Formal Semantics

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

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?

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

The lambda calculus

- The essential core of a (functional) programming language
  - Developed by Alonzo Church in the 1930's
    - Before computers were invented!
- Outline:
  - Untyped: syntax, dynamic semantics, cool properties
  - Simply typed: static semantics, soundness, more cool properties
  - Polymorphic: fancier static semantics

Untyped l-calculus: syntax

- (Abstract) syntax:

e ::= x	variable
| lx. e	function/abstraction
	(`@ fn x => e`)
| 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**

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

  freeVars(x) $\equiv$ x
  freeVars($\lambda x.\ e$) $\equiv$ freeVars(e) – {x}
  freeVars($e_1\ e_2$) $\equiv$ freeVars(e_1) $\cup$ freeVars(e_2)
- e is **closed** iff freeVars(e) = {}

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

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"
- Can do this rewrite anywhere inside an expression

Examples

Substitution

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

  $[x \rightarrow e]x \equiv e$
  $[x \rightarrow e]y \equiv y$ if $x \neq y$
  $[x \rightarrow e](\lambda x.\ e_2) \equiv (\lambda x.\ e_2)$
  $[x \rightarrow e](\lambda y.\ e_2) \equiv (\lambda y.\ [x \rightarrow e]e_2)$ if $x \neq y$ and y not free in e
  $[x \rightarrow e](e_1\ e_2) \equiv ([x \rightarrow e]e_1)\ ([x \rightarrow e]e_2)$
  - can use α-renaming to ensure "y not free in e"

Result of reduction

- To fully evaluate a lambda calculus 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?

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? Must we do them all?
  - Does it matter?
    - To the final result?
    - To how long it takes to compute?
    - To whether the result is computed at all?

13

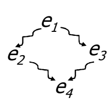
Two reduction orders

- 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 (i.e., is a value)*

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!

15

Existence of normal forms?

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

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

Weak head normal form

- What should this evaluate to?
  $(\lambda y.\ (\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$
  - Normal-order and applicative-order evaluation run forever
  - But in regular languages, wouldn't evaluate the function's body until we called it
- "Head" normal form doesn't evaluate arguments until function expression is a lambda
- "Weak" evaluation doesn't evaluate under lambda
- With these alternative definitions of reduction:
  - Reduction terminates on more lambda terms
  - Correspond more closely to real languages (particularly "weak")

18

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

- But the λ-calculus is too weak, right?
  - No multiple arguments!
  - No numbers or arithmetic!
  - No booleans or if!
  - No data structures!
  - No loops or recursion!

19

Multiple arguments: currying

- **Encode** multiple arguments via curried functions, just as in regular ML

$\lambda(x_1, x_2).\ e \quad \Rightarrow \quad \lambda x_1.\ (\lambda x_2.\ e) \quad (\equiv \lambda x_1\ x_2.\ e)$

$f(e_1, e_2) \quad \Rightarrow \quad (f\ e_1)\ e_2$

20

Church numerals

- **Encode** natural numbers using stylized lambda terms

  $zero \equiv \lambda s.\ \lambda z.\ z$

  $one \equiv \lambda s.\ \lambda z.\ s\ z$

  $two \equiv \lambda s.\ \lambda z.\ s\ (s\ z)$

  ...

  $n \equiv \lambda s.\ \lambda z.\ s^n\ z$
- A unary encoding using functions
  - No stranger than binary encoding

21

Arithmetic on Church numerals

- Successor function:
  take (the encoding of) a number,
  return (the encoding of) its successor
  - I.e., add an *s* to the argument's encoding

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

    $succ\ zero \rightarrow_\beta$
    $\lambda s.\ \lambda z.\ s\ (zero\ s\ z) \rightarrow_\beta^*$
    $\lambda s.\ \lambda z.\ s\ z\ =\ one$

    $succ\ two \rightarrow_\beta$
    $\lambda s.\ \lambda z.\ s\ (two\ s\ z) \rightarrow_\beta^*$
    $\lambda s.\ \lambda z.\ s\ (s\ (s\ z))\ =\ three$

22

Addition

- To add *x* and *y*, apply *succ* to *y* *x* times
  - Key idea: *x* is a function that, given a function and a base, applies the function to the base *x* times
    - "a number is as a number does"

    $plus \equiv \lambda x.\ \lambda y.\ x\ succ\ y$

    $plus\ two\ three \rightarrow_\beta^*$
    $two\ succ\ three \rightarrow_\beta^*$
    $succ\ (succ\ three)\ =\ five$
- Multiplication is repeated addition, similarly

23

Booleans

- Key idea: true and false are **encoded** as functions that do different things to their arguments, i.e., make a choice

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

  $true \equiv \lambda t.\ \lambda e.\ t$

  $false \equiv \lambda t.\ \lambda e.\ e$

  $if\ false\ four\ six \rightarrow_\beta^*$
  $false\ four\ six \rightarrow_\beta^*$
  six

