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 λ-calculus: syntax
- (Abstract) syntax:
  - \( e ::= x \) variable
  - \( \lambda x. e \) function/abstraction
  - \( e_1 e_2 \) 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 \( \lambda \) function.
  
  \[
  \text{freeVars}(x) = x \\
  \text{freeVars}(\lambda x. e) = \text{freeVars}(e) - \{x\} \\
  \text{freeVars}(e_1 e_2) = \text{freeVars}(e_1) \cup \text{freeVars}(e_2)
  \]
- \( e \) is **closed** if \( \text{freeVars}(e) = \{\} \)

a-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
- **\( \alpha \)-equivalent** term trees
  - \( (\lambda x. x) x \), \( (\lambda y. x) y \)
  - cannot rename free variables
- **term \( e \)** and all \( \alpha \)-equivalent term trees
  - Can freely rename bound vars whenever helpful

Evaluation: \( \beta \)-reduction

- Define what it means to "run" a lambda-calculus program by giving simple reduction/rewriting/simplification rules
  - "\( e \) evaluates to \( e' \)" means "\( e \) reduces to \( e' \) in one step"
- One case:
  - \( (\lambda x. e) e_2 \) reduces to \( [x \mapsto e_2]e \)
  - "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
  
  \[
  [x \mapsto e] x = e \\
  [x \mapsto e] y = y \text{ if } x \neq y \\
  [x \mapsto e] (\lambda x. e) = (\lambda x. e) \\
  [x \mapsto e] (\lambda y. e) = (\lambda y. [x \mapsto e] e) \text{ if } x \neq y \\
  [x \mapsto e] e_1 e_2 = ([x \mapsto e] e_1) ([x \mapsto e] e_2)
  \]
- Can use \( \alpha \)-renaming to ensure "\( y \) not free in \( e' \)"

Result of reduction

- To fully evaluate a lambda calculus term, simply perform \( \beta \)-reduction until you can't any more
  - \( \text{refl} \) = reflexive, transitive closure of \( \beta \)
- 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?

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)

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

- Thm. If \( e_1 \overset{\beta}{\rightarrow} e_2 \) and \( e_1 \overset{\beta}{\rightarrow} e_3 \), then
  \[ e_4 \] such that \( e_2 \overset{\beta}{\rightarrow} e_4 \) and \( e_3 \overset{\beta}{\rightarrow} e_4 \)

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

Existence of normal forms?

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

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:
  - \((\lambda x \lambda y. x y)((\lambda x. x x)(\lambda x. x x))\)

Weak head normal form

- What should this evaluate to?
  \((\lambda x. (\lambda x. x x)(\lambda x x))\)
  - 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")
Amazing fact #3:
1-calculus is Turing-complete!

- But the 1-calculus is too weak, right?
  - No multiple arguments!
  - No numbers or arithmetic!
  - No booleans or if!
  - No data structures!
  - No loops or recursion!

Multiple arguments: currying

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

\[
\begin{align*}
1(x_1, x_2), e & \Rightarrow 1x_1 (1x_2, e) \quad (\ast 1x_1, x_2, e) \\
\kappa(e_1, e_2) & \Rightarrow (\kappa e_1) e_2
\end{align*}
\]

Church numerals

- Encode natural numbers using stylized lambda terms

\[
\begin{align*}
\text{zero} & \coloneqq \lambda s. \lambda z. z \\
\text{one} & \coloneqq \lambda s. \lambda z. s z \\
\text{two} & \coloneqq \lambda s. \lambda z. s (s z) \\
\vdots \\
\text{\vdots} \\
\end{align*}
\]

- A unary encoding using functions
  - No stranger than binary encoding

Arithmetic on Church numerals

- Successor function:
  take (the encoding of) a number, return (the encoding of) its successor

\[
\begin{align*}
succ & \coloneqq \lambda n. \lambda s. \lambda z. s (n s z) \\
succ \text{zero} & \coloneqq \lambda s. \lambda z. s z \\
succ \text{two} & \coloneqq \lambda s. \lambda z. s (two s z) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{if} & \coloneqq \lambda b. \lambda t. \lambda e. b \ e t e \\
\text{true} & \coloneqq \lambda t. \lambda e. t \\
\text{false} & \coloneqq \lambda t. \lambda e. e \\
\text{if false four six} & \coloneqq \lambda s. \lambda z. s (s (s (s z))) \\
\text{false four six} & \coloneqq \lambda s. \lambda z. s (s (s (s z))) \\
\text{six} & \coloneqq \lambda s. \lambda z. s (s (s (s (s (s z)))))
\end{align*}
\]

