A (Relatively) Pragmatic Introduction to the Formal Study of Programming Languages

Todd Millstein

CSE505
October 26, 2001

What Language Theory?

Syntax

- What constitutes a well-formed program?
- BNF grammar

Dynamic Semantics

- How is a program evaluated?
- Denotational, axiomatic, operational semantics

Static Semantics

- Which well-formed programs "make sense" (i.e. typecheck)?
- Typing rules, typechecking algorithms

Type Soundness

- What does "make sense" mean?
- Soundness proofs

Why Language Theory?

Elucidates the core ideas of programming languages.

- reduction, values, type errors, type soundness

Clarifies a language design and implementation.

- Which features are primitives, and which are "syntactic sugars"?
- How does this particular weird feature actually work?

Allows rigorous statements to be made about a program.

- Program P is (not) well-formed.
- P will evaluate to value v.
- Certain kinds of errors will not occur when P is run.

Provides a platform for language experimentation.

- Augment an existing language with my favorite construct.
- Augment an existing type system with my favorite kind of type.

It's fun. Really.

How Language Theory?

A pragmatic approach.

- Focus on the core techniques used by language theorists today.
- Give up on traditional topics like domain theory, denotational semantics, and Hoare logic.

Place less emphasis on a particular language, concentrating instead on the (largely language-independent) techniques.

- Goal: Students should be able to read an avarage POPL paper and understand the goals of the work, key concepts, notation.

Relying on your questions, comments, feedback.

How Language Theory?

The λ-calculus

- Intimidating name, simple idea.
- No need to know Greek, derivatives, or integrals.
- Foundation of all functional programming languages.

Dynamic semantics for the λ-calculus

- Encodings of standard language constructs
- Structural operational semantics
- Specifying lazy vs. eager evaluation

Simply-typed λ-calculus

- The core of every type system.
- Simple and intuitive.

Type Soundness for the Simply-typed λ-calculus

Polymorphic Type Systems

λ-calculus: Syntax

e	$::=$	x	variable
		$\lambda x.e$	abstraction (function)
		$e_1\ e_2$	application (function call)

Conventions

- Metavariable x ranges over an infinite set of variable names.
- Metavariable e ranges over expressions (or terms) of the λ-calculus.

Where is the data that we pass to functions?

Some terms

- x
- $\lambda x.x$
- $(\lambda x_1.x_1\ x_1)\ \lambda x_2.x_2$

Free and Bound Variables

The abstraction $\lambda x.e$ binds x in the body of e.

A variable reference x is bound if it appears in the scope of a binder of x. Otherwise the reference is free.

A term is closed if it has no free variable references.

α-renaming

- Bound variables can be renamed without changing a term's "meaning." $\lambda x_1.(x_2\ x_1)$, $\lambda x_3.(x_2\ x_3)$
- Free variables cannot be renamed. $\lambda x_1.(x_2\ x_1)$, $\lambda x_1.(x_3\ x_1)$

Computing in the λ-calculus

The only way to evaluate terms is via function application.

$$(\lambda x.e_1)\ e_2 \longrightarrow [x \mapsto e_2]e_1 \qquad (\beta\text{-reduction})$$

- $e \longrightarrow e'$ means e "evaluates in one step to" e'
- $[x \mapsto e_2]e_1$ means "the term obtained by replacing all free occurrences of x in e_1 with e_2"

$$\begin{array}{lcll}
[x \mapsto e]x & = & e & \\
[x \mapsto e]x' & = & x' & \text{if } x \neq x' \\
[x \mapsto e](\lambda x'.e') & = & \lambda x'.[x \mapsto e]e' & \text{if } x \neq x' \\
 & & & \text{and } x' \text{ not free in } e \\
[x \mapsto e](e_1\ e_2) & = & [x \mapsto e]e_1\ [x \mapsto e]e_2 &
\end{array}$$

Examples

- $[x \mapsto x_0](x(\lambda x_1.(x_1\ x))) = (x_0(\lambda x_1.(x_1\ x_0))$
- $[x \mapsto x_0](x(\lambda x.x)) = (x_0\ [x \mapsto x_0](\lambda x_1.x_1))$
- $[x \mapsto x_0](x(\lambda x_0.(x_0\ x))) = (x_0\ [x \mapsto x_0](\lambda x_1.(x_1\ x)))$

Reduction

A redex is an expression that matches a reduction rule.

