Types

Major new topic worthy of several lectures: Type systems
  - Continue to use (CBV) Lambda Calculus 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

de ::= \lambda x. e | x | ee

dv ::= \lambda x. e

Implicit systematic renaming of bound variables
  ▶ \alpha\text{-equivalence on expressions ("the same term")}

\[
\begin{align*}
ed &\rightarrow e' \\
(\lambda x. e) v &\rightarrow e[v/x] \\
e_1 e_2 &\rightarrow e'_1 e'_2 \\
v e_2 &\rightarrow v e'_2 \\
e_1[e_2/x] &\equiv e_3 \\
\frac{y \neq x}{y[e/x] = y} \\
\frac{e_1[e/x] = e'_1 \quad e_2[e/x] = e'_2}{(\lambda y. e_1)[e/x] = \lambda y. e'_1}
\end{align*}
\]

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

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++)

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) and maybe (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 \( \text{int} \) if \( e_1, e_2 \) have type \( \text{int} \) (else no type)
  ▶ A sound (?) abstraction of computation
    ▶ E.g., if \( e_1 + e_2 \) has type \( \text{int} \), 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
  ▶ Fairly syntax directed
  ▶ Non-example (?): \( e \) terminates within 100 steps

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 \( \lambda \) 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 strictly necessary, but makes types seem more natural

\[
e ::= \lambda x. e \mid x \mid ee \mid c
\]

\[
v ::= \lambda x. e \mid c
\]

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:
  \( \lambda \text{plus}, \lambda \text{times} \ldots e \)
  ▶ Like Pervasives in OCaml
  ▶ A great way to keep language definitions small

What’s stuck?

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

▶ Note: Explicitly defining the stuck states is unusual

\[
e ::= \lambda x. e \mid x \mid ee \mid c
\]

\[
v ::= \lambda x. e \mid c
\]

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

(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: \( c v \rightarrow v \)
▶ In unsafe languages, stuck states can set the computer on fire

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

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

\[ \tau ::= \text{int} \mid \text{fn} \]

\[ \Gamma \vdash e : \tau \]

\[ \Gamma \vdash \lambda x. e : \text{fn} \]

\[ \Gamma \vdash c : \text{int} \]

\[ \Gamma \vdash e_1 e_2 : \text{int} \]

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): \(\tau ::= \text{int} \mid \tau \rightarrow \tau\)

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

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

---

### A closer look

\[ \Gamma, x : \tau_1 \vdash e : \tau_2 \]

\[ \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \]

Where did \(\tau_1\) come from?

- Our rule “inferred” or “guessed” it
- To be syntax directed, change \(\lambda x. e\) to \(\lambda x : \tau. e\) and use that \(\tau\)

Can think of “adding \(x\)” as shadowing or requiring \(x \not\in \text{Dom}(\Gamma)\)

- Systematic renaming (α-conversion) ensures \(x \not\in \text{Dom}(\Gamma)\) is not a problem

---

### 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 `e 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 \( e \rightarrow^* v \) 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