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

Todd Millstein
CSE505
October 26, 2001

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.

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

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 average POPL paper and understand the goals of the work, key concepts, notation.
Relying on your questions, comments, feedback.
How Language Theory?

The \( \lambda \)-calculus

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

Dynamic semantics for the \( \lambda \) calculus

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

Simply-typed \( \lambda \)-calculus

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

Type Soundness for the Simply typed \( \lambda \) calculus

Polymorphic Type Systems

**\( \lambda \)-calculus: Syntax**

\[
\begin{align*}
\mathit{e} & ::= \quad \mathit{x} \quad \text{variable} \\
& \quad \lambda x.e \quad \text{abstraction (function)} \\
& \quad e_1 e_2 \quad \text{application (function call)}
\end{align*}
\]

Conventions

- Metavariable \( x \) ranges over an infinite set of variable names,
- Metavariable \( e \) ranges over expressions (or terms) of the \( \lambda \)-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.

\( \alpha \) 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 \( \lambda \)-calculus

The only way to evaluate terms is via function application,

\[
(\lambda x.e_1) e_2 \rightarrow [x \mapsto e_2]e_1 \quad \text{(\( \beta \)-reduction)}
\]

- \( e \rightarrow e' \) 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 \( e_2 \)”

\[
\begin{align*}
[x \mapsto e]x & = e \\
[x \mapsto e]x' & = x' \quad \text{if } x \neq x' \\
[x \mapsto e](\lambda x'.e') & = \lambda x'.[x \mapsto e]e' \quad \text{if } x \neq x' \quad \text{and } x' \text{ not free in } e
\end{align*}
\]

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.

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

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

Reduction Strategies (cont.)

Let $\rightarrow^*$ be the reflexive, transitive closure of the $\rightarrow$ relation.

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

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

But not every term has a normal form,

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

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

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

Expression Power

Believe it or not, the $\lambda$-calculus is fully general: Church’s thesis is that every “effectively computable” function can be encoded as a $\lambda$-term.

Turing showed that every Turing machine can be encoded as a $\lambda$-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 $\lambda$-calculus extensions!

What is the $\lambda$ 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: \( \text{ifthenelse} \equiv \lambda b.\lambda t.\lambda e.b \, t \, e \)
- \( \text{ifthenelse} \, \text{tru} \, v \, w \rightarrow* \)
- \( \text{tru} \, v \, w \rightarrow \)
- \( (\lambda f.\text{v}) \, w \rightarrow \)
- \( v \)
and \( \equiv \lambda b_1.\lambda b_2.\text{ifthenelse} \, b_1 \, b_2 \, \text{fls} \equiv \lambda b_1.\lambda b_2.\, b_1 \, b_2 \, \text{fls} \)
- and \( \text{tru} \, \text{fls} \rightarrow* \)
- \( \text{tru} \, \text{fls} \, \text{fls} \rightarrow \)
- \( (\lambda f.\text{fls}) \, \text{fls} \)
- \( \text{fls} \)

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 \( \rightarrow \)
- \( \lambda s.\lambda z.s(\text{one} \, s \, z) \equiv \)
- \( \lambda s.\lambda z.s((\lambda s.\lambda z.s) \, s \, z) \)
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.
- \( \text{fact} \equiv \lambda \, n. \, \text{if } n=0 \, \text{then } 1 \, \text{else } n \times \text{fact}(n-1) \)

Replace recursive references with a call to an extra parameter,
- \( \text{factf} \equiv \lambda \, f, \, \lambda \, n. \, \text{if } n=0 \, \text{then } 1 \, \text{else } n \times f(n-1) \)

Iteratively define partial factorial functions.
- \( \text{fact}_0 \equiv \text{factf} \, \lambda x. \, x \)
- \( \text{fact}_1 \equiv \text{factf} \, \text{fact}_0 \)
- \( \text{fact}_2 \equiv \text{factf} \, \text{fact}_1 \)
- \( \ldots \)

The function \( \text{fact}^\infty \) is equivalent to \( \text{fact} \).

Recursion (cont.)

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

\[ \text{fix} \equiv \lambda g. \, (\lambda f. \, g(f \, f))(\lambda f. \, g(f \, f)) \]

- \( (\lambda f. \, g(f \, f)) \) corresponds to the transformation of \( \text{factf} \) to \( \text{factff} \).
- \( (\lambda f. \, g(f \, f))(\lambda f. \, g(f \, f)) \) corresponds to the call \( \text{factf} \, \text{factff} \).

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

Recursion (cont.)

\[ \text{fact} \equiv \lambda n. \, \text{if } n=0 \, \text{then } 1 \, \text{else } n \times \text{fact}(n-1) \]

\[ \text{factf} \equiv \lambda f, \, \lambda n. \, \text{if } n=0 \, \text{then } 1 \, \text{else } n \times f(n-1) \]

Let’s play a similar trick on \( f \) to the one we played on \( \text{fact} \).
- \( \text{factff} \equiv \lambda f, \, \lambda n. \, \text{if } n=0 \, \text{then } 1 \, \text{else } n \times (f \, f)(n-1) \)

Alternatively, let’s make the change “non invasively.”
- \( \text{factff} \equiv \lambda f. \, \text{factf} \, (f \, f) \)

Now pass \( \text{factff} \) to itself!