24

Combining numerals & booleans

- To complete Peano arithmetic, need an isZero predicate
  - Key idea: call the argument number on a successor function that always returns false (not zero) and a base value that's true (is zero)

    isZero " l*n*. *n* (l*x*. *false*) *true*

    isZero zero fi $_b^*$
      zero (l*x*. *false*) *true* fi $_b^*$
      true
    isZero two fi $_b^*$
      two (l*x*. *false*) *true* fi $_b^*$
      (l*x*. *false*) ((l*x*. *false*) *true*) fi $_b^*$
      false

Data structures

- Try to encode simple pairs
  - Can build more complex data structures out of them
- Key idea: a pair is a function that remembers its two input values, and passes them to a client function on demand
  - First and second are client functions that just return one or the other remembered value

    mkPair " l*f*. l*s*. l*x*. *x f s*
    first " l*p*. *p* (l*f*. l*s*. *f*)
    second " l*p*. *p* (l*f*. l*s*. *s*)

    second (*mkPair true four*) fi $_b^*$
      second (l*x*. *x true four*) fi $_b^*$
      (l*x*. *x true four*) (l*f*. l*s*. *s*) fi $_b^*$
      (l*f*. l*s*. *s*) *true four* fi $_b^*$
      four

Loops and recursion

- l-calculus can write infinite loops
  - E.g. (l*x*. *x x*) (l*x*. *x x*)
- What about useful loops?
  - I.e., recursive functions?
- Ill-defined attempt:

    fact " l*n*.
      if (*isZero n*) *one* (*times n* (***fact*** (*minus n one*)))

  - Recursive reference isn't defined in our simple short-hand notation
  - We're trying to define what recursion means!

Amazing fact #*N*: Can define recursive funs non-recursively!

- Step 1: replace the bogus self-reference with an explicit argument

    factG " l***f***. l*n*.
      if (*isZero n*) *one* (*times n* (***f*** (*minus n one*)))

- Step 2: use the **paradoxical Y combinator** to "tie the knot"

    fact " *Y factG*

- Now all we need is a magic *Y* that makes its non-recursive argument act like a recursive function…

Y combinator

- A definition of *Y*:

    Y " l*f*. (l*x*. *f* (*x x*)) (l*x*. *f* (*x x*))

- When applied to a function *f*:

    Y f fi $_b$
      (l*x*. *f* (*x x*)) (l*x*. *f* (*x x*)) fi $_b$
      f ((l*x*. *f* (*x x*)) (l*x*. *f* (*x x*))) = *f* (*Y f*) fi $_b^*$
      f (*f* (*Y f*)) fi $_b^*$ *f* (*f* (*f* (*Y f*))) fi $_b^*$ …

  - Applies its argument to itself as many times as desired
  - "Computes" the **fixed point** of *f*
    - Often called **fix**

Y for factorial

fact two fi $_b^*$
  (*Y factG*) *two* fi $_b^*$
  factG (*Y factG*) *two* fi $_b^*$
  if (*isZero two*) *one*
    (*times two* ((*Y factG*) (*minus two one*))) fi $_b^*$
  times two ((*Y factG*) *one*) fi $_b^*$
  times two (*factG* (*Y factG*) *one*) fi $_b^*$
  times two (*if* (*isZero one*) *one*
    (*times one* ((*Y factG*) (*minus one one*)))) fi $_b^*$
  times two (*times one* ((*Y factG*) *zero*)) fi $_b^*$
  times two (*times one* (*factG* (*Y factG*) *zero*)) fi $_b^*$
  times two (*times one* (*if* (*isZero zero*) *one*
    (*times zero* ((*Y factG*) (*minus zero one*))))) fi $_b^*$
  times two (*times one one*) fi $_b^*$ *two*

Some intuition (?)

- Y passes a recursive call of a function to the function
- Will lead to infinite reduction, unless one recursive call chooses to ignore its recursive function argument
  - I.e., have a base case that's not defined recursively
  - Relies on normal-order evaluation to avoid evaluating the recursive call argument until needed

31

Summary, so far

- Saw untyped l-calculus syntax
- Saw some rewriting rules, which defined the semantics of l-terms
  - a-renaming for changing bound variable names
  - b-reduction for evaluating terms
    - Normal form when no more evaluation possible
    - Normal-order vs. applicative-order strategies
- Saw some amazing theorems
- Saw the power of l-calculus to encode lots of higher-level constructs

32

Simply-typed lambda calculus

- Now, let's add static type checking
- Extend syntax with types:

  t ::= t_1 fi t_2 |

  e ::= lx:t. e | x | e_1 e_2

  - (The dot   is just the base case for types, to stop the recursion. Values of this type will never be invoked, just passed around.)