Addition

- To add \(x\) and \(y\), apply \textit{succ} to \(y\) \(x\) times

\[
\begin{align*}
\text{plus} & \coloneqq \lambda x. \lambda y. x \text{ succ } y \\
\text{plus two three} & \coloneqq \lambda z. z s (s (s z)) \\
\text{two succ three} & \coloneqq \lambda z. s (s (s (s z))) \\
\text{succ (succ three)} & \coloneqq \lambda z. s (s (s (s (s (s z))))
\end{align*}
\]

- Multiplication is repeated addition, similarly

Booleans

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

\[
\begin{align*}
\text{if} & \coloneqq \lambda b. \lambda t. \lambda e. b \ e t e \\
\text{true} & \coloneqq \lambda t. \lambda e. t \\
\text{false} & \coloneqq \lambda t. \lambda e. e \\
\text{if false four six} & \coloneqq \lambda s. \lambda z. s (s (s (s z))) \\
\text{false four six} & \coloneqq \lambda s. \lambda z. s (s (s (s z))) \\
\text{six} & \coloneqq \lambda s. \lambda z. s (s (s (s (s (s z)))))
\end{align*}
\]
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 : n. n (l.x. false) true
  ```

- isZero zero

- isZero two

Data structures

- Try to encode simple pairs
- Can build more complex data structures out of them
  ```
  mkPair : l.f. l.s. l.x. x f s
  ```

- first : p. p (l.f. l.s. f)

- second : p. p (l.f. l.s. s)

Loops and recursion

- 1-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:
  ```
  Yf = (l.x. f(x x)) (l.x. f(x x)) = f(Y f)
  ```

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

Y for factorial

- fact two
  ```
  (Y factG) two
  ```

- factG (Y factG) two
  ```
  if (isZero two) one (times two ((Y factG) (minus two one)))
  ```

- times two ((Y factG) (minus two one))
  ```
  times two (Y factG) one
  ```

- times two (Y factG) one

- times two (Y factG) one
  ```
  times two ((Y factG) (minus zero one)))
  ```

- times two (Y factG) one

- times two (Y factG) one
  ```
  times two (Y factG) one
  ```

- times two (Y factG) one

- times two (Y factG) one
  ```
  times two (Y factG) one
  ```

- times two (Y factG) one

- times two (Y factG) one
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

Summary, so far

- Saw untyped λ-calculus syntax
- Saw some rewriting rules, which defined the semantics of λ-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 λ-calculus to encode lots of higher-level constructs

Simply-typed lambda calculus

- Now, let’s add static type checking
- Extend syntax with types:
  - $t ::= t_1 \cdot t_2 | e \cdot t_1 \cdot e | x \cdot e_1 \cdot e_2$
  - (The dot is just the base case for types, to stop the recursion. Values of this type will never be invoked, just passed around.)

Typing judgments

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

Typing rules

- Give typechecking rule(s) for each kind of expression
- Write as a logical inference rule
  - premise, … premise, (α i 0)
  - 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

Typing rules for simply-typed λ-calculus

- $\alpha, \Gamma, e \vdash e : t$
  - [T-ABS]
  - $\alpha \vdash (\lambda x : t \cdot \phi) \cdot e_1 : \phi$
    - [T-VAR]
    - $\alpha \vdash x : \phi$\n  - $\alpha \vdash e_1 \cdot e_2 : t$
    - [T-APP]
    - $\alpha \vdash e_1 : e_2 : t$
Examples

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

Examples

Formalizing variable lookup
- What does $G(x)$ mean?
- What if $G$ includes several different types for $x$?
  - $G = x, y, x : \text{fi}, x, y : \text{fi}$
  - Can this happen?
  - If it can, what should it mean?
    - Any of the types is OK?
    - Just the leftmost? rightmost?
    - None are OK?

An example
- What context is built in the typing derivation for this expression?
  - $\lambda x : t_1. (\lambda x : t_2. x)$
- What should the type of $x$ in the body be?
- How should $G(x)$ be defined?

Formalizing using judgments
- $\frac{}{G, x : \varepsilon \vdash x : \varepsilon}$
  - $[\text{T-VAR-1}]$
- $\frac{G \vdash x : \varepsilon}{G, y : t_2 \vdash x : \varepsilon}$
  - $[\text{T-VAR-2}]$
- What about the $G = \emptyset$ case?
Type-checking self-application

- What type should I give to \( x \) in this term?
  \( \lambda x : ? . (x \; x) \)

- What type should I give to the \( f \) and \( x \)'s in \( Y \)?
  \( Y = \lambda f : ? . (\lambda x : ? . f (x \; x)) \; (\lambda x : ? . f (x \; x)) \)

Adding an explicit recursion operator

- Several choices; here's one:
  add an expression "fix \( e \)"

- Define its reduction rule:
  \[ \text{fix } e :: e (\text{fix } e) \]

- Define its typing rule:
  \[ \begin{align*}
    G \vdash e : ? \\
    G \vdash (\text{fix } e) : ? 
  \end{align*} \]

Defining reduction precisely

- Use inference rules to define \( \text{fix }_f \) redexes precisely
  \[ \begin{align*}
    (\lambda x : ? . e_1) \; v_2 & \rightarrow (\lambda x : ? . e_1) \; v_2 \\
