Programming Language Theory

Simply-Typed Lambda Calculus
Looking back, looking forward

Finished a number of major topics:

▶ **IMP**, modeling assignment and local control flow
▶ Lambda Calculus, modeling scope and functions
▶ Formalizing abstract syntax and operational semantics

Major new topic: Types and Type Systems!

▶ Continue using (CBV) Lambda Calculus as our core model
▶ But will soon enrich with other common primitives
Types and Type Systems!

Worthy of several lectures.

This lecture:

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

Next lecture:

- Prove that the Simply-Typed Lambda Calculus is sound.
Review: Lambda Calculus Syntax

\[ e ::= \lambda x. e \mid x \mid e \; e \]

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

Work with terms “up to renaming of bound variables” (“up to alpha-conversion”).
Review: Lambda Calculus Substitution

Work with terms “up to renaming of bound variables” (“up to alpha-conversion”).

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\lambda x. e) &= FV(e) \setminus \{x\}
\end{align*}
\]

\[e_1[e_2/x] = e_3\]

\[
\begin{align*}
x[e/x] &= e \\
y[e/x] &= y \\
e_1[e/x] &= e_1' \\
e_2[e/x] &= e_2'
\end{align*}
\]

\[
\begin{align*}
(e_1 e_2)[e/x] &= e_1' e_2'
\end{align*}
\]

\[
\begin{align*}
\lambda y. e_1'[e/x] &= \lambda y. e_1'
\end{align*}
\]

Substitution usually treated as a metafunction, not a judgement.
Review: Lambda Calculus Semantics

Small-step, call-by-value (CBV), left-to-right operational semantics:

\[ e \rightarrow_{\text{cbv}} e' \]

\[
(\lambda x. e_1) v_2 \rightarrow_{\text{cbv}} e_1[v_2/x]
\]

\[
e_1 \rightarrow_{\text{cbv}} e_1'
\]

\[
e_1 e_2 \rightarrow_{\text{cbv}} e_1' e_2
\]

\[
e_2 \rightarrow_{\text{cbv}} e_2'
\]

\[
v_1 e_2 \rightarrow_{\text{cbv}} v_1 e_2'
\]

Small-step, call-by-name (CBN) operational semantics:

\[ e \rightarrow_{\text{cbn}} e' \]

\[
(\lambda x. e_1) e_2 \rightarrow_{\text{cbn}} e_1[e_2/x]
\]

\[
e_1 \rightarrow_{\text{cbn}} e_1'
\]

\[
e_1 e_2 \rightarrow_{\text{cbn}} e_1' e_2
\]
Review: Lambda Calculus Semantics

Large-step, call-by-value (CBV) operational semantics:

\[ e \Downarrow_{\text{cbv}} v' \]

\[
\lambda x. e \Downarrow_{\text{cbv}} \lambda x. e
\]

\[
e_1 \Downarrow_{\text{cbv}} \lambda x. e_1' \quad e_2 \Downarrow_{\text{cbv}} v_2 \quad e_1'[v_2/x] \Downarrow_{\text{cbv}} v_3
\]

\[
\lambda x. e_1 e_2 \Downarrow_{\text{cbv}} v_3
\]

Large-step, call-by-name (CBN) operational semantics:

\[ e \Downarrow_{\text{cbn}} v' \]

\[
\lambda x. e \Downarrow_{\text{cbn}} \lambda x. e
\]

\[
e_1 \Downarrow_{\text{cbn}} \lambda x. e_1' \quad e_1'[e_2/x] \Downarrow_{\text{cbn}} v_3
\]

\[
\lambda x. e_1 e_2 \Downarrow_{\text{cbn}} v_3
\]
Introduction to Types

Naïve thought: More powerful PLs are \textit{always} better

- Be Turing Complete
- Have really flexible features (lambda, continuations, \ldots)
- Have conveniences to keep programs short
Introduction to Types

Naïve thought: More powerful PLs are always better

- Be Turing Complete
- Have really flexible features (lambda, continuations, ...)
- Have conveniences to keep programs short

By this metric, types are a step backward

- Whole point is to allow fewer programs
  - (by rejecting some “bad” programs)
- A “filter” between abstract syntax and compiler/interpreter
  - We don’t run all possible programs.
- Why are types a great idea?
  - If types are a great idea, then they must help with other desirable properties of PLs
Why types?