33

Typing judgments

- Introduce a compact notation for defining typechecking rules
- A **typing judgment**: G ├ e : t
  - "In the typing context G, expression e has type t"
- A **typing context**: a mapping from variables to their types
  - Syntax: G ::= {} | G, x : t

34

Typing rules

- Give typechecking rule(s) for each kind of expression
- Write as a logical **inference rule**

  $$\frac{\textit{premise}_1 \ldots \textit{premise}_n \quad (n ‡ 0)}{\textit{conclusion}}$$

  - Whenever all the premises are true, can deduce that the conclusion is true
  - If no premises, then called an "axiom"
- Each premise and conclusion has the form of a typing judgment

35

Typing rules for simply-typed l-calculus

$$\frac{\text{G}, x{:}t_1 \vdash e : t_2}{\text{G} \vdash (\text{l}x{:}t_1.\ e) : t_1 \text{ fi } t_2} \text{ [T-ABS]}$$

$$\frac{}{\text{G} \vdash x : \text{G}(x)} \text{ [T-VAR]}$$

$$\frac{\text{G} \vdash e_1 : t_2 \text{ fi } t \qquad \text{G} \vdash e_2 : t_2}{\text{G} \vdash (e_1\ e_2) : t} \text{ [T-APP]}$$

36

Examples

37

Typing derivations

- To prove that a term has a type in some typing context, chain together a tree of instances of the typing rules, leading back to axioms
  - If can't make a derivation, then something isn't true

38

Examples

39

Formalizing variable lookup

- What does $\Gamma(x)$ mean?
- What if Γ includes several different types for x?

  $\Gamma = x{:}\ ,\ y{:}\ ,\ x{:} \rightarrow\ ,\ x{:}\ ,\ y{:} \rightarrow\ \rightarrow$
  - Can this happen?
  - If it can, what should it mean?
    - Any of the types is OK?
    - Just the leftmost? rightmost?
    - None are OK?

40

An example

- What context is built in the typing derivation for this expression?

  $\lambda x{:}\tau_1.\ (\lambda x{:}\tau_2.\ x)$
- What should the type of x in the body be?
- How should $\Gamma(x)$ be defined?

41

Formalizing using judgments

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau}\ \text{[T-VAR-1]}$$

$$\frac{\Gamma \vdash x : \tau \quad x \neq y}{\Gamma, y{:}\tau_2 \vdash x : \tau}\ \text{[T-VAR-2]}$$

- What about the $\Gamma = \{\}$ case?

42

Type-checking self-application

- What type should I give to *x* in this term?
  - l*x*:**?**. (*x x*)
- What type should I give to the *f* and *x*'s in *Y*?
  - *Y* " l*f*:**?**. (l*x*:**?**. *f* (*x x*)) (l*x*:**?**. *f* (*x x*))

43

Amazing fact #*N*+1: All simply-typed l-calculus exprs terminate!

- Cannot express looping or recursion in simply-typed l-calculus
  - Requires self-application, which requires recursive types, which simply-typed l-calculus doesn't have
- So all programs are guaranteed to never loop or recur, and terminate in a finite number of reduction steps!
  - (Simply-typed l-calculus could be a good basis for programs that must be guaranteed to finish, e.g. typecheckers, OS packet filters, …)

44

Adding an explicit recursion operator

- Several choices; here's one: add an expression "fix *e*"
- Define its reduction rule:
  - fix *e* fi b *e* (fix *e*)
- Define its typing rule:

G ├ *e* : *t* fi *t*
――――――――― [T-FIX]
G ├ (fix *e*) : *t*

45

Defining reduction precisely

- Use inference rules to define fi b redexes precisely

――――――――― [E-ABS]
(l*x*:*t*. e_1) e_2 fi b [*x* fi e_2]e_1

――――――――― [E-FIX]
fix *e* fi b *e* (fix *e*)

e_1 fi b e_1'
――――――――― [E-APP1]
e_1 e_2 fi b e_1' e_2

e_2 fi b e_2'
――――――――― [E-APP2]
e_1 e_2 fi b e_1 e_2'

e_1 fi b e_1'
――――――――― [E-BODY] *optional*
l*x*:*t*. e_1 fi b l*x*:*t*. e_1'

46

Formalizing evaluation order

- Can specify evaluation order by identifying which computations have been fully evaluated (have no redexes left), i.e., **values** *v*
  - one option:
    - *v* ::= l*x*:*t*. *e*
  - another option:
    - *v* ::= l*x*:*t*. *v*
  - what's the difference?

47

Example: call-by-value rules

v ::= l*x*:*t*. *e*

――――――――― [E-ABS]
(l*x*:*t*. e_1) $\boldsymbol{v_2}$ fi b [*x* fi $\boldsymbol{v_2}$]e_1

――――――――― [E-FIX]
fix ***v*** fi b ***v*** (fix ***v***)

e_1 fi b e_1'
――――――――― [E-APP1]
e_1 e_2 fi b e_1' e_2

e_2 fi b e_2'
――――――――― [E-APP2]
$\boldsymbol{v_1}$ e_2 fi b $\boldsymbol{v_1}$ e_2'

48

Type soundness

- What's the point of a static type system?
  - Identify inconsistencies in programs
    - Early reporting of possible bugs
  - Document (one aspect of) interfaces precisely
  - Provide info for more efficient compilation
- Most assume that type system "agrees with" evaluation semantics, i.e., is **sound**
  - Two parts to type soundness: preservation and progress

