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 λ-calculus: syntax

Syntax:

```
E     ::= λI. E          function / abstraction
        | E E            call / application
        | I              variable
```

[That's it!]

Application binds tighter than `.`

Can freely parenthesize as needed

Example (with minimum parens):

```
(λx. λy. x y) λz.z
```

ML analogue (if ignore types):

```
(fn x => (fn y => x y)) (fn z => 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

$FV(I) = \{I\}$

$FV(\lambda I.E) = FV(E) - \{I\}$

$FV(E_1\ E_2) = FV(E_1) \cup FV(E_2)$

$FV(E) = \emptyset \Leftrightarrow E$ is **closed**

α-renaming

First semantic property of λ-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:

α-equivalence: $\lambda I_1.E \Leftrightarrow \lambda I_2.[I_2/I_1]E$ (if $I_2 \notin FV(E)$)

$[E_2/I]E_1$: **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 α-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 α-renaming to make this so

Evaluation, β-reduction

Define how a λ-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 **β-reduction** rule

Other rules state that the **β-reduction** rule can be applied to nested subexpressions, too

- (formalized later)

Define how a λ-calculus program "runs" to compute a final result as the reflexive, transitive closure of one-step reduction

- "$E \rightarrow^* V$" means "E reduces to result value V"
- (formalized later)

That's it!

Examples

Substitution

Substitution is suprisingly 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:

  $[E_2/I]I = E_2$

  $[E_2/I]J = J$ (if $J \neq I$)
- applications:

  $[E_2/I](E_1\ E_3) = ([E_2/I]E_1)\ ([E_2/I]E_3)$
- abstractions:

  $[E_2/I](\lambda I.E) = \lambda I.E$

  $[E_2/I](\lambda J.E) = \lambda J.[E_2/I]E$

  (if $J \neq I$ and $J \notin FV(E_2)$)
  - use α-renaming on $(\lambda J.E)$ to ensure $J \notin FV(E_2)$

Defines the scoping rules of the λ-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 β-reduction

- such expressions are called **normal forms**

Can define formally:

V ::= $\lambda I.V$
| $I\ V$
| I

(I.e., any E except one containing $(\lambda I.E_1)E_2$ somewhere)

Q: does every λ-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 **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:

```
(λx.(λy.x) x) ((λz.z) (λw.(λv.v) w))
```

normal-order reduction:
always choose leftmost, outermost redex

- call-by-name, lazy evaluation:
  same, and ignore redexes underneath λ

applicative-order reduction:
always choose leftmost, outermost redex
whose argument is in normal form

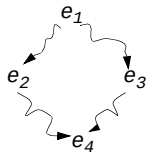
- call-by-value, eager evaluation:
  same, and ignore redexes underneath λ

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 $\exists\ e_4$ s.t. $e_2 \rightarrow^* e_4$ and $e_3 \rightarrow^* e_4$.



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:

```
(λx.x x) (λx.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:

```
(λx.(λy.y)) ((λz.z z) (λz.z z))
```

Same example, but using abbreviations:

```
id ≡ (λy.y)
loop ≡ ((λz.z z) (λz.z z))
(λx.id) loop
```

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

Q: How can I tell whether a term has a normal form?

Amazing fact #3: λ-calculus is Turing-complete!

Can translate any Turing machine program into
an equivalent λ-calculus program, and vice versa

But how?

λ-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 λ terms

$zero \equiv (\lambda s.\lambda z.z) \equiv (\lambda s.\lambda z.s^0\ z)$
$one \equiv (\lambda s.\lambda z.s\ z) \equiv (\lambda s.\lambda z.s^1\ z)$
$two \equiv (\lambda s.\lambda z.s\ (s\ z)) \equiv (\lambda s.\lambda z.s^2\ z)$
...
$\overline{N} \equiv (\lambda s.\lambda z.s^N\ z)$

($\overline{N}$ is the λ-calculus encoding of the mathematical number N)

A unary representation of numbers,
but one that can be used to do computation

- a "number" $\overline{N}$ is a function that applies
a "successor" function (s) N times to a "zero" value (z)

Arithmetic on Church numerals

A basic arithmetic function: succ

- $succ\ \overline{N} \rightarrow^* \overline{N+1}$

Definition:

$succ \equiv (\lambda n.\ \lambda s.\lambda z.s\ (n\ s\ z))$

Examples:

