Formal Semantics

Why formalize?
• some language features are tricky,
  e.g. generalizable type variables, nested functions
• some features have subtle interactions,
  e.g. polymorphism and mutable references
• some aspects often overlooked in informal descriptions,
  e.g. evaluation order, handling of errors

Want a clear and unambiguous specification that can be used by
language designers and language implementors
(and programmers when necessary)

Ideally, would allow rigorous proof of
• desired language properties, e.g. safety
• correctness of implementation techniques

Aspects to formalize

Syntax: what’s a syntactically well-formed program?
• formalize by a context-free grammar, e.g. in EBNF notation

Static semantics:
  which syntactically well-formed programs are also
  semantically well-formed?
  • i.e., name resolution, type checking, etc.
  • formalize using typing rules, well-formedness judgments

Dynamic semantics:
  to what does a semantically well-formed program evaluate?
  • i.e., run-time behavior of a type-correct program
  • formalize using operational, denotation, and/or axiomatic
    semantics rules

Metatheory:
  what are the properties of the formalization itself?
  • e.g., is static semantics sound w.r.t. dynamic semantics?

Approach

Formalizing & proving properties about a full language
  is very hard, very tedious
  • many, many cases to consider
  • lots of interacting features

Better approach:
  boil full-sized language down into its essential core, then
  formalize and study the core
  • cut out much of the complication as possible,
    without losing the key parts that need formal study
  • hope that insights gained about the core
    carry over to the full language

Can study language features in stages:
  • a very tiny core
  • then extend with an additional feature
  • then extend again (or separately)

Lambda calculus

The tiniest core of a functional programming language
  • Alonzo Church, 1930s

The foundation for all formal study of programming languages

Outline of study:
  • untyped λ-calculus:
    syntax, dynamic semantics, properties
  • simply typed λ-calculus:
    also static semantics, soundness
  • standard extensions to λ-calculus:
    syntax, dynamic semantics, static semantics
  • polymorphic λ-calculus:
    syntax, dynamic semantics, static semantics
Untyped \(\lambda\)-calculus: syntax

Syntax:

\[
E ::\! =\! \lambda I. E \quad \text{function / abstraction}
\mid E E \quad \text{call / application}
\mid I \quad \text{variable}
\]

[That's it!]

Application binds tighter than .
Can freely parenthesize as needed

Example (with minimum parens):

\((\lambda x. \lambda y. x \ y) \ \lambda z. z\)

ML analogue (if ignore types):

\(\texttt{fn x \Rightarrow (fn y \Rightarrow x \ y)} \ (\texttt{fn z \Rightarrow z})\)

Trees described by this grammar are called term trees

Free and bound variables

\(\lambda I. E\) binds \(I\) in \(E\)

An occurrence of a variable \(I\) is free in an expression \(E\)
if it’s not bound by some enclosing lambda in \(E\)

\(FV(E)\): set of free variables in \(E\)

\[
\begin{align*}
FV(I) & = \{I\} \\
FV(\lambda I. E) & = FV(E) - \{I\} \\
FV(E_1; E_2) & = FV(E_1) \cup FV(E_2)
\end{align*}
\]

\(FV(E) = \emptyset \iff E\) is closed

\(\alpha\)-renaming

First semantic property of \(\lambda\)-calculus:

a bound variable in a term tree (and all its references) can be renamed without affecting the semantics of the term tree

• cannot rename free variables

Precise definition:

\(\alpha\) equivalence: \(\lambda I_1. E \leftrightarrow \lambda I_2. ([I_2/I_1]E) \quad (\text{if } I_2 \not\in FV(E))\)

\([E_2/I]E_1\): substitute all free occurrences of \(I\) in \(E_1\) with \(E_2\)

• (formalized soon)

Since names of bound variables “don’t matter”, it’s convenient to treat all \(\alpha\) equivalent term trees as a single term