Preservation

- Type preservation: if an expression has a type, and that expression reduces to another expression/value, then that other expression/value has the same type
  - If G ├ *e* : *t* and *e* fi $_b$ *e'*, then G ├ *e'* : *t*
- Implies that types correctly "abstract" evaluation, i.e., describe what evaluation will produce

Progress

- If an expression successfully typechecks, then either the expression is a value, or evaluation can take a step
  - If G ├ *e* : *t*, then *e* is a *v* or $*e'* s.t. *e* fi $_b$ *e'*
- Implies that static typechecking guarantees successful evaluation without getting stuck
  - "well-typed programs don't go wrong"

Soundness

- Soundness = preservation + progress
  - If G ├ *e* : *t*, then *e* is a *v* or $*e'* s.t. *e* fi $_b$ *e'* and G ├ *e'* : *t*
    - preservation sets up progress, progress sets up preservation
- Soundness ensures a very strong match between evaluation and typechecking

Other ways to formalize semantics

- We've seen evaluation formalized using *small-step (structural) operational semantics*
- An alternative: *big step (natural) operational semantics*
  - Judgments of the form $e \Downarrow v$
    - "Expression *e* evaluates **fully** to value *v*"

Big-step call-by-value rules

$$\frac{}{(\mathrm{l}x{:}t.\ e) \Downarrow (\mathrm{l}x{:}t.\ e)}\ \text{[E-ABS]}$$

$$\frac{e_1 \Downarrow (\mathrm{l}x{:}t.\ e) \quad e_2 \Downarrow v_2 \quad ([x\,\mathrm{fi}\ v_2]e) \Downarrow v}{(e_1\ e_2) \Downarrow v}\ \text{[E-APP]}$$

$$\frac{e_1 \Downarrow (\mathrm{l}x{:}t.\ e) \quad ([x\,\mathrm{fi}\ (\mathrm{fix}\ (\mathrm{l}x{:}t.\ e))]e) \Downarrow v}{(\mathrm{fix}\ e_1) \Downarrow v}\ \text{[E-FIX]}$$

- Simpler, fewer tedious rules than small-step; "natural"
- Cannot easily prove soundness for non-terminating programs
- Typing judgments are "big step"; why?

Yet another variation

- Real machines and interpreters don't do *substitution* of values for variables when calling functions
  - Expensive!
- Instead, they maintain *environments* mapping variables to their values
  - A.k.a. stack frames
- We can formalize this
  - For big step, judgments of the form $\mathtt{r} \vdash e \Downarrow v$ where $\mathtt{r}$ is a list of $x=v$ bindings
    - "In environment $\mathtt{r}$, expr. e evaluates fully to value v"

Explicit environment rules

$$\frac{}{\mathtt{r} \vdash (\mathtt{l}x{:}t.\ e) \Downarrow (\mathtt{l}x{:}t.\ e)} \text{[E-ABS]}$$

$$\frac{\mathtt{r} \vdash e_1 \Downarrow (\mathtt{l}x{:}t.\ e) \quad \mathtt{r} \vdash e_2 \Downarrow v_2 \quad \mathbf{r, x=v_2} \vdash e \Downarrow v}{\mathtt{r} \vdash (e_1\ e_2) \Downarrow v} \text{[E-APP]}$$

$$\frac{\mathtt{r} \vdash e_1 \Downarrow (\mathtt{l}x{:}t.\ e) \quad \mathbf{r, x=(fix\ (l x{:}t.\ e))} \vdash e \Downarrow v}{\mathtt{r} \vdash (\text{fix } e_1) \Downarrow v} \text{[E-FIX]}$$

- Problems handling fix, since need to delay evaluation of recursive call
- Wrong! specifies dynamic scoping!

Explicit environments with closure values

$v ::= \langle \mathtt{l}x{:}t.\ e, \mathtt{r} \rangle$

$$\frac{}{\mathtt{r} \vdash (\mathtt{l}x{:}t.\ e) \Downarrow \mathbf{\langle (l x{:}t.\ e), r \rangle}} \text{[E-ABS]}$$

$$\frac{\mathtt{r} \vdash e_1 \Downarrow \mathbf{\langle (l x{:}t.\ e), r' \rangle} \quad \mathtt{r} \vdash e_2 \Downarrow v_2 \quad \mathbf{r', x=v_2} \vdash e \Downarrow v}{\mathtt{r} \vdash (e_1\ e_2) \Downarrow v} \text{[E-APP]}$$

- Does static scoping, as desired
- Allows formal reasoning about explicit environments
  - We found a bug in implementation of substitution via environments
- Makes proofs much more complicated

Other semantic frameworks

- We've seen several examples of *operational semantics*
  - Like specifying an interpreter, or a virtual machine
- An alternative: *denotational semantics*
  - Specifies the meaning of a term via *translation* into another (well-specified) language, usually mathematical functions
    - Like specifying a compiler!
  - More "abstract" than operational semantics
- Another alternative: *axiomatic semantics*
  - Specifies the result of expressions and effect of statements on properties known before and after
  - Suitable for formal verification proofs

