lambda z.\ z$

- Does not get stuck: Evaluates to 3
- Does not type-check:
  - There is no τ_1, τ_2 such that $x : \tau_1 \vdash (x\ (\lambda y.\ y))\ (x\ 3) : \tau_2$ because you have to pick *one* type for x

Always restrictive

Whether or not a program "gets stuck" is undecidable:

- If e has no constants or free variables, then $e\ (3\ 4)$ or $e\ x$ gets stuck if and only if e terminates (cf. the halting problem)

Old conclusion: "Strong types for weak minds"

- Need a back door (unchecked casts)

Modern conclusion: Unsafe constructs almost never worth the risk

- Make "false positives" (rejecting safe program) rare enough
  - Have compile-time resources for "fancy" type systems
- Make workarounds for false positives convenient enough

How does STLC measure up?

So far, STLC is sound:

- As language dictators, we decided $c\ v$ and undefined variables were "bad" meaning neither values nor reducible
- Our type system is a conservative checker that an expression will never get stuck

But STLC is far too restrictive:

- In practice, just too often that it prevents safe and natural code reuse
- More fundamentally, it's not even Turing-complete
  - Turns out all (well-typed) programs terminate
  - A good-to-know and useful property, but inappropriate for a general-purpose PL
  - That's okay: We will add more constructs and typing rules

Type Soundness

We will take a *syntactic* (operational) approach to soundness/safety

- The popular way since the early 1990s

Theorem (Type Safety): If $\cdot \vdash e : \tau$ then e diverges or $e \rightarrow^n v$ for an n and v such that $\cdot \vdash v : \tau$

- That is, if $\cdot \vdash e : \tau$, then e cannot get stuck

Proof: Next lecture