• define all later semantics for terms

• can assume that all bound variables are distinct

• for any particular term tree, do \(\alpha\)-renaming to make this so

Evaluation, \(\beta\)-reduction

Define how a \(\lambda\)-calculus program “runs” via a set of rewrite rules, a.k.a. reductions

• \(E_1 \rightarrow E_2\) means “\(E_1\) reduces to \(E_2\) in one step”

One rule: \(\lambda I. E_1) E_2 \rightarrow ([E_2/I]E_1)\)

• “applying a function to an argument expression reduces to the function’s body after substituting the argument expression for the function’s formal”

• this rule is called the \(\beta\)-reduction rule

Other rules state that the \(\beta\)-reduction rule can be applied to nested subexpressions, too

• (formalized later)

Define how a \(\lambda\)-calculus program “runs” to compute a final result as the reflexive, transitive closure of one-step reduction

• \(E \rightarrow^* \nu\) means “\(E\) reduces to result value \(\nu\)”

• (formalized later)

That’s it!
Examples

Substitution

Substitution is surprisingly tricky
• must avoid changing the meaning of any variable reference, in either substitutee or substituted expressions
• “capture-avoiding substitution”

Define formally by cases, over the syntax of the substitutee:
• identifiers:
\[ [E_2/I] I = E_2 \]
\[ [E_2/I] J = J \quad \text{(if } J \neq I) \]
• applications:
\[ [E_2/I] (E_1 E_3) = ([E_2/I] E_1) ([E_2/I] E_3) \]
• abstractions:
\[ [E_2/I] (\lambda I. E) = \lambda I. E \]
\[ [E_2/I] (\lambda J. E) = \lambda J. [E_2/I] E \quad \text{(if } J \neq I \text{ and } J \notin \text{FV}(E_2)) \]
• use \( \alpha \)-renaming on \((\lambda J. E)\) to ensure \( J \notin \text{FV}(E_2) \)

Defines the scoping rules of the \( \lambda \)-calculus

Normal forms

\( E \rightarrow^* V \): \( E \) evaluates fully to a value \( V \)
• \( \rightarrow^* \) defined as the reflexive, transitive closure of \( \rightarrow \)

What is \( V \)?
• an expression with no opportunities for \( \beta \)-reduction
• such expressions are called normal forms

Can define formally:
\[
V ::= \lambda I. V \\
\text{I V} \\
\text{I I}
\]
(I.e., any \( E \) except one containing \( \lambda I. E_1 \) \( E_2 \) somewhere)

Q: does every \( \lambda \)-calculus term have a normal form?
Q: is a term’s normal form unique?

Reduction order

Can have several places in an expression where a lambda is applied to an argument
• each is called a \textbf{redex}

\((\lambda x. (\lambda y. x) \ x) \ (\lambda z. z) \ (\lambda w. (\lambda v. v) \ w))\)

Therefore, have a choice in what reduction to make next

Which one is the right one to choose to reduce next?

Does it matter?
• to the final result?
• to how long it takes to compute it?
• to whether the result is computed at all?
Some possible reduction strategies

Example:
\((\lambda x. (\lambda y. x) x) ((\lambda z. z) (\lambda w. (\lambda v. v) w))\)

**normal-order** reduction:
- always choose leftmost, outermost redex
  - call-by-name, lazy evaluation:
    - same, and ignore redexes underneath \(\lambda\)

**applicative-order** reduction:
- always choose leftmost, outermost redex
  - whose argument is in normal form
  - call-by-value, eager evaluation:
    - same, and ignore redexes underneath \(\lambda\)

Again, does it matter?
- to the final result?
- to how long it takes to compute it?
- to whether the result is computed at all?

Amazing fact #1: Church-Rosser Thm., Part 1

Thm (Confluence). If \(e_1 \rightarrow^* e_2\) and \(e_1 \rightarrow^* e_3\), then \(\exists e_4 \text{ s.t. } e_2 \rightarrow^* e_4\) and \(e_3 \rightarrow^* e_4\).