Richer languages

- To gain experience formalizing language constructs, consider:
  - ints, bools
  - let
  - records
  - tagged unions
  - recursive types, e.g. lists
  - mutable refs

Basic types

- Enrich syntax:

  t ::= … | int | bool

  e ::= … | 0 | … | true | false
  | $e_1 + e_2$ | …
  | if e_1 then e_2 else e_3

  v ::= … | 0 | … | true | false

Add evaluation rules

- E.g., using big-step operational semantics

$$\frac{}{v \Downarrow v}\ \text{[E-VAL]} \qquad \textit{(generalizes E-ABS)}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1, v_2 \textit{ in Int} \quad v = v_1 + v_2}{(e_1 + e_2) \Downarrow v}\ \text{[E-PLUS]}$$

$$\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v_2}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \Downarrow v_2}\ \text{[E-IF-true]}$$

$$\frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v_3}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \Downarrow v_3}\ \text{[E-IF-false]}$$

- If no old rules need to be changed, then orthogonal
- \+ and if might not always reduce; evaluation can get **stuck**

61

Add typing rules

$$\frac{}{\Gamma \vdash 0 : \text{int}}\ \text{[T-INT]}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}\ \text{[T-TRUE]}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (e_1 + e_2) : \text{int}}\ \text{[T-PLUS]}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ then } e_3) : t}\ \text{[T-IF]}$$

- Type soundness: if e typechecks, then can't get stuck

62

Let

e ::= … | let x=e_1 in e_2

$$\frac{e_1 \Downarrow v_1 \quad ([x \mapsto v_1]e_2) \Downarrow v_2}{(\text{let } x = e_1 \text{ in } e_2) \Downarrow v_2}\ \text{[E-LET]}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x{:}t_1 \vdash e : t_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : t_2}\ \text{[T-LET]}$$

63

Records

- Syntax:

t ::= … | $\{n_1{:}t_1, …, n_n{:}t_n\}$

e ::= … | $\{n_1{=}e_1, …, n_n{=}e_n\}$ | $\#n\ e$

v ::= … | $\{n_1{=}v_1, …, n_n{=}v_n\}$

64

Evaluation and typing

$$\frac{e_1 \Downarrow v_1 \quad \dots \quad e_n \Downarrow v_n}{\{n_1{=}e_1, …, n_n{=}e_n\} \Downarrow \{n_1{=}v_1, …, n_n{=}v_n\}}\ \text{[E-RECORD]}$$

$$\frac{e \Downarrow \{n_1{=}v_1, …, n_n{=}v_n\}}{(\#n_i\ e) \Downarrow v_i}\ \text{[E-PROJ]}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \dots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \{n_1{=}e_1, …, n_n{=}e_n\} : \{n_1{:}t_1, …, n_n{:}t_n\}}\ \text{[T-RECORD]}$$

$$\frac{\Gamma \vdash e : \{n_1{:}t_1, …, n_n{:}t_n\}}{\Gamma \vdash (\#n_i\ e) : t_i}\ \text{[T-PROJ]}$$

65

Tagged unions

- A union of several cases, each of which has a tag
  - Type-safe: cannot misinterpret value under tag

    t ::= … | <$n_1{:}t_1$, …, $n_n{:}t_n$>

    e ::= … | <n=e>

    | case e of <n_1=x_1> => e_1 … <n_n=x_n> => e_n

    v ::= … | <n=v>
- Example:

```
val u:<a:int, b:bool> =
  if … then <a=3> else <b=true>
case u of
  <a=j> => j+4
  <b=t> => if t then 8 else 9
```

66

Evaluation and typing

$$\frac{e \Downarrow v}{\langle n{=}e \rangle \Downarrow \langle n{=}v \rangle}\ \text{[E-UNION]}$$

$$\frac{e \Downarrow \langle n_i{=}v_i \rangle \qquad ([x_i \mapsto v_i]e_i) \Downarrow v}{(\text{case } e \text{ of } \langle n_1{=}x_1 \rangle {=>} e_1 \ldots \langle n_n{=}x_n \rangle {=>} e_n) \Downarrow v}\ \text{[E-CASE]}$$

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash \langle n_i{=}e_i \rangle : \langle n_1{:}\tau_1, \ldots, n_n{:}\tau_n \rangle}\ \text{[T-UNION]}$$

$$\frac{\Gamma \vdash e : \langle n_1{:}\tau_1, \ldots, n_n{:}\tau_n \rangle \qquad \Gamma, x_1{:}\tau_1 \vdash e_1 : \tau \ \ldots\ \Gamma, x_n{:}\tau_n \vdash e_n : \tau}{\Gamma \vdash (\text{case } e \text{ of } \langle n_1{=}x_1 \rangle {=>} e_1 \ldots \langle n_n{=}x_n \rangle {=>} e_n) : \tau}\ \text{[T-CASE]}$$

- Where get the full type of the union in T-UNION?

