Formal Semantics

1

Why formalize?

- ML is tricky, particularly in corner cases
  - generalizable type variables?
  - polymorphic references?
  - exceptions?
- Some things are often overlooked for any language
  - evaluation order? side-effects? errors?
- Therefore, want to formalize what a language's definition really is
  - Ideally, a clear & unambiguous way to define a language
  - Programmers & compiler writers can agree on what's supposed to happen, for *all* programs
  - Can try to prove rigorously that the language designer got all the corner cases right

2

Aspects to formalize

- **Syntax**: what's a syntactically well-formed program?
  - EBNF notation for a context-free grammar
- **Static semantics**: which *syntactically* well-formed programs are *semantically* well-formed? which programs type-check?
  - typing rules, well-formedness judgments
- **Dynamic semantics**: what does a program *evaluate to* or *do* when it runs?
  - operational, denotational, or axiomatic semantics
- **Metatheory**: properties of the formalization itself
  - E.g. do the static and dynamic semantics match? i.e., is the static semantics **sound** w.r.t. the dynamic semantics?

3

Approach

- Formalizing full-sized languages is very hard, tedious
  - many cases to consider
  - lots of interacting features
- Better: boil full-sized language down into *essential core*, then formalize and study the core
  - cut out as much complication as possible, without losing the key parts that need formal study
  - hope that insights gained about core will carry back to full-sized language

4

The lambda calculus

- The essential core of a (functional) programming language
  - The tiniest Turing-complete programming language
- Outline:
  - Untyped: syntax, dynamic semantics, cool properties
  - Simply typed: static semantics, soundness, more cool properties
  - Polymorphic: fancier static semantics

5

Untyped λ-calculus: syntax

- (Abstract) syntax:

$e ::= x$	variable
$\mid \lambda x.\ e$	function/abstraction
	($\cong$ `fn x => e`)
$\mid e_1\ e_2$	call/application

  - Freely parenthesize in concrete syntax to imply the right abstract syntax
- The trees described by this grammar are called **term trees**

6

Free and bound variables

- $\lambda x.\ e$ **binds** x in e
- An occurrence of a variable x is **free** in e if it's not bound by some enclosing lambda

$$\text{freeVars}(x) \equiv x$$
$$\text{freeVars}(\lambda x.\ e) \equiv \text{freeVars}(e) - \{x\}$$
$$\text{freeVars}(e_1\ e_2) \equiv \text{freeVars}(e_1) \cup \text{freeVars}(e_2)$$

- e is **closed** iff freeVars(e) = {}

7

α-renaming

- First semantic property of lambda calculus: bound variables in a term tree can be renamed (properly) without affecting the semantics of the term tree
  - α-equivalent term trees
    - $(\lambda x_1.\ x_2\ x_1) \Leftrightarrow_\alpha (\lambda x_3.\ x_2\ x_3)$
  - cannot rename free variables
- **term** e: e and all α-equivalent term trees
  - Can freely rename bound vars whenever helpful

8

Evaluation: β-reduction

- Define what it means to "run" a lambda-calculus program by giving simple reduction/rewriting/simplification rules
  - "$e_1 \rightarrow_\beta e_2$" means "$e_1$ evaluates to e_2 in one step"
- One case:
  - $(\lambda x.\ e_1)\ e_2 \rightarrow_\beta [x \rightarrow e_2]e_1$
  - "if you see a lambda applied to an argument expression, rewrite it into the lambda body where all free occurrences of the formal in the body have been replaced by the argument expression"

9

Examples

10

Substitution

- When doing substitution, must avoid changing the meaning of a variable occurrence

$$[x \rightarrow e]x \equiv e$$
$$[x \rightarrow e]y \equiv y, \text{ if } x \neq y$$
$$[x \rightarrow e](\lambda x.\ e_2) \equiv (\lambda x.\ e_2)$$
$$[x \rightarrow e](\lambda y.\ e_2) \equiv (\lambda y.\ [x \rightarrow e]e_2), \text{ if } x \neq y \text{ and } y \text{ not free in } e$$
$$[x \rightarrow e](e_1\ e_2) \equiv ([x \rightarrow e]e_1)\ ([x \rightarrow e]e_2)$$

  - use α-renaming to ensure "y not free in e"

11

Result of reduction

- To fully evaluate a lambda calculus term, i.e., to determine the **meaning** of a term, simply perform β-reduction until you can't any more
  - $\rightarrow_\beta{}^* \equiv$ reflexive, transitive closure of $\rightarrow_\beta$
- When you can't any more, you have a **value**, which is a **normal form** of the input term
  - Does every lambda-calculus term have a normal form?

12

Reduction order

- Can have several lambdas applied to an argument in one expression
  - Each called a **redex**
- Therefore, several possible choices in reduction
  - Which to choose?
  - Does it matter?
    - To the final result?
    - To how long it takes to compute?
    - To whether the result is computed at all?

13

Two reduction orders

- Normal-order reduction
  (a.k.a. call-by-name, lazy evaluation)
  - reduce leftmost, outermost redex
- Applicative-order reduction
  (a.k.a. call-by-value, eager evaluation)
  - reduce leftmost, outermost redex
    whose argument is in normal form

14

Amazing fact #1: Church-Rosser Theorem, Part 1

- Thm. If $e_1 \rightarrow_\beta^* e_2$ and $e_1 \rightarrow_\beta^* e_3$, then $\exists$ e_4 such that $e_2 \rightarrow_\beta^* e_4$ and $e_3 \rightarrow_\beta^* e_4$
- Corollary. Every term has a unique normal form, if it has one
  - No matter what reduction order is used!
  - Wow!

15

Existence of normal forms?

- Does every term have a normal form?
- Consider: $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

16

Amazing fact #2: Church-Rosser Theorem, Part 2

- If a term has a normal form, then normal-order reduction will find it!
  - Applicative-order reduction might not!
- Example:
  - $(\lambda x_1.\ (\lambda x_2.\ x_2))\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$

17