![Diagram](image)

Corollary (Normalization). Every term has a unique normal form, if it exists
- No matter what reduction order is used!

Proof? [e.g. by contradiction]

Existence of normal form?

Does every term have a normal form?
- (If it does, we already know it's unique)

Consider:
\((\lambda x. x) (\lambda x. x)\)

Amazing fact #2: Church-Rosser Thm., Part 2

Thm. If a term has a normal form, then
normal-order reduction will find it!
- applicative-order reduction might not!

Example:
\((\lambda x. (\lambda y. y)) ((\lambda z. z) (\lambda z. z))\)

Same example, but using abbreviations:

- \(id = (\lambda y. y)\)
- \(loop = ((\lambda z. z) (\lambda z. z))\)
- \((\lambda x. id) \ loop\)

(Abbreviations are not really in the \(\lambda\)-calculus;
expand away textually before evaluating)

Q: How can I tell whether a term has a normal form?
Amazing fact #3: $\lambda$-calculus is Turing-complete!

Can translate any Turing machine program into an equivalent $\lambda$-calculus program, and vice versa

But how?

$\lambda$-calculus lacks:
- functions with multiple arguments
- numbers and arithmetic
- booleans and conditional branches
- data structures
- local variables
- recursive definitions and loops

All it’s got are one-argument, non-recursive functions...

Multiple arguments, via currying

Encode multiple arguments by currying

$\lambda(x, y).E \Rightarrow \lambda x. (\lambda y. E)$

$E(E_1, E_2) \Rightarrow (E_{E_1}) E_2$

Multiple arguments can be had via a syntactic sugar, so they’re not essential, and they can be dropped from the core language

Church numerals

Encode natural numbers using stylized $\lambda$ terms

$\text{zero} = (\lambda s. \lambda z. z) = (\lambda s. \lambda z. z^0 z)$

$\text{one} = (\lambda s. \lambda z. z)$

$\text{two} = (\lambda s. \lambda z. s (s z)) = (\lambda s. \lambda z. s^2 z)$

$\vdots$

$\bar{N} = (\lambda s. \lambda z. s^N z)$

($\bar{N}$ is the $\lambda$-calculus encoding of the mathematical number $N$)

A unary representation of numbers, but one that can be used to do computation
- a “number” $\bar{N}$ is a function that applies a “successor” function $(s)$ $N$ times to a “zero” value $(z)$

Arithmetic on Church numerals

A basic arithmetic function: $\text{succ}$

- $\text{succ} \bar{N} \rightarrow \bar{N+1}$

Definition:

$\text{succ} = (\lambda n. \lambda s. \lambda z. s (n s z))$

Examples:

$\text{succ zero} \rightarrow (\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s’. \lambda z’. z’)$

$\rightarrow (\lambda s. \lambda z. s (\lambda s’. \lambda z’. z’) s z))$

$\rightarrow (\lambda s. \lambda z. s (\lambda z’. z’) z))$

$\rightarrow (\lambda s. \lambda z. s z) = \text{one}$

$\text{succ two} \rightarrow (\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s’. \lambda z’. s’ (s’ z’))$

$\rightarrow (\lambda s. \lambda z. s (\lambda s’. \lambda z’. s’ (s’ z’)) s z))$

$\rightarrow (\lambda s. \lambda z. s (\lambda z’. s (s z’)) z))$

$\rightarrow (\lambda s. \lambda z. s (s (s z))) = \text{three}$
Addition

Another basic arithmetic function: `add`

- `add \vec{x} \vec{y} \rightarrow \vec{x} + \vec{y}`

Algorithm: to add \(x\) to \(y\), apply `succ` to \(y\) \(X\) times

Key trick: \(X\) is a function that applies its first argument to its second argument \(X\) times

- “a number is as a number does”