67

Lists

- Use tagged unions to define lists:
  - int_list ≡ <nil: unit, cons: {hd:int, tl:int_list}>
- But int_list is defined recursively
  - As with recursive function definitions, need to carefully define what this means

68

Recursive types

- Introduce a recursive type: $\mu X.\ \tau$
  - τ can refer to X to mean the whole type, recursively
    - int_list ≡ **μL.**<nil: unit, cons: {hd:int, tl:**L**}>
  - This type means the infinite tree of "unfoldings" of the recursive reference
  - If τ contains a union type with non-recursive cases (base cases for the recursively defined type), then can have finite values of this "infinite" type
    - <nil=()>
    - <cons={hd=3, tl=<nil=()>}>
    - <cons={hd=3, tl=<cons={hd=4, tl=<nil=()>}>}>
    - …

69

Folding and unfolding

- What values have recursive types?
  What can we do with a value of recursive type?
  - Can take a value of the body of the recursive type, and "fold" it up to make a recursive type

    int_list ≡ μL.<nil: unit, cons: {hd:int, tl:L}>
    <nil=()> : <nil: unit, cons: {hd:int, tl:int_list}>
    fold <nil=()> : int_list

  - Can "unfold" it to do the reverse
    - Exposes the underlying type, so operations on it typecheck
- Can introduce fold & unfold expressions, or can make when to do folding & unfolding implicit

70

Typing of fold and unfold

$$\frac{\Gamma \vdash e : [X \mapsto (\mu X.\tau)]\tau}{\Gamma \vdash (\text{fold } e) : \mu X.\ \tau}\ \text{[T-FOLD]}$$

$$\frac{\Gamma \vdash e : \mu X.\ \tau}{\Gamma \vdash (\text{unfold } e) : [X \mapsto (\mu X.\tau)]\tau}\ \text{[T-UNFOLD]}$$

- Evaluation ignores fold & unfold

71

Using recursive values and types

- double: double all elems of an int_list

```
int_list ≡ μL.<nil: unit, cons: {hd:int, tl:L}>
double ≡ fix (λdouble:(int_list→ int_list).
  λlst:int_list.
    case (unfold lst) of
      <nil=x> => fold <nil=()>
      <cons=r> =>
        fold <cons={hd=(#hd r) + (#hd r),
                    tl=double (#tl r)}>)
```

72

References and mutable state

- Syntax:
  - τ ::= … | τ ref
  - e ::= … | ref e | ! e | e_1 := e_2
  - v ::= … | ref v
- Typing:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\text{ref } e) : \tau\ \text{ref}}\ [\text{T-REF}]$$

$$\frac{\Gamma \vdash e : \tau\ \text{ref}}{\Gamma \vdash (!\ e) : \tau}\ [\text{T-DEREF}]$$

$$\frac{\Gamma \vdash e_1 : \tau\ \text{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 := e_2) : \text{unit}}\ [\text{T-ASSIGN}]$$

Evaluation of references

$$\frac{e \Downarrow v}{(\text{ref } e) \Downarrow (\text{ref } v)}\ [\text{E-REF}]$$

$$\frac{e \Downarrow (\text{ref } v)}{(!\ e) \Downarrow v}\ [\text{E-DEREF}]$$

$$\frac{e_1 \Downarrow (\text{ref } v_1) \quad e_2 \Downarrow v_2}{(e_1 := e_2) \Downarrow \text{unit}}\ [\text{E-ASSIGN}]$$

- But where'd the assignment go?

Example

(let r = ref 0 in (let x = (r := 2) in (! r)))

Stores

- Introduce a *store* σ to keep track of the contents of references
  - A map from *locations* to values
    - "ref e" allocates a new location and initializes it with (the result of evaluating) e
    - "! e" looks up the contents of the location (resulting from evaluating) e in the store
    - "e_1 := e_2" updates the location (resulting from evaluating) e_1 to hold (the result of evaluating) e_2, returning the updated store
- Evaluation now passes along the current store in which to evaluate expressions
  - Big-step judgments of the form $<e,\sigma> \Downarrow <v,\sigma'>$

Big-step semantics with stores

$$\frac{}{<v,\sigma> \Downarrow <v,\sigma>}\ [\text{E-VAL}]$$

$$\frac{<e_1,\sigma> \Downarrow <(\lambda x{:}\tau.\ e),\sigma'> \quad <e_2,\sigma'> \Downarrow <v_2,\sigma''> \quad <([x \mapsto v_2]e),\sigma''> \Downarrow <v,\sigma'''>}{<(e_1\ e_2),\sigma> \Downarrow <v,\sigma'''>}\ [\text{E-APP}]$$