- $(\lambda x.e_1)e_2$

Reduce each redex in a term until reaching a term with no redices, which is the "result" of the computation.

- $\underline{(\lambda x.(x\ x))((\lambda x.x)(\lambda x.x))} \longrightarrow$
- $\underline{((\lambda x.x)(\lambda x.x))}((\lambda x.x)(\lambda x.x)) \longrightarrow$
- $\underline{(\lambda x.x)((\lambda x.x)(\lambda x.x))} \longrightarrow$
- $\underline{(\lambda x.x)(\lambda x.x)} \longrightarrow$
- $\lambda x.x$

A term that cannot be reduced further is in normal form.

Reduction Strategies

Normal-order reduction (call-by-name, lazy)

- Reduce the leftmost, outermost redex.
- $\underline{(\lambda x.(x\ x))((\lambda x.x)(\lambda x.x))} \longrightarrow$
- $\underline{((\lambda x.x)(\lambda x.x))}((\lambda x.x)(\lambda x.x)) \longrightarrow$
- $\underline{(\lambda x.x)((\lambda x.x)(\lambda x.x))} \longrightarrow$
- $\underline{(\lambda x.x)(\lambda x.x)} \longrightarrow$
- $\lambda x.x$

Applicative-order reduction (call-by-value, eager)

- Reduce the leftmost, outermost redex whose arg is in normal form.
- $(\lambda x.(x\ x))\underline{((\lambda x.x)(\lambda x.x))} \longrightarrow$
- $\underline{(\lambda x.(x\ x))(\lambda x.x)} \longrightarrow$
- $\underline{(\lambda x.x)(\lambda x.x)} \longrightarrow$
- $\lambda x.x$

Reduction Strategies (cont.)

Let $\stackrel{*}{\longrightarrow}$ be the reflexive, transitive closure of the $\longrightarrow$ relation.

Theorem (Church-Rosser #1): If $e_1 \stackrel{*}{\longrightarrow} e_2$ and $e_1 \stackrel{*}{\longrightarrow} e_3$, then there exists e_4 such that $e_2 \stackrel{*}{\longrightarrow} e_4$ and $e_3 \stackrel{*}{\longrightarrow} e_4$.

Corollary: Each term has a unique normal form (if any).

But not every term has a normal form.

- $(\lambda x.(x\ x))\ (\lambda x.(x\ x)) \longrightarrow (\lambda x.(x\ x))\ (\lambda x.(x\ x))$

Theorem (Church-Rosser #2): If e has a normal form, then the normal-order (lazy) reduction strategy will find it.

- $(\lambda x.(\lambda x_2.x_2))\ ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$

Expressive Power

Believe it or not, the λ-calculus is fully general: Church's thesis is that every "effectively computable" function can be encoded as a λ-term.

Turing showed that every Turing machine can be encoded as a λ-term, and vice versa.

Practical impact: Useful as a platform for language design experimentation.

- See how a new construct works in a fully general setting.
- Caveat: No guarantee the new construct will interact well with other λ-calculus extensions!

What is the λ-calculus analogue of the halting problem?

Multiple Arguments

Simulate multiple arguments to a function via higher-order functions.

$\lambda(x_1, x_2).(x_1\ x_2)$ becomes $\lambda x_1.\lambda x_2.(x_1\ x_2)$

Technique known as currying, after the logician Haskell Curry.

Church Booleans

A boolean value is a choice between two alternatives.

- tru $\equiv \lambda t.\lambda f.t$
- fls $\equiv \lambda t.\lambda f.f$

A conditional "executes" the choice: ifthenelse $\equiv \lambda b.\lambda t.\lambda e.b\ t\ e$

- ifthenelse tru $v\ w \xrightarrow{*}$
- $\underline{\text{tru}\ v}\ w \longrightarrow$
- $(\lambda f.v)\ w \longrightarrow$
- v

and $\equiv \lambda b_1.\lambda b_2.$ifthenelse $b_1\ b_2$ fls $\equiv \lambda b_1.\lambda b_2.b_1\ b_2$ fls

- and tru fls $\xrightarrow{*}$
- $\underline{\text{tru fls}}$ fls $\longrightarrow$
- $\underline{(\lambda f.\text{fls})\ \text{fls}}$
- fls

Church Booleans (cont.)

A boolean value is a choice between two alternatives.

- tru $\equiv \lambda t.\lambda f.t$
- fls $\equiv \lambda t.\lambda f.f$

What would "or" look like?