1. Catch “simple” mistakes early (at compile-time)
   ▶ 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., \( x e \))
   ▶ Ensure execution never does a “wrong/bad/meaningless” operation
   ▶ But “wrong/bad/meaningless” depends on the semantics
   ▶ Each PL typically makes some things type errors (compile-time errors) and other things run-time errors

3. Enforce encapsulation (an abstract type)
   ▶ Clients can’t break invariants
   ▶ Clients can’t assume an implementation
   ▶ requires safety, meaning no “wrong/bad/meaningless” operations that corrupt run-time (e.g., C/C++)
   ▶ Can enforce encapsulation without static types (e.g., contracts), but types are a particularly nice way

Continued…
Why types? (continued)

4. Assuming well-typedness allows faster implementations
   ▶ Smaller interfaces enable optimizations
   ▶ Don’t have to check for “wrong/bad/meaningless” 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
   ▶ Uncaught exceptions, IO performed, tainted data,
     dangling pointers, data races, non-termination, …
   ▶ Deep similarities in these analyses suggest that type systems are
     a good way to think-about/define/prove what you’re checking.

We’ll really focus on (1), (2), and (3) (plus (6) if there is time).
What is a type system?

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

▶ A decidable (?) judgement for classifying (accepting/rejecting) programs
  ▶ (e.g., \( e_1 + e_2 \) has type int if \( e_1 \) and \( e_2 \) have type int (else it has no type))

▶ A sound (?) abstraction of computation
  ▶ (e.g., if \( e_1 + e_2 \) has type int, then evaluation yields an int (with caveats!))

▶ Fairly syntax directed
  ▶ (non-e.g., \( e \) 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 are proof systems for logics.
▶ We’ll (briefly) look at the connection in a later lecture.
Plan for next few weeks

- Simply-Typed Lambda Calculus (STLC)
- (Syntactic) Type Soundness (i.e., safety)
- Extensions (pairs, sums, lists, recursion)
- Digression on the Curry-Howard isomorphism
- Digression on evaluation contexts and continuations

break for Midterm Exam (Monday, November 4)

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

Homework(s): Adding back mutation

Omitted: Type inference
Adding constants

Enrich the Lambda Calculus with integer constants:

\[
\begin{align*}
    e & ::= \lambda x. \ e \mid x \mid e\ e \mid c \\
    v & ::= \lambda x. \ e \mid c
\end{align*}
\]

- Not strictly necessary, but will make types seem more natural.
- No need for new operational-semantics rules (since constants are values).
- We could add \(\oplus\), \(\ast\), and other \textit{primitives}
  - Would need new operational-semantics rules (e.g., 3 small-step rules for \(\oplus\)).
- Alternatively, just parameterize “programs” by them: \(\lambda\texttt{plus}.\ \lambda\texttt{times}.\ e\).
  - Like top-level Basis Library functions in Standard ML.
  - A great way to keep language definitions small.
Stuck

Key issue: can a program “get stuck” (reach a “wrong/bad/meaningless” state)?

Definition: $e$ is stuck if

- $e$ is not a value
- and there is no $e'$ such that $e \xrightarrow{\text{cbv}} e'$

Definition $e$ can get stuck if there exists an $e'$ such that

- $e \xrightarrow{*} \text{cbv} e'$
- $e'$ is stuck

(In a deterministic language, $e$ gets stuck.)
Stuck

Key issue: can a program “get stuck” (reach a “wrong/bad/meaningless” state)?

Definition: \( e \) *is stuck* if

- \( e \) is not a value
- and there is no \( e' \) such that \( e \rightarrow_{cbv} e' \)

Definition \( e \) *can get stuck* if there exists an \( e' \) such that

- \( e \rightarrow^*_{cbv} e' \)
- \( e' \) is stuck

(In a deterministic language, \( e \) *gets stuck*.)

Note: “is/gets stuck” depends on the operational semantics. This is inherent in the definitions above (we mention \( e \rightarrow_{cbv} e' \)).
LC+C Stuck

For the Lambda Calculus with Constants, what are the stuck expressions?

Note: Explicitly defining the stuck states is unusual.

\[ e \rightarrow_{cbv} e' \]

\[
\begin{align*}
(\lambda x. e_1) v_2 & \rightarrow_{cbv} e_1[v_2/x] \\
e_1 e_2 & \rightarrow_{cbv} e_1' e_2 \\
v_1 e_2 & \rightarrow_{cbv} v_1 e_2'
\end{align*}
\]

(Hint: The full set of stuck expressions is recursively defined.)

\[ S ::= \]
LC+C Stuck

For the Lambda Calculus with Constants, what are the stuck expressions?

- Note: Explicitly defining the stuck states is unusual.

\[ e \rightarrow_{cbv} e' \]

\[
(\lambda x. e_1) v_2 \rightarrow_{cbv} e_1[v_2/x]
\]

\[
e_1 \rightarrow_{cbv} e'_1
\]

\[
e_1 e_2 \rightarrow_{cbv} e'_1 e_2
\]

\[
e_2 \rightarrow_{cbv} e'_2
\]

\[
v_1 e_2 \rightarrow_{cbv} v'_1 e'_2
\]

(Hint: The full set of stuck expressions is recursively defined.)

\[
S ::= x \mid c v \mid S e \mid v S
\]
Stuckness

Most people don’t realize that “safety” depends on the semantics:

► We can add “cheat” rules to “avoid” being stuck.

\[ c \ v \rightarrow_{\text{cbv}} 0 \quad \quad \quad x \ v \rightarrow_{\text{cbv}} 13 \]

► Unsafe languages don’t “get stuck”, they just “do anything”.

\[ H; e \downarrow c \quad H(x) = \langle c_0, \ldots, c_{n-1} \rangle \quad (0 > c \lor c \geq n) \]

\[ H; x[e] := e' \rightarrow H'; s' \]

► \(H'\) might be the heap that describes the state “computer on fire”.

Matthew Fluet
Programming Language Theory
Lecture 09 16
Soundness and Completeness

A *type system* is a judgement for classifying (accepting or rejecting) programs.
- “accept” program if some complete derivation gives the program a type
- “reject” otherwise (if no complete derivation gives the program a type)

A *sound* type system never accepts a program that can get stuck.
- If the type system accepts a program, then it does not get stuck.
- If a program gets stuck, then the type system rejects it.
- No false negatives.

A *complete* type system never rejects a program that can’t get stuck.
- If a program doesn’t get stuck, then the type system accepts it.
- If the type system rejects a program, then it gets stuck.
- No false positives.
Soundness and Completeness

A type system is a judgement for classifying (accepting or rejecting) programs.
- “accept” program if some complete derivation gives the program a type
- “reject” otherwise (if no complete derivation gives the program a type)

A sound type system never accepts a program that can get stuck.
A complete type system never rejects a program that can’t get stuck.

Typically, it is undecidable whether a program can get stuck.
- If we want an algorithm to decide if a type system accepts a program, then the type system cannot be both sound and complete.
- Be _________ and don’t be _________.
Soundness and Completeness

A type system is a judgement for classifying (accepting or rejecting) programs.
- “accept” program if some complete derivation gives the program a type
- “reject” otherwise (if no complete derivation gives the program a type)

A sound type system never accepts a program that can get stuck.
A complete type system never rejects a program that can’t get stuck.

Typically, it is undecidable whether a program can get stuck.
- If we want an algorithm to decide if a type system accepts a program, then the type system cannot be both sound and complete.
- Be sound and don’t be complete.

Matthew Fluet
Programming Language Theory
Lecture 09 18
Soundness and Completeness

A *type system* is a judgement for classifying (accepting or rejecting) programs.

- “accept” program if some complete derivation gives the program a type
- “reject” otherwise (if no complete derivation gives the program a type)

A *sound* type system never accepts a program that can get stuck.
A *complete* type system never rejects a program that can’t get stuck.

Typically, it is *undecidable* whether a program can get stuck.

- If we want an *algorithm* to decide if a type system accepts a program, then the type system cannot be both sound and complete.
- Be **sound** and don’t be **complete**.
- Try to reduce false positives in practice.
- Full employment theorem for type-system designers.
Wrong Attempt

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

\[ \vdash e : \tau \]

\[ \vdash \lambda x. e : \text{fn} \quad \vdash c : \text{int} \]

\[ \vdash e_1 : \text{fn} \quad \vdash e_2 : \text{int} \]

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

1. NO: unsound
   ▶ because \((\lambda x. y) 3\) is accepted, but gets stuck

2. NO: too restrictive (e.g., disallows function arguments)
   ▶ because \((\lambda x. x) 3 (\lambda y. y)\) is rejected, but does not get stuck

3. NO: types not preserved
   ▶ because \((\lambda x. \lambda y. y) 3\) evaluates to a non-integer
Wrong Attempt

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

\[
\vdash e : \tau
\]

\[
\begin{array}{c}
\vdash \lambda x. e : \text{fn} \\
\vdash c : \text{int}
\end{array}
\]

\[
\begin{array}{c}
\vdash e_1 : \text{fn} \\
\vdash e_2 : \text{int}
\end{array}
\qquad
\vdash e_1 e_2 : \text{int}
\]

1. NO: unsound
   - because \((\lambda x. y) \, 3\) is accepted, but gets stuck

2. NO: too restrictive (e.g., disallows function arguments)
   - because \((\lambda x. x \, 3) \, (\lambda y. y)\) is rejected, but does not get stuck

3. NO: types not preserved
   - because \((\lambda x. \lambda y. y) \, 3\) evaluates to a non-integer
Getting it right

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

For (1): \[ \Gamma ::= \cdot \mid \Gamma, x : \tau \quad \text{and} \quad \Gamma \vdash e : \tau. \]

- Require whole program to type-check under the empty context \( \cdot \)

For (2): \[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]

- (an infinite number of types)
- e.g.s: \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 \( \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3) \).
STLC Type System

\[\begin{align*}
\tau & ::= \text{int} \mid \tau \rightarrow \tau \\
\Gamma & ::= \cdot \mid \Gamma, x:\tau
\end{align*}\]

The function-introduction rule is the interesting one . . .
A closer look

\[ \Gamma, x : \tau_a \vdash e : \tau_r \]

\[ \Gamma \vdash \lambda x. e : \tau_a \rightarrow \tau_r \]

Where did \( \tau_a \) come from?

- Our rule “inferred” or “guessed” it.
- To be (completely) syntax directed, change \( \lambda x. e \) to \( \lambda x : \tau. e \) and use that \( \tau \).
  - Like Java, C, etc., where one must declare the types of fn arguments

Can think of “adding \( x \)” as shadowing or requiring \( x \not\in \text{Dom}(\Gamma) \).
Systematic renaming (\( \alpha \)-conversion) ensures \( x \not\in \text{Dom}(\Gamma) \) is not a problem.
A closer look

\[ \Gamma, x : \tau_a \vdash e : \tau_r \]
\[ \Gamma \vdash \lambda x. e : \tau_a \rightarrow \tau_r \]

Is our type system too restrictive?

- 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.
- But does not type check.
  - There is no \(\tau\) such that \(\cdot \vdash (\lambda x. (x (\lambda y. y)) (x 3)) (\lambda z. z) : \tau\)
  - Need to pick exactly one type for \(x\), but first occurrence wants \((\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})\)
    and second occurrence wants \(\text{int} \rightarrow \text{int}\)
Type systems are 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 iff $e$ terminates.
- A complete type system would need to decide the Halting Problem.