Definition:

\[
add = (\lambda x.\lambda y. x \ succ y)
\]

Example:

\[
add \ two \ three = (\lambda x.\lambda y. x \ succ y) \ two \ three
\]
\[
\rightarrow^* \ two \ succ \ three = (\lambda s.\lambda z. (s \ z)) \ succ \ three
\]
\[
\rightarrow^* \ succ \ (succ \ three)
\]
\[
\rightarrow^* \ five
\]

(pred is tricky, but doable; sub then is similar to add)

Multiplication

Another basic arithmetic function: `mul`

- `mul \vec{x} \vec{y} \rightarrow \vec{x} \times \vec{y}`

Booleans and conditionals

How to make choices? We only have functions...

Key idea:

- true and false are encoded as functions that work differently
- call the boolean value to control evaluation

\[
true = (\lambda t.\lambda e. t)
\]
\[
false = (\lambda t.\lambda e. e)
\]

\[
if = (\lambda b.\lambda t.\lambda e. b \ t \ e)
\]

Example:

\[
if \ false \ loop \ three
\]
\[
= (\lambda b.\lambda t.\lambda e. b \ t \ e) \ false \ loop \ three
\]
\[
\rightarrow^* \ false \ loop \ three = (\lambda t.\lambda e.e) \ loop \ three
\]
\[
\rightarrow^* \ three
\]

Testing numbers

To complete Peano arithmetic, need an `isZero` predicate

- `isZero \vec{N} \rightarrow^* \vec{N}=0`

Idea: implement by calling the number on a successor function that always returns false and a zero value that is true

Definition:

\[
isZero = (\lambda n. (\lambda x.\lambda.e) true)
\]

Examples:

\[
isZero \ zero = (\lambda n. (\lambda x.\lambda.e) true) \ zero
\]
\[
\rightarrow (\lambda s'.\lambda z'.z') \ (\lambda x.\false) \ true
\]
\[
\rightarrow^* \ true
\]

\[
isZero \ two = (\lambda n. (\lambda x.\lambda.e) true) \ two
\]
\[
\rightarrow (\lambda s'.\lambda z'.s' \ (s' \ z')) \ (\lambda x.\false) \ true
\]
\[
\rightarrow^* (\lambda x.\false) ((\lambda x.\false) true)
\]
\[
\rightarrow \ false
\]
Data structures

E.g., pairs

Idea: a pair is a function that remembers its two parts
(via lexical scoping & closures)
  • pair function takes a selector function that’s
    passed both parts and then chooses one

pair = (λf.λs.λb.f s)

fst = (λp.p (λf.λs.f))
snd = (λp.p (λf.λs.s))

Examples:

pair true four = (λf.λs.λb.f s) true four
  → (λb.b true four)

snd (pair true four) = (λp.p (λf.λs.s)) (p t f)
  → (pair true four) (λf.λs.s)
  → (λb.b true four) (λf.λs.s)
  → (λf.λs.s) true four
  → four

Local variables

Encode let using functions

let I = E₁ in E₂  ⇒  (λI.E₂) E₁

Example:

let x = one in
  let y = two in
  add x y
  ⇒  (λx.(λy.add x y) two) one

Doesn’t handle recursive declarations, though:

let fact = ... fact ... in
  fact two
  ⇒  (λfact.fact two) (... fact ...)

Loops and recursion

We’ve seen that we can write infinite loops in the λ-calculus

loop = ((λz.z z) (λz.z z))

Can we write useful loops?
I.e., can we write recursive functions?

The let encoding won’t work, as we saw

How about this?

fact = (λn.
  if (isZero n) one
  (mul n (fact (pred n)))))

Amazing fact #4:
Can define recursive functions non-recursively!

