CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

Lecture 7— Simply Typed Lambda Calculus

Where we are

- You've starting HW2 due October 28, which is pre-$\boldsymbol{\lambda}$.
- Midterm November 4 in class
- Our CBV $\boldsymbol{\lambda}$ calculus models higher-order functions in languages like ML and Scheme very well.
- But once "not everything is a function" we need some type-checking
- After a couple weeks on types for functional languages, we'll move to object-oriented languages

Why types?

Our *untyped λ-calculus* is universal, like assembly language. But we might want to allow *fewer programs* (whether or not we remain Turing complete):

1. Catch "simple" mistakes (e.g., "if" applied to "mkpair") early (too early? not usually)
2. (Safety) Prevent getting stuck (e.g., $x\ e$) (but for pure λ-calculus, just need to prevent free variables)
3. Enforce encapsulation (an *abstract type*)
   - clients can't break invariants
   - clients can't assume an implementation
   - requires safety

4. Assuming well-typedness allows faster implementations
    - E.g., don't have to encode constants and plus as functions
    - Don't have to check for being stuck
    - orthogonal to safety (e.g., C)
5. Syntactic overloading (not too interesting)
    - "late binding" (via run-time types) very interesting
6. Novel uses in vogue (e.g., prevent data races)

We'll mostly focus on (2)

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 and e_2 have type int else it *has no type*)
- Fairly syntax directed (non-example??: e terminates within 100 steps)
- A sound (?) abstraction of computation (e.g., if $e_1 + e_2$ has type int, then evaluation produces an int (with caveats!))

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

Plan for a couple weeks

- Simply typed λ calculus (STλC)
- (Syntactic) Type Soundness (i.e., safety)
- Extensions (pairs, sums, lists, recursion)
- Termination (coolest proof in the course)
- Type variables ($\forall$, $\exists$, μ)
- References and exceptions (interesting even w/o types)
- Relation to ML (throughout)

And some other cool stuff as time permits...

Adding constants

Let's add integers to our CBV small-step λ-calculus:

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

We could add $+$ and other *primitives* or just paramterize "programs" by them: $\lambda plus.\ e$. (Like Pervasives in OCaml.)

(Could do the same with constants, but there are lots of them)

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

What are the *stuck* states? Why don't we want them?

Wrong Attempt

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

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

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

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

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): $\Gamma ::= \cdot \mid \Gamma, x : \tau$ (a "compile-time heap"??) and $\Gamma \vdash e : \tau$.

For (2): $\tau ::= \textbf{int} \mid \tau \rightarrow \tau$ (an infinite number of types)

E.g.s: $\textbf{int} \rightarrow \textbf{int}$, $(\textbf{int} \rightarrow \textbf{int}) \rightarrow \textbf{int}$, $\textbf{int} \rightarrow (\textbf{int} \rightarrow \textbf{int})$.

Concretely, $\rightarrow$ is right-associative (like term application) $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ is $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

STλC Type System

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

$$\frac{}{\Gamma \vdash c : \textsf{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

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

A closer look

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

1. 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.
2. Can make Γ an abstract *partial function* if $x \notin \mathbf{Dom}(\Gamma)$. Systematic renaming ($\alpha$-conversion) allows it.
3. Still "too restrictive". E.g.: $\lambda x.\ (x\ (\lambda y.\ y))\ (x\ 3)$ applied to $\lambda z.\ z$ does not get stuck.

Always restrictive

"gets stuck" undecidable: If e has no constants or free variables, then $e\ (\mathbf{3}\ \mathbf{4})$ (or $e\ x$) gets stuck iff e terminates.

Old conclusion: "Strong types for weak minds" – need back door (unchecked cast)

Modern conclusion: Make "false positives" (reject safe program) rare and "false negatives" (allow unsafe program) impossible. Be Turing-complete and convenient even when having to "work around" a false positive.

Justification: false negatives too expensive, have resources to use fancy type systems to make "rare" a reality.