    e_1 \; \text{fix }_f \; e_2 & \rightarrow e_1 \; (\text{fix }_f e_2) \\
    (\text{fix }_f e) \; e_2 \; e_3 & \rightarrow (\lambda x : ? . e) \; \text{fix }_f \; e \; e_2 \; e_3
  \end{align*} \]

Example: call-by-value rules

- Can specify evaluation order by identifying which computations have been fully evaluated (have no redexes left), i.e., values \( V \)
  - one option:
    \[ V ::= \lambda x : ? . e \]
  - another option:
    \[ V ::= \lambda x : ? \cdot V \]
  - what's the difference?
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 \( \Gamma \vdash e : \tau \) and \( e \rightarrow e' \), then \( \Gamma \vdash e' : \tau \)
- 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 \( \Gamma \vdash e : \tau \), then \( e \) is a value or \( e \) is a value and \( \Gamma \vdash e' : \tau \)
- Implies that static typechecking guarantees successful evaluation without getting stuck
  - "well-typed programs don't go wrong"

Soundness

- Soundness = preservation + progress
  - If \( \Gamma \vdash e : \tau \) then \( e \) is a value or \( \exists e' \text{ s.t. } e \rightarrow e' \) and \( \Gamma \vdash e' : \tau \)
- 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 \( e \Downarrow \nu \)
  - "Expression \( e \) evaluates fully to value \( \nu \)"

Big-step call-by-value rules

\[
\begin{align*}
\lambda x : \tau. e & \Downarrow (\lambda x : \tau. e) \Downarrow (\lambda x : \tau. e) \\
\text{[E-ABS]} \\
\text{\( e \Downarrow \nu \)} & \Downarrow (\text{fix (\lambda x : \tau. e)}) \Downarrow \nu \\
\text{[E-FIX]} \\
\text{\( (\text{fix } \nu) \Downarrow \nu \)} & \Downarrow v \\
\text{[E-FIX]}
\end{align*}
\]

- 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 $r \vdash e \Downarrow v$
  - "In environment $r$, expr. $e$ evaluates fully to value $v"

Explicit environment rules

\[
\begin{align*}
r \vdash (\lambda x.e) \Downarrow (\lambda x.e) \\
(r \vdash e_1 \Downarrow v_1) \quad (r \vdash e_2 \Downarrow v_2) \quad e \Downarrow v & \quad \text{[E-APP]} \\
r \vdash e_1 \Downarrow (\lambda x.e) & \quad r, x=v_2 \vdash e \Downarrow v & \quad \text{[E-FIX]} \\
\end{align*}
\]

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

Explicit environments with closure values

\[
v ::= \langle l x: t.e, r \rangle
\]

\[
\begin{align*}
r \vdash \langle l x: t.e, e \rangle & \quad \text{[E-ABS]} \\
r \vdash e_1 \Downarrow \langle l x: t.e, r \rangle & \quad r \vdash e_2 \Downarrow \langle l x: t.e, r \rangle & \quad v \Downarrow \langle l x: t.e, r \rangle & \quad \text{[E-APP]} \\
r \vdash \langle l x: t.e, e \rangle \Downarrow v & \quad \text{[E-FIX]} \\
\end{align*}
\]

- 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:
  \[
  \begin{align*}
  \tau & ::= \ldots \mid \text{int} \mid \text{bool} \\
  e & ::= \ldots \mid 0 \mid \ldots \mid \text{true} \mid \text{false} \\
  & \quad | \ e_1 \ e_2 \ | \ldots \\
  & \quad | \ \text{if } e_1 \ \text{then } e_2 \ \text{else } e_3 \\
  v & ::= \ldots \mid 0 \mid \ldots \mid \text{true} \mid \text{false}
  \end{align*}
  \]
Add evaluation rules

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

\[ \vdash v : v \quad \text{(generalizes E-ABS)} \]

\[ \vdash v_1 \quad \text{(E-VAL)} \]

\[ \vdash v_2 \quad \text{(E-PLUS)} \]

\[ \vdash v_3 \quad \text{(E-IF-tru)} \]

\[ \vdash v_4 \quad \text{(E-IF-fal)} \]

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

Add typing rules

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

Let

\[ e ::= \ldots \mid \text{let } x=e_1 \text{ in } e_2 \]

\[ e_1 \Downarrow v_1 \quad \text{(E-LET)} \]

\[ (\text{let } x=e_1 \text{ in } e_2) \Downarrow v_2 \]

\[ \vdash e_1 : t_1 \quad \vdash x : t_2 \quad \vdash e : t_2 \quad \text{(T-LET)} \]

Evaluation and typing

\[ e_1 \Downarrow v_1 \quad \vdash e_1 : t_1 \quad \text{(E-RECORD)} \]

\[ (e_1, e_2, \ldots, e_n) \Downarrow (t_1, t_2, \ldots, t_n) \]

\[ \vdash (e_1, e_2, \ldots, e_n) : (t_1, t_2, \ldots, t_n) \quad \text{(E-RECORD)} \]

\[ \vdash e : (t_1, t_2, \ldots, t_n) \quad \text{(T-RECORD)} \]

\[ \vdash (e_1, e_2, \ldots, e_n) : t_1 \quad \text{(T-RECORD)} \]

\[ \vdash e : t \quad \text{(T-RECORD)} \]

Tagged unions

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

\[ e ::= \ldots \mid <n_1=e_1, \ldots, n_m=e_m> \]

\[ \text{val } u : \text{int} \cdot \text{bool} = \text{if } a=3 \text{ then } v \text{ else } b \]

\[ \text{case } u \text{ of } <a=x> \Rightarrow \text{if } a=3 \text{ then } v \text{ else } b \]

\[ \text{if } b=\text{true} \Rightarrow \text{if } t \text{ then } 8 \text{ else } 9 \]

Records

- Syntax:

\[ t ::= \ldots \mid \{n_1=t_1, \ldots, n_m=t_m\} \]

\[ e ::= \ldots \mid \{n_1=e_1, \ldots, n_m=e_m\} \mid \#n e \]

\[ v ::= \ldots \mid \{n_1=v_1, \ldots, n_m=v_m\} \]
### Evaluation and typing