What would "not" look like?

Are these booleans any less "real" than booleans in traditional programming languages?

- What advantages do these booleans have?
- What disadvantages do they have?

Church Numerals

Define numbers in unary, via "zero" and "successor" (Peano arithmetic).

- zero $\equiv \lambda s.\lambda z.z$
- one $\equiv \lambda s.\lambda z.s\ z$;
- two $\equiv \lambda s.\lambda z.s(s\ z)$;

The successor function just "adds another s".

- succ $\equiv \lambda n.\lambda s.\lambda z.s(n\ s\ z)$
- succ one $\longrightarrow$
- $\lambda s.\lambda z.s(\text{one}\ s\ z) \equiv$
- $\lambda s.\lambda z.s((\lambda s.\lambda z.s\ z)\ s\ z)$

How would "plus" be defined?

Recursion

Surprisingly, recursion can be encoded, without any additional mechanism! It's mind-bending, but here's some intuition:

Start with factorial.

- fact ≡ λ n. if n=0 then 1 else n * fact(n-1)

Replace recursive references with a call to an extra parameter.

- factf ≡ λ f. λ n. if n=0 then 1 else n * f(n-1)

Iteratively define partial factorial functions.

- fact0 ≡ factf $\lambda x.x$
- fact1 ≡ factf fact0
- fact2 ≡ factf fact1
- ...

The function fact∞ is equivalent to fact.

Recursion (cont.)

fact ≡ λ n. if n=0 then 1 else n * fact(n-1)

factf ≡ λ f. λ n. if n=0 then 1 else n * f(n-1)

Let's play a similar trick on f to the one we played on fact.

- factff ≡ λ f. λ n. if n=0 then 1 else n * (f f)(n-1)

Alternatively, let's make the change "non-invasively."

- factff ≡ λ f. factf (f f)

Now pass factff to itself!

Claim: factff factff ≡ fact

- ((factff factff) 0) works trivially.
- ((factff factff) n) ≡ (n * ((factff factff) n-1))

Notice the two uses of self-application!

Recursion (cont.)

The fixpoint (Y) combinator performs the transformations of the previous slide.

fix ≡ $\lambda g.(\lambda f.g(f\ f))(\lambda f.g(f\ f))$

- $(\lambda f.g(f\ f))$ corresponds to the transformation of factf to factff.
- $(\lambda f.g(f\ f))(\lambda f.g(f\ f))$ corresponds to the call (factff factff).

This version only works under lazy evaluation; the call-by-value version is a little hairier.

Recursion Example

fix ≡ $\lambda g.(\lambda f.g(f\ f))(\lambda f.g(f\ f))$

factf ≡ λ f. λ n. if n=0 then 1 else n * f(n-1)

Claim: fix factf ≡ fact

Let h ≡ $(\lambda f.\text{factf}(f\ f))$

- fix factf 0 ⟶
- $(\lambda f.\text{factf}(f\ f))(\lambda f.\text{factf}(f\ f))$0 ⟶
- (factf (h h)) 0 ⟶
- (λ n. if n=0 then 1 else n * (h h)(n-1)) 0 ⟶
- if 0=0 then 1 else 0 * (h h)(0-1) ⟶
- if true then 1 else 0 * (h h)(0-1) ⟶
- 1

Recursion Example (cont.)

$\text{fix} \equiv \lambda g.(\lambda f.g(f\ f))(\lambda f.g(f\ f))$

factf $\equiv$ λ f. λ n. if n=0 then 1 else n * f(n-1)

Let h $\equiv (\lambda f.\text{factf}(f\ f))$

- $\underline{\text{fix factf}}\ 1 \longrightarrow$
- $\underline{(\lambda f.\text{factf}(f\ f))(\lambda f.\text{factf}(f\ f))1} \longrightarrow$
- $\underline{(\text{factf (h h)})}\ 1 \longrightarrow$
- $\underline{(\lambda \text{ n. if n=0 then 1 else n * (h h)(n-1)) 1}} \longrightarrow$
- if 1=0 then 1 else 1 * (h h)(1-1) $\xrightarrow{*}$
- 1 * $\underline{\text{(h h)(1-1)}} \xrightarrow{*}$
- 1 * 1 $\longrightarrow$
- 1

Values

$$\begin{array}{lll} e & ::= & x \\ & & \lambda x.e \\ & & e_1\ e_2 \end{array}$$

Some terms are in normal form, but don't make semantic sense.