Evaluating ST$\boldsymbol{\lambda}$C

1. Does ST$\boldsymbol{\lambda}$C prevent false negatives? Yes.
2. Does ST$\boldsymbol{\lambda}$C make false positives rare? No. (A starting point)

Big note: “Getting stuck” depends on the semantics. If we add $\boldsymbol{c\ v \rightarrow 0}$ and $\boldsymbol{x\ v \rightarrow 42}$ we “don't need” a type system. Or we could say $\boldsymbol{c\ v}$ and $\boldsymbol{x\ v}$ “are values”.

That is, the language dictator deemed $\boldsymbol{c\ e}$ and free variables bad. Our type system is a conservative checker that they won't occur.

Type Soundness

We will take a *syntactic* (operational) approach to soundness/safety (the popular way for almost 10 years)...

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

Proof: By induction on n using the next two lemmas.

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

Lemma (Progress): If $\cdot \vdash e : \tau$, then e is a value or there exists an e' such that $e \rightarrow e'$.

Progress

Lemma: If $\cdot \vdash e : \tau$, then e is a value or there exists an e' such that $e \to e'$.

Proof: We first prove this lemma:

Lemma (Canonical Forms): If $\cdot \vdash v : \tau$, then:

- if τ is **int**, then v is some c
- if τ has the form $\tau_1 \to \tau_2$ then v has the form $\lambda x.\ e$.

Proof: By inspection of the form of values and typing rules.

We now prove Progress by structural induction on e...

Progress continued

The structure of e has one of these forms:

- x — impossible because $\cdot \vdash e : \tau$.
- c or $\lambda x.\ e'$ — then e is a value
- $e_1\ e_2$ — By induction either e_1 is some v_1 or can become some e_1'. If it becomes e_1', then $e_1\ e_2 \rightarrow e_1'\ e_2$. Else by induction either e_2 is some v_2 or can become some e_2'. If to becomes e_2', then $v_1\ e_2 \rightarrow v_1\ e_2'$. Else e is $v_1\ v_2$. *Inverting the assumed typing derivation* ensures $\cdot \vdash v_1 : \tau' \rightarrow \tau$ for some τ'. So Canonical Forms ensures v_1 has the form $\lambda x.\ e'$. So $v_1\ v_2 \rightarrow e'[v_2/x]$.

Note: If we add $+$, we need the other part of Canonical Forms.

Preservation

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

Proof: By induction on the derivation of $\cdot \vdash e : \tau$. Bottom rule could conclude:

- $\cdot \vdash c : \textbf{int}$ or $\cdot \vdash \lambda x.\ e : \tau$ — then $e \rightarrow e'$ is impossible, so lemma holds *vacuously*.
- $\cdot \vdash x : \cdot(x)$ — actually, it can't; $\cdot(x)$ doesn't exist.
- $\cdot \vdash e_1\ e_2 : \tau$ — Then we know $\cdot \vdash e_1 : \tau' \rightarrow \tau$ and $\cdot \vdash e_2 : \tau'$ for some τ'. There are 3 ways to derive $e_1\ e_2 \rightarrow e'$...

Preservation, app case

We have: $\cdot \vdash e_1 : \tau' \rightarrow \tau$, $\cdot \vdash e_2 : \tau'$, and $e_1\ e_2 \rightarrow e'$. We need: $\cdot \vdash e' : \tau$. The derivation of $e_1\ e_2 \rightarrow e'$ ensures 1 of these:

- e' is $e_1'\ e_2$ and $e_1 \rightarrow e_1'$: So with $\cdot \vdash e_1 : \tau' \rightarrow \tau$ and induction, $\cdot \vdash e_1' : \tau' \rightarrow \tau$. So with $\cdot \vdash e_2 : \tau'$ we can derive $\cdot \vdash e_1'\ e_2 : \tau$.
- e' is $e_1\ e_2'$ and $e_2 \rightarrow e_2'$: So with $\cdot \vdash e_2 : \tau'$ and induction, $\cdot \vdash e_2' : \tau'$. So with $\cdot \vdash e_1 : \tau' \rightarrow \tau$ we can derive $\cdot \vdash e_1\ e_2' : \tau$.
- e_1 is some $\lambda x.\ e_3$, e_2 is some v, and e' is $e_3[v/x]$...

