CSE 505: Programming Languages

Lecture 12 - STLC

Zach Tatlock

What Are We Doing?

Covered a lot of ground! Easy to lose sight of the big picture.

The Big Picture

Skill Building:

- Defining languages and semantics
  - Grammars and inductive definitions
  - Abstract vs. concrete syntax
- Formal proofs and structural induction
- Invariants, determinism, equivalence
- Lambda calculus
  - Small, simple model of computation

The Big Picture

Skill Building:

- Defining languages and semantics
  - Grammars and inductive definitions
  - Abstract vs. concrete syntax
- Formal proofs and structural induction
- Invariants, determinism, equivalence
- Lambda calculus
  - Small, simple model of computation

Onward and Upward!

- Today: TYPES!
- What do they do?
- Why do we want them?
- How do we formalize them?
- What makes a good type system?

Types

Types are a major new topic worthy of a lifetime's study

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

Types

Types are a major new topic worthy of a lifetime's study

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

Today:

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

Introduction to Types

Aren't more powerful PLs are *always* better?

- Turing Complete so we can do anything ($\boldsymbol{\lambda}$-Calculus / x86)
- Super flexible (e.g., higer order functions)
- Conveniences to keep programs short (concision is king!)

Introduction to Types

Aren't more powerful PLs are *always* better?

- Turing Complete so we can do anything ($\boldsymbol{\lambda}$-Calculus / x86)
- Super flexible (e.g., higer order functions)
- Conveniences to keep programs short (concision is king!)

If so, types are taking us in the wrong direction!

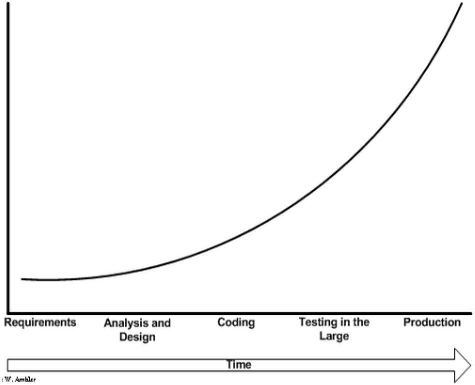
- Type systems restrict which programs we can write :(
- If types are any good, must help in some other PL dimension...

Why types? Let's think about it...

This is a game called "read the professor's mind."

Why types? (Part 1/∞)

1. Catch "stupid" mistakes early, even before testing!
   - Example: "if" applied to "mkpair"
   - Stupid, too-clever occasionally indistinguishable (ptr xor)



: W. Ambler

Why types? (Part 2/∞)

2. (Safety) Prevent getting stuck (e.g., $x\ v$)
    - Ensure execution never gets to a "meaningless" state
    - But "meaningless" depends on the semantics
    - PLs typically have some type errors, others run-time errors

Why types? (Part 2/∞)

2. (Safety) Prevent getting stuck (e.g., $x\ v$)
   - Ensure execution never gets to a "meaningless" state
   - But "meaningless" depends on the semantics
   - PLs typically have some type errors, others run-time errors

You're gonna do it anyway...

> *As our system has grown, a lot of the logic in our Ruby system sort of replicates a type system... I think it may just be a property of large systems in dynamic languages, that eventually you end up rewriting your own type system, and you sort of do it badly. Youre checking for null values all over the place. Theres lots of calls to Rubys kind_of? method, which asks, Is this a kind of User object? Because thats what were expecting. If we dont get that, this is going to explode. It is a shame to have to write all that when there is a solution that has existed in the world of programming languages for decades now.*
>
> *– Alex Payne (Twitter)*

Why types? (Part 3/∞)

3. Help our compiler friends out a bit
   - "filter" between AST and compiler/interpreter
   - Strengthen compiler assumptions, help optimizer
   - Don't have to check for impossible states
   - Orthogonal to safety (e.g. C/C++)

Why types? (Part 3/∞)

3. Help our compiler friends out a bit
   - "filter" between AST and compiler/interpreter
   - Strengthen compiler assumptions, help optimizer
   - Don't have to check for impossible states
   - Orthogonal to safety (e.g. C/C++)

4. Enforce encapsulation (an *abstract type*)
   - Clients can't break invariants
   - Hide implementation (now can change list/pair)
   - 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 3/∞)

3. Help our compiler friends out a bit
    - "filter" between AST and compiler/interpreter
    - Strengthen compiler assumptions, help optimizer
    - Don't have to check for impossible states
    - Orthogonal to safety (e.g. C/C++)

4. Enforce encapsulation (an *abstract type*)
    - Clients can't break invariants
    - Hide implementation (now can change list/pair)
    - 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

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

What is a type system?