Old conclusion: “Strong types for weak minds”

- need/provide back door (unchecked cast)
Type systems are 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 iff $e$ terminates.

- A complete type system would need to decide the Halting Problem.

Modern conclusion: Unsafe constructs are almost never worth the risk

- Make “false positives” (rejecting safe program) rare enough.
  - Have compile-time resources for “fancy” type systems.

- Make “false negatives” (accepting unsafe program) impossible
  and make alternatives for false positives convenient enough.
Evaluating STLC

Does STLC make false negatives impossible? Yes. So, STLC is sound:

- As language dictators, we deemed *c e* and free variables “bad”
  - neither answers/values nor reducible
- Our type system is a *conservative* checker that an expression will never get stuck

Does STLC make false positives rare? No. So, STLC is incomplete (and far too restrictive):

- In practice: it often prevents safe and natural code reuse
- In theory: it is not even Turing-complete
  - Theorem: All well-typed STLC programs terminate
  - A good-to-know and useful property, but inappropriate for a general-purpose PL
- Nonetheless, a good starting point (we’ll extend with more expressions and typing rules)
Type Soundness

We will take a *syntactic* (operational) approach to soundness/safety
▶ (the popular way since the early 90s)

Theorem (Type Safety): If $\cdot \vdash e : \tau$, then either $e$ diverges or there exists a $v''$ such that $e \rightarrow^{*} \text{cbv} v''$ (and $\cdot \vdash v'' : \tau$).
▶ Alt: If $\cdot \vdash e : \tau$ and $e \rightarrow^{*} \text{cbv} e'$, then $e'$ is not stuck ($e'$ might be a value).

Proof: By induction on (the derivation) $e \rightarrow^{*} \text{cbv} e'$ using the next two lemmas.

Lemma (Preservation): If $\cdot \vdash e : \tau$ and $e \rightarrow \text{cbv} e'$, then $\cdot \vdash e' : \tau$.

Lemma (Progress): If $\cdot \vdash e' : \tau$, then either $e'$ is a value or there exists an $e''$ such that $e' \rightarrow \text{cbv} e''$. 
Type Soundness

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

▶ (the popular way since the early 90s)

Theorem (Type Safety): If \( \cdot \vdash e : \tau \), then either \( e \) diverges or there exists a \( v'' \) such that \( e \rightarrow_{\text{cbv}}^* v'' \) (and \( \cdot \vdash v'' : \tau \)).

▶ Alt: If \( \cdot \vdash e : \tau \) and \( e \rightarrow_{\text{cbv}}^* e' \), then \( e' \) is not stuck (\( e' \) might be a value).

Proof: By induction on (the derivation) \( e \rightarrow_{\text{cbv}}^* e' \) using the next two lemmas.
Type Soundness

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

▶ (the popular way since the early 90s)

Theorem (Type Safety): If \( \cdot \vdash e : \tau \), then either \( e \) diverges or there exists a \( v'' \) such that \( e \rightarrow^{*}_{\text{cbv}} v'' \) (and \( \cdot \vdash v'' : \tau \)).

▶ Alt: If \( \cdot \vdash e : \tau \) and \( e \rightarrow^{*}_{\text{cbv}} e' \), then \( e' \) is not stuck (\( e' \) might be a value).

Proof: By induction on (the derivation) \( e \rightarrow^{*}_{\text{cbv}} e' \) using the next two lemmas.

Lemma (Preservation): If \( \cdot \vdash e : \tau \) and \( e \rightarrow_{\text{cbv}} e' \), then \( \cdot \vdash e' : \tau \).

Lemma (Progress): If \( \cdot \vdash e' : \tau \), then either \( e' \) is a value or there exists an \( e'' \) such that \( e' \rightarrow_{\text{cbv}} e'' \).