App case, β case

Because $\cdot \vdash \lambda x.\ e_3 : \tau' \to \tau$, we know $\cdot, x{:}\tau' \vdash e_3 : \tau$. So with $\cdot, x{:}\tau' \vdash e_3 : \tau$ and $\cdot \vdash e_2 : \tau'$, we need $\cdot \vdash e_3[v/x] : \tau$.

The Substitution Lemma proves a strengthened result (must be stronger to prove the lemma)

Lemma (Substitution): If $\Gamma, x{:}\tau' \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau'$, then $\Gamma \vdash e_1[e_2/x] : \tau$.

Proof: By induction on derivation of $\Gamma, x{:}\tau' \vdash e_1 : \tau$.

Proving Substitution

Bottom rule of $\Gamma, x{:}\tau' \vdash e_1 : \tau$ could conclude (page 1 of 2):

- $\Gamma, x{:}\tau' \vdash c : \textbf{int}$. Then $c[e_2/x] = c$ and $\Gamma \vdash c : \textbf{int}$.
- $\Gamma, x{:}\tau' \vdash y : (\Gamma, x{:}\tau')(y)$. Either $y = x$ or $y \neq x$.
  If $y = x$, then $(\Gamma, x{:}\tau')(x)$ is τ' and $x[e_2/x]$ is e_2.
  So $\Gamma \vdash e_2 : \tau'$ satisfies the lemma.
  If $y \neq x$ then $(\Gamma, x{:}\tau')(y)$ is $\Gamma(y)$ and $y[e_2/x]$ is y.
  So we can derive $\Gamma \vdash y : \Gamma(y)$.
- $\Gamma, x{:}\tau' \vdash e_a\ e_b : \tau$. Then for some τ_a and τ_b,
  $\Gamma, x{:}\tau' \vdash e_a : \tau_a$ and $\Gamma, x{:}\tau' \vdash e_b : \tau_b$.
  So by induction $\Gamma \vdash e_a[e_2/x] : \tau_a$ and $\Gamma \vdash e_b[e_2/x] : \tau_b$.
  So we can derive $\Gamma \vdash e_a[e_2/x]\ e_b[e_2/x] : \tau$.
  And $(e_a\ e_b)[e_2/x]$ is $e_a[e_2/x]\ e_b[e_2/x]$.

Proving Substitution Cont'd

- $\Gamma, x{:}\tau' \vdash \lambda y.\ e_a : \tau$. (We can assume $y \neq x$ and $y \notin \mathbf{Dom}(\Gamma)$.) Then for some τ_a and τ_b, $\Gamma, x{:}\tau', y{:}\tau_a \vdash e_a : \tau_b$ and τ is $\tau_a \rightarrow \tau_b$.
  By an *Exchange Lemma* $\Gamma, y{:}\tau_a, x{:}\tau' \vdash e_a : \tau_b$.
  By a *Weakening Lemma* and $\Gamma \vdash e_2 : \tau'$, we know $\Gamma, y{:}\tau_a \vdash e_2 : \tau'$.
  So by induction (using $\Gamma, y{:}\tau_a$ for Γ (!!)), $\Gamma, y{:}\tau_a \vdash e_a[e_2/x] : \tau_b$.
  So we can derive $\Gamma \vdash \lambda y.\ e_a[e_2/x] : \tau_a \rightarrow \tau_b$.
  And $(\lambda y.\ e_a)[e_2/x]$ is $\lambda y.\ (e_a[e_2/x])$.

Exchange: If $\Gamma, x{:}\tau_1, y{:}\tau_2 \vdash e : \tau$, then $\Gamma, y{:}\tau_2, x{:}\tau_1 \vdash e : \tau$

Weakening: If $\Gamma \vdash e : \tau$, then $\Gamma, x{:}\tau' \vdash e : \tau$ (if $x \notin \mathbf{Dom}(\Gamma)$)

Summary

What may seem a weird lemma pile is a powerful recipe:

Soundness: We don't get stuck because our induction hypothesis (typing) holds (Preservation) and stuck terms aren't well typed (contrapositive of Progress).

Preservation holds by induction on typing (replace subterms with same type) and Substitution (for β-reduction). Substitution must work over open terms and requires Weakening and Exchange.

Progress holds by induction on expressions (or typing) because either a subexpression progresses or we can make a *primitive reduction* (using Canonical Forms).