Semantics of references

- Add locations l as a new kind of value (not "ref v")
  - v ::= … | l
- New semantics

$$\frac{<e,\sigma> \Downarrow <v,\sigma'> \quad l \notin \text{dom}(\sigma') \quad \sigma'' = \sigma'[l \mapsto v]}{<(\text{ref } e),\sigma> \Downarrow <l,\sigma''>}\ [\text{E-REF}]$$

$$\frac{<e,\sigma> \Downarrow <l,\sigma'> \quad v = \sigma'(l)}{<(!\ e),\sigma> \Downarrow <v,\sigma'>}\ [\text{E-DEREF}]$$

$$\frac{<e_1,\sigma> \Downarrow <l,\sigma'> \quad <e_2,\sigma'> \Downarrow <v_2,\sigma''> \quad \sigma''' = \sigma''[l \mapsto v_2]}{<(e_1 := e_2),\sigma> \Downarrow <\text{unit},\sigma'''>}\ [\text{E-ASSIGN}]$$

Example again

(let r = ref 0 in (let x = (r := 2) in (! r)))

Summary, so far

- Now have also seen simply typed λ-calculus
  - Saw inference rules, derivations
  - Saw several ways to formalize operational semantics and typing rules
- Saw many extensions to this core language
  - Typical of how real PL theorists work
  - Usually orthogonal to underlying semantics
  - References required redoing underlying semantics
- Would you want to use this language?
  - If it had suitable syntactic sugar?

Polymorphic types

- Simply typed λ-calculus is "simply typed", i.e., it has no polymorphic or parameterized types
- "Good" programming languages have polymorphic types
  - And there are tricky issues relating to polymorphic types
- So we'd like to capture the essense of polymorphic types in our calculus
  - So we'll really understand it

Polymorphic λ-calculus

- Also known as **System F**
- Extend type syntax with a forall type

  τ ::= … | ∀*X*. *τ* | *X*
- Now can write down the types of polymorphic values

  id ″ ∀T. T→ T

  map ″ ∀'a. ∀'b. ('a→ 'b)→ 'a list→ 'b list

  nil ″ ∀T. T list

Values of polymorphic type

- Introduce explicit notation for values of polymorphic type and their instantiations
  - A polymorphic value: Λ*X*. *e*
    - Λ*X*. *e* is a function that, given a type *τ*, gives back *e* with *τ* substituted for *X*
  - Use such values by instantiating them: *e*[*τ*]
    - *e*[*τ*] is like function application
- Syntax:

  e ::= … | Λ*X*. *e* | *e*[*τ*]

  v ::= … | Λ*X*. *e*

An example

(* fun id x = x; *id : 'a->'a* *)

id ″ Λ'a. λx:'a. x

: ∀'a. 'a→ 'a

id [int] 3 $\rightarrow_\beta$

(λx:int. x) 3 $\rightarrow_\beta$

3

id [bool] $\rightarrow_\beta$

λx:bool. x

Another example

(* fun doTwice f x = f (f x);
doTwice: ('a->'a)->'a->'a *)

doTwice ≡ Λ'a. λf:'a→'a. λx:'a. f (f x)
: ∀'a. ('a→'a)→'a→'a

doTwice [int] succ 3 $\to_\beta$
(λf:int→int. λx:int. f (f x)) succ 3 $\to_\beta^*$
succ (succ 3) $\to_\beta^*$
3

85

Yet another example

map ≡ Λ'a. Λ'b. fix (λmap:('a→'b)→'a list→'b list.
λf:'a→'b. λlst:'a list.
fold (case (unfold lst) of
<nil=n> => <nil=()>
<cons=r> => <cons={hd=f (#hd r),
tl=map f (#tl r)}>))
: ∀'a. ∀'b. ('a→'b)→'a list→'b list

map [int] [bool] isZero [3,0,5] $\to_\beta^*$ [false,true,false]

- ML infers what the Λ*T* and [*t*] should be

86

A final example

(* fun cool f = (f 3, f true) *)
cool ≡ λf:(Λ'a. 'a→'a). (f [int] 3, f [bool] true)
*: (Λ'a. 'a→'a)→(int * bool)*

cool id $\to_\beta$
(*id* [int] 3, *id* [bool] true) $\to_\beta^*$
((λx:int. x) 3, (λx:bool. x) true) $\to_\beta^*$
(3, true)

- Note: Λ inside of λ and →
  - Can't write this in ML; not "prenex" form

87

Evaluation and typing rules

- Evaluation:

$$\frac{e \Downarrow (\Lambda X.\ e_1) \quad ([X \mapsto t]e_1) \Downarrow v}{(e[t]) \Downarrow v}\ \text{[E-INST]}$$

- Typing:

$$\frac{\Gamma, X{:}\text{type} \vdash e : t}{\Gamma \vdash (\Lambda X.\ e) : \forall X.\ t}\ \text{[T-POLY]}$$

$$\frac{\Gamma \vdash e : \forall X.\ t'}{\Gamma \vdash (e[t]) : [X \mapsto t]t'}\ \text{[T-INST]}$$

88

Different kinds of functions

- λ*x*. *e* is a function from *values* to *values*
- Λ*X*. e is a function from *types* to *values*
- What about functions from *types* to *types*?
  - **Type constructors** like →, list, BTree
    - We want them!
- What about functions from values to types?
  - **Dependent types** like the type of arrays of length n, where n is a run-time computed value
    - Pretty fancy, but would be very cool