$succ\ zero$
$= (\lambda n.\lambda s.\lambda z.s\ (n\ s\ z))\ (\lambda s'.\lambda z'.z')$
$\rightarrow (\lambda s.\lambda z.s\ ((\lambda s'.\lambda z'.z')\ s\ z))$
$\rightarrow (\lambda s.\lambda z.s\ ((\lambda z'.z')\ z))$
$\rightarrow (\lambda s.\lambda z.s\ z) = one$

$succ\ two$
$= (\lambda n.\lambda s.\lambda z.s\ (n\ s\ z))\ (\lambda s'.\lambda z'.s'\ (s'\ z'))$
$\rightarrow (\lambda s.\lambda z.s\ ((\lambda s'.\lambda z'.s'\ (s'\ z'))\ s\ z))$
$\rightarrow (\lambda s.\lambda z.s\ ((\lambda z'.s\ (s\ z'))\ z))$
$\rightarrow (\lambda s.\lambda z.s\ (s\ (s\ z))) = three$

Addition

Another basic arithmetic function: `add`

- $add\ \overline{X}\ \overline{Y} \rightarrow^* \overline{X+Y}$

Algorithm: to add $\overline{X}$ to $\overline{Y}$, apply *succ* to $\overline{Y}$ X times

Key trick: $\overline{X}$ is a function that applies its first argument to its second argument X times

- "a number is as a number does"

Definition:

$add \equiv (\lambda x.\lambda y.x\ succ\ y)$

Example:

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

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

Multiplication

Another basic arithmetic function: `mul`

- $mul\ \overline{X}\ \overline{Y} \rightarrow^* \overline{X*Y}$

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

$true \equiv (\lambda t.\lambda e.t)$
$false \equiv (\lambda t.\lambda e.e)$

$if \equiv (\lambda b.\lambda t.\lambda e.b\ t\ e)$

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

- $isZero\ \overline{N} \rightarrow^* \overline{N=0}$

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

Definition:

$isZero \equiv (\lambda n.n\ (\lambda x.false)\ true)$

Examples:

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

$isZero\ two = (\lambda n.n\ (\lambda x.false)\ true)\ two$
$\rightarrow (\lambda s'.\lambda z'.s'\ (s'\ z'))\ (\lambda x.false)\ true$
$\rightarrow^* (\lambda x.false)\ ((\lambda x.false)\ true)$
$\rightarrow false$

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.b f s)

fst  ≡ (λp.p (λf.λs.f))
snd  ≡ (λp.p (λf.λs.s))
```

Examples:

```
pair true four = (λf.λs.λb.b 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 ≡ (λf.(λx.f (x x)) (λx.f (x x)))
```

Example:

```
Y fG = (λf.(λx.f (x x)) (λx′.f (x′ x′))) fG
  → (λx.fG (x x)) (λx′.fG (x′ x′))
  → fG ((λx′.fG (x′ x′)) (λx′.fG (x′ x′))))
  = fG (Y fG)
```

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

- normal-order evaluation will only reduce "recursive" argument (*Y fG*) on demand, as needed

Example

A concrete example:

```
factG ≡ (λfact.λn.
  if (isZero n) one
     (mul n (fact (pred n))))
fact ≡ (Y factG)
(* Y fG →* fG (Y fG) *)

fact two = Y factG two
  →* factG (Y factG) two
  →* if (isZero two) one
         (mul two ((Y factG) (pred two)))
  →* mul two ((Y factG) (pred two))
```

[doing some applicative-order reduction, for simplicity]

```
  →* mul two (factG (Y factG) one)
  →* mul two
       (if (isZero one) one
           (mul one ((Y factG) (pred one))))
  →* mul two
       (mul one ((Y factG) (pred one)))
  →* mul two (mul one
       (if (isZero zero) one (mul zero ...)))
  →* mul two (mul one one) →* two
```


Letrec

Can now define a recursive version of `let`:

letrec *I* = E_1 **in** E_2 ⇒ **let** *I* = *Y* (λ*I*.E_1) **in** E_2

- can now reference *I* recursively inside E_1

Example:

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


Summary, so far

Saw untyped λ-calculus

Saw α-renaming, β-reduction rules

- both relied on capture-avoiding substitution
- α-renaming defined families of equivalent term trees
  - name choice of formals doesn't matter to semantics
- β-reduction defined "evaluation" of a λ-calculus "program"
  - normal forms: no more β-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 λ-calculus suggested by successfully encoding many standard PL features