Claim: \( \text{factff} \, \text{factff} \equiv \text{fact} \)

- \( ((\text{factff} \, \text{factff}) \, 0) \) works trivially.
- \( ((\text{factff} \, \text{factff}) \, n) \equiv (n \times ((\text{factff} \, \text{factff}) \, n-1)) \)

Notice the two uses of self application!

Recursion Example

\[ \text{fix} \equiv \lambda g. \, (\lambda f. \, g(f \, f))(\lambda f. \, g(f \, f)) \]

\[ \text{factf} \equiv \lambda f, \, \lambda n. \, \text{if } n=0 \, \text{then } 1 \, \text{else } n \times f(n \, 1) \]

Claim: \( \text{fix} \, \text{factf} \equiv \text{fact} \)

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

- \( \text{fix} \, \text{factf} \, 0 \rightarrow \)
- \( (\lambda f. \, \text{factf}(f \, f))(\lambda f. \, \text{factf}(f \, f))0 \rightarrow \)
- \( (\text{factf} \, (h \, h)) \, 0 \rightarrow \)
- \( (\lambda n. \, \text{if } n=0 \, \text{then } 1 \, \text{else } n \times (h \, h)(n-1)) \, 0 \rightarrow \)
- \( \text{if } 0 \neq 0 \, \text{then } 1 \, \text{else } 0 \times (h \, h)(0 \, -1) \rightarrow \)
- \( \text{if } \text{true} \, \text{then } 1 \, \text{else } 0 \times (h \, h)(0 \, -1) \rightarrow \)
- \( 1 \)
Recursion Example (cont.)

\[ \text{fix } \equiv \lambda g. (\lambda f.g(f \ f))(\lambda f.g(f \ f)) \]
\[ \text{factf } \equiv \lambda \ n. \ \text{if } n=0 \ \text{then} \ 1 \ \text{else} \ n \ * \ \text{f}(n-1) \]

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

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

---

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:

\[
\frac{\text{<premise}_1 \ \cdots \ \text{<premise}_n}{\text{<conclusion>}}
\]

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

---

Values

\[
\begin{align*}
  e & := x \\
  & \quad \lambda x.e \\
  & \quad e_1 \ e_2
\end{align*}
\]

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

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

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 \( \lambda \)-calculus?

---

Call-by-Value Semantics

Syntax:

\[
\begin{align*}
  e & := x \\
  & \quad \lambda x.e \\
  v & := e_1 \ e_2 \\
\end{align*}
\]

Structural (“small-step”) Operational Semantics:

\[
\begin{align*}
(\lambda x.e)v & \rightarrow [x \mapsto v]e \quad \text{(E-AppRed)} \\
\frac{e_1 \rightarrow e'_1 \ e_2 \rightarrow e'_2}{e_1 \ e_2 \rightarrow e'_1 \ e'_2} \quad \text{(E-App1)}
\end{align*}
\]

\[
\frac{e \rightarrow e'}{v \ e \rightarrow v \ e'} \quad \text{(E-App2)}
\]
An Example Derivation

\[(\lambda x.e) v \longrightarrow [x \mapsto v] e \quad (E\text{-AppRed}) \quad \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad (E\text{-App1}) \quad \frac{e \longrightarrow e'}{v \ e \longrightarrow v \ e'} \quad (E\text{-App2})\]

A derivation tree defines one step of the machine.

\[
\begin{align*}
(\lambda x.e)(\lambda x.(x \ x)) & \longrightarrow (\lambda x.(x \ x)) \\
((\lambda x.e)(\lambda x.(x \ x)))x & \longrightarrow (\lambda x.(x \ x))x \\
(\lambda x.e)((\lambda x.x)(\lambda x.(x \ x))x) & \longrightarrow (\lambda x.x)((\lambda x.(x \ x))x)
\end{align*}
\quad (E\text{-App2})
\]

Derive reduction steps until reaching a normal form.

\[
\text{Stuck Expressions}
\]

\[(\lambda x.e) v \longrightarrow [x \mapsto v] e \quad (E\text{-AppRed}) \quad \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad (E\text{-App1}) \quad \frac{e \longrightarrow e'}{v \ e \longrightarrow v \ e'} \quad (E\text{-App2})\]

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

\[
\text{stuck} ::= x \quad \text{stuck} e \quad v \text{ stuck}
\]

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

\[
\text{Eventually Stuck Expressions}
\]

\[(\lambda x.e) v \longrightarrow [x \mapsto v] e \quad (E\text{-AppRed}) \quad \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad (E\text{-App1}) \quad \frac{e \longrightarrow e'}{v \ e \longrightarrow v \ e'} \quad (E\text{-App2})\]

An expression \( e \) is eventually stuck if \( e \overset{\ast}{\longrightarrow} e' \) 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:

\[
e ::= x \\
\text{\textlambda}x.e \\
e_1 e_2 \\
\text{true} \\
\text{false} \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]

\[
v ::= \text{\textlambda}x.e
\]

Operational Semantics of Booleans

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

Booleans (cont.)

What is the grammar of stuck expressions?

\[
\text{stuck} ::= x \\
\text{stuck } e \\
v \text{ stuck}
\]