Step 1: replace the bogus recursive reference with an explicit argument

factG = (λfact.λn.
  if (isZero n) one
  (mul n (fact (pred n)))))

Step 2: use the “paradoxical Y combinator” to pass factG to itself in a funky way to yield plain fact

fact = (Y factG)

Now all we have to do is write Y in the raw λ-calculus
The Y combinator

A definition of Y:
\[ Y = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \]

Example:
\[ Y f G = (\lambda f. (\lambda x. f (x x)) (\lambda x'. f (x' x'))) f G \]
\[ \longrightarrow (\lambda x. f G (x x)) (\lambda x'. f G (x' x')) \]
\[ \longrightarrow f G ((\lambda x'. f G (x' x')) (\lambda x'. f G (x' x')))) \]
\[ \equiv f G (Y f G) \]

So: \((Y f G)\) reduces to a call to \(f G\), whose argument is an expression that, if evaluated inside \(f G\), will reinvoke \(f G\) again with the same argument

- normal-order evaluation will only reduce “recursive” argument \((Y f G)\) on demand, as needed

Example

A concrete example:
\[ \text{factG} = (\lambda \text{fact}. \lambda n. \]
\[ \quad \text{if} \ (\text{isZero} \ n) \text{ one} \]
\[ \quad \text{mul} \ n \ (\text{fact} \ (\text{pred} \ n))) \]
\[ \text{fact} = (Y \text{factG}) \]
\[ (* \ Y f G \rightarrow^* f G (Y f G) *) \]
\[ \text{fact two} = Y \text{factG two} \]
\[ \rightarrow^* \text{factG} (Y \text{factG} \text{ two}) \]
\[ \rightarrow^* \text{if} \ (\text{isZero two}) \text{ one} \]
\[ \quad \text{mul} \ two \ ((Y \text{factG}) \ (\text{pred two})) \]
\[ [\text{doing some applicative-order reduction, for simplicity}] \]
\[ \rightarrow^* \text{mul} \ two \ ((Y \text{factG}) \text{ one}) \]
\[ \rightarrow^* \text{mul} \ two \]
\[ \quad \text{if} \ (\text{isZero one}) \text{ one} \]
\[ \quad \text{mul} \ one \ ((Y \text{factG}) \ (\text{pred one})) \]
\[ \rightarrow^* \text{mul} \ two \]
\[ \quad \text{mul} \ one \ ((Y \text{factG}) \ (\text{pred one})) \]
\[ \rightarrow^* \text{mul} \ two \ (\text{if} \ (\text{isZero zero}) \text{ one} \ (\text{mul} \ zero \ ...)) \]
\[ \rightarrow^* \text{mul} \ two \ (\text{mul} \ one \ one) \rightarrow^* \text{two} \]

Letrec

Can now define a recursive version of let:
\[ \text{letrec I} = E_1 \text{ in } E_2 \Rightarrow \text{let } I = (\lambda I. E_1) \text{ in } E_2 \]
- can now reference \(I\) recursively inside \(E_2\)

Example:
\[ \text{letrec} \]
\[ \quad \text{fact} = (\lambda n. \text{if} \ (\text{isZero} \ n) \text{ one} \]
\[ \quad \quad \text{mul} \ n \ (\text{fact} \ (\text{pred} \ n))) \]
\[ \text{in } \]
\[ \quad \ldots \text{fact} \ldots \]

Summary, so far

Saw untyped \(\lambda\)-calculus

Saw \(\alpha\)-renaming, \(\beta\)-reduction rules
  - both relied on capture-avoiding substitution
  - \(\alpha\)-renaming defined families of equivalent term trees
    - name choice of formals doesn’t matter to semantics
  - \(\beta\)-reduction defined “evaluation” of a \(\lambda\)-calculus “program”
    - normal forms: no more \(\beta\)-reduction possible
    - the “results” of a “program”
    - reduction strategies such as normal-order & applicative-order
      had different termination properties, but not different results

Church-Rosser: key confluence & normalization thms.

Turing-completeness of untyped \(\lambda\)-calculus suggested by successfully encoding many standard PL features