- x
- $x\ (\lambda x.x)$

The subset of normal-form terms that "make semantic sense" are called values.

Values are the legal results of computations. This is a language-specific notion.

What should the values be for the λ-calculus?

Operational Semantics

The "meaning" of a term is the value (if any) that it reduces to (along with the sequence of steps to get there).

Define an abstract machine that "computes" the value of any term.

A state of the machine consists of the term being evaluated, as well as any other auxiliary information necessary.

The transition relation is defined by a set of inference rules:

$$\frac{<\text{premise}_1> \quad \cdots \quad <\text{premise}_n>}{<\text{conclusion}>}$$

"if $<\text{premise}_1>, \ldots, <\text{premise}_n>$ hold, then so does <conclusion>".

Call-by-Value Semantics

Syntax:

$$\begin{array}{lll} e & ::= & x \\ & & \lambda x.e \\ & & e_1\ e_2 \\ v & ::= & \lambda x.e \end{array}$$

Structural ("small-step") Operational Semantics:

$$\frac{}{(\lambda x.e)v \longrightarrow [x \mapsto v]e}\ \text{(E-AppRed)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{(E-App1)} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}\ \text{(E-App2)}$$

An Example Derivation

$$\frac{}{(\lambda x.e)v \longrightarrow [x \mapsto v]e}\text{(E-AppRed)} \quad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\text{(E-App1)} \quad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}\text{(E-App2)}$$

A derivation tree defines one step of the machine.

$$\cfrac{\cfrac{\cfrac{}{(\lambda x.x)(\lambda x.(x\ x)) \longrightarrow (\lambda x.(x\ x))}\text{(E-AppRed)}}{((\lambda x.x)(\lambda x.(x\ x)))x \longrightarrow (\lambda x.(x\ x))x}\text{(E-App1)}}{(\lambda x.x)(((\lambda x.x)(\lambda x.(x\ x)))x) \longrightarrow (\lambda x.x)((\lambda x.(x\ x))x)}\text{(E-App2)}$$

Derive reduction steps until reaching a normal form.

Call-by-need Semantics?

Syntax:

$$\begin{array}{rcl} e & ::= & x \\ & & \lambda x.e \\ & & e_1\ e_2 \\ \\ v & ::= & \lambda x.e \end{array}$$

Structural Operational Semantics:

Stuck Expressions

$$\frac{}{(\lambda x.e)v \longrightarrow [x \mapsto v]e}\text{(E-AppRed)} \quad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\text{(E-App1)} \quad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}\text{(E-App2)}$$

An expression e is stuck if e is not a value, but e cannot take a step (i.e. a derivation cannot be found).

$$\begin{array}{rcl} \text{stuck} & ::= & x \\ & & \text{stuck}\ e \\ & & v\ \text{stuck} \end{array}$$

Grammar is deduced by case analysis of the syntax

- x cannot take a step
- $\lambda x.e$ is a value
- $(e_1\ e_2)$: each e_i either can take a step, is stuck, or is a value

Eventually Stuck Expressions

$$\frac{}{(\lambda x.e)v \longrightarrow [x \mapsto v]e}\text{(E-AppRed)} \quad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\text{(E-App1)} \quad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}\text{(E-App2)}$$

An expression e is eventually stuck if $e \xrightarrow{*} e'$ and e' is stuck.

- $(\lambda x_1.x_2)(\lambda x.x)$

What is the grammar representing eventually stuck expressions?

What do stuck and eventually stuck expressions correspond to in “real” programming languages?

Booleans

Syntax:

$$
\begin{array}{lcl}
e & ::= & x \\
 & & \lambda x.e \\
 & & e_1\ e_2 \\
 & & \text{true} \\
 & & \text{false} \\
 & & \text{if } e_1 \text{ then } e_2 \text{ else } e_3
\end{array}
$$

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

Operational Semantics of Booleans

$$
\frac{}{(\lambda x.e)v \longrightarrow [x \mapsto v]e}\ \text{(E-AppRed)} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{(E-App1)} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}\ \text{(E-App2)}
$$

Booleans (cont.)

What is the grammar of stuck expressions?

$$
\begin{array}{lcl}
\text{stuck} & ::= & x \\
 & & \text{stuck } e \\
 & & v \text{ stuck}
\end{array}
$$