What **isn't** a type system?

What **isn't** a type system?

Appel's Axiom:

> *The difference between a program analysis and a type system is that when a type system rejects a program it's the programmer's fault.*

What is a type system?

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

- A decidable (?) judgment for classifying programs
  - E.g., $\mathbf{e_1} + \mathbf{e_2}$ has type int if $\mathbf{e_1}$, $\mathbf{e_2}$ have type int (else *no type*)

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!))

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

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*
  - Often a more natural framework for *flow-sensitive* properties
  - Types often more natural for *higher-order programs*

What is a type system?

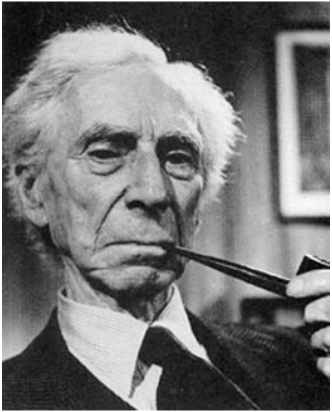
Er, uh, you know it when you see it. Some clues:

- 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*
  - Often a more natural framework for *flow-sensitive* properties
  - Types often more natural for *higher-order programs*

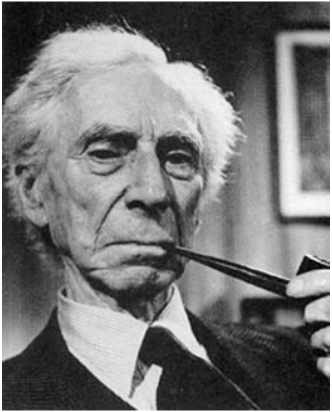
This is a CS-centric, PL-centric view.

- Foundational type theory has more rigorous answers :)
- Type systems have a long history...

Roots in Paradox

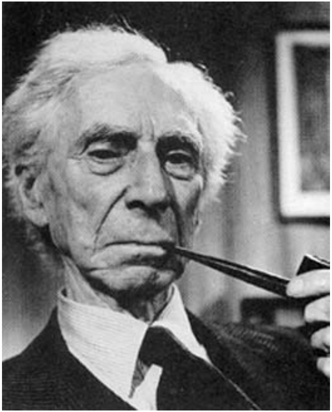


Roots in Paradox



Let $\boldsymbol{R} = \{\boldsymbol{x} \mid \boldsymbol{x} \notin \boldsymbol{x}\}$, then
$\boldsymbol{R} \in \boldsymbol{R} \iff \boldsymbol{R} \notin \boldsymbol{R}$

Roots in Paradox

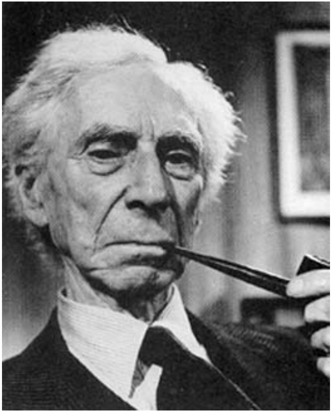


Let $\boldsymbol{R} = \{\boldsymbol{x} \mid \boldsymbol{x} \notin \boldsymbol{x}\}$, then
$\boldsymbol{R} \in \boldsymbol{R} \iff \boldsymbol{R} \notin \boldsymbol{R}$

The barber is a man in a town who shaves exactly those men who do not shave themselves. All men in this town are clean shaven.

Who shaves the barber?

Roots in Paradox



Let $\boldsymbol{R} = \{\boldsymbol{x} \mid \boldsymbol{x} \notin \boldsymbol{x}\}$, then
$\boldsymbol{R} \in \boldsymbol{R} \iff \boldsymbol{R} \notin \boldsymbol{R}$

The barber is a man in a town who shaves exactly those men who do not shave themselves. All men in this town are clean shaven.

Who shaves the barber?

And thus type theory was born ...

Adding constants

Enrich the Lambda Calculus with integer constants:

- Not stricly necessary, but makes types seem more natural

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

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 OCaml
  - 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 $e \to e'$
- Definition: e *can get stuck* if there exists an e' such that $e \to^* e'$ 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

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

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

(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

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

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

(Hint: The full set is recursively defined.)

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

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

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

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

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

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 $x \notin \mathbf{Dom}(\Gamma)$

- Systematic renaming (α-conversion) ensures $x \notin \mathbf{Dom}(\Gamma)$ 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