89

Type constructors

- What's the "type" of *list*?
  - Not a simple type, but a function from types to types
    - e.g. list(int) = int_list
  - There are lots of type constructors that take a single type and return a type
    - They all have the same "meta-type"
  - Other things take two types and return a type (e.g. →, assoc_list)
- A "meta-type" is called a **kind**

90

Kinds

- A *type* describes a *set of values* or value constructors (a.k.a. functions) with a common structure
  - $\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \ldots$
- A *kind* describes a *set of types* or type constructors with a common structure
  - $\kappa ::= \text{type} \mid \kappa_1 \Rightarrow \kappa_2$
- Write $\tau :: \kappa$ to say that a type τ has kind κ
  - int :: type
  - int→ int :: type
  - list :: type ⇒ type
  - list int :: type
  - assoc_list :: type ⇒ type ⇒ type
  - assoc_list string int :: type

91

Kinded polymorphic λ-calculus

- Also called **System F_ω**
- Full syntax:
  - $\boldsymbol{\kappa ::= \text{type} \mid \kappa_1 \Rightarrow \kappa_2}$
  - $\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \forall X{::}\kappa.\ \tau \mid X \mid \boldsymbol{\lambda X{::}\kappa.\ \tau \mid \tau_1\ \tau_2}$
  - $e ::= \lambda x{:}\tau.\ e \mid x \mid e_1\ e_2 \mid \Lambda X{::}\kappa.\ e \mid e[\tau]$
  - $v ::= \lambda x.\ e \mid \Lambda X{::}\kappa.\ e$
  - Functions and applications at both the value and the type level
  - Arrows at both the type and kind level

92

Examples

pair ≡
λ'a::type. λ'b::type. {first:'a, second:'b}
:: type ⇒ type ⇒ type

pair int bool "→$_\beta$" {first:int, second:bool}

{first=5, second=true} : *pair* int bool

swap ≡
Λ'a::type. Λ'b::type.
λp:pair 'a 'b.
{first=#second p, second=#first p}
: ∀'a::type. ∀'b::type. (pair 'a 'b) → (pair 'b 'a)

93

Expression typing rules

$$\frac{\Gamma \vdash \tau_1{::}\text{type} \quad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x{:}\tau_1.\ e) : \tau_1 \rightarrow \tau_2}\ [\text{T-ABS}]$$

$$\frac{\Gamma, X{::}\kappa \vdash e : \tau}{\Gamma \vdash (\Lambda X{::}\kappa.\ e) : \forall X{::}\kappa.\ \tau}\ [\text{T-POLY}]$$

$$\frac{\Gamma \vdash e : \forall X{::}\kappa.\ \tau' \quad \Gamma \vdash \tau{:}\kappa}{\Gamma \vdash (e[\tau]) : [X \mapsto \tau]\tau'}\ [\text{T-INST}]$$

(T-VAR and T-APP unchanged)

94

Type kinding rules

$$\frac{}{\Gamma \vdash \text{int} :: \text{type}}\ [\text{K-INT}] \qquad \frac{\Gamma \vdash \tau_1 :: \text{type} \quad \Gamma \vdash \tau_2 :: \text{type}}{\Gamma \vdash (\tau_1 \rightarrow \tau_2) :: \text{type}}\ [\text{K-ARROW}]$$

$$\frac{\Gamma, X{::}\kappa \vdash \tau :: \text{type}}{\Gamma \vdash (\forall X{::}\kappa.\ \tau) :: \text{type}}\ [\text{K-FORALL}] \qquad \frac{}{\Gamma \vdash X :: \Gamma(X)}\ [\text{K-VAR}]$$

$$\frac{\Gamma, X{::}\kappa_1 \vdash \tau :: \kappa_2}{\Gamma \vdash (\lambda X{::}\kappa_1.\ \tau) :: \kappa_1 \rightarrow \kappa_2}\ [\text{K-ABS}] \qquad \frac{\Gamma \vdash \tau_1 :: \kappa_2 \rightarrow \kappa \quad \Gamma \vdash \tau_2 :: \kappa_2}{\Gamma \vdash (\tau_1\ \tau_2) :: \kappa}\ [\text{K-APP}]$$

95

Summary

- Saw ever more powerful static type systems for the λ-calculus
  - Simply typed λ-calculus
  - Polymorphic λ-calculus, a.k.a. System F
  - Kinded poly. λ-calculus, a.k.a. System F_ω
- Exponential ramp-up in power, once build up sufficient critical mass
- Real languages typically offer some of this power, but in restricted ways
  - Could benefit from more expressive approaches

96

Other uses

- Compiler internal representations for advanced languages
  - E.g. FLINT: compiles ML, Java, …
- Checkers for interesting non-type properties, e.g.:
  - proper initialization
  - static null pointer dereference checking
  - safe explicit memory management
  - thread safety, data-race freedom