```
\[ \frac{a \downarrow \nu \quad \text{[E-UNION]}}{a \downarrow \nu \times \nu} \]
\[ \frac{\vdash a \downarrow \nu \times \nu \quad \text{[E-CASE]}}{\vdash \text{case } a \text{ of } \langle n_1 = x_1 \Rightarrow e_1 \ldots n_n = x_n \Rightarrow e_n \rangle \downarrow \nu} \]
\[ \frac{\vdash a \downarrow \nu}{\vdash e : \nu} \quad \text{[T-UNION]} \]
\[ \frac{\vdash a \downarrow \nu \times \nu \quad \text{[T-CASE]}}{\vdash \text{case } a \text{ of } \langle n_1 = x_1 \Rightarrow e_1 \ldots n_n = x_n \Rightarrow e_n \rangle} \]
```

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

### Lists

- Use tagged unions to define lists:
  
  \[
  \text{int_list} = \langle \text{nil: unit, cons: (hd: int, tl: int_list)} \rangle
  \]

- But int_list is defined recursively
  
  - As with recursive function definitions, need to carefully define what this means

### Recursive types

- Introduce a recursive type: \( mX. \, \varepsilon \)
  
  - \( \varepsilon \) can refer to \( X \) to mean the whole type, recursively
    
    \[
    \text{int_list} = mX \langle \text{nil: unit, cons: (hd: int, tl: L)} \rangle
    \]
  
  - This type means the infinite tree of "unfoldings" of the recursive reference

  - If \( \varepsilon \) contains a union type with non-recursive cases (base cases for the recursively defined type), then can have finite values of this "infinite" type

    \[
    \text{<nil=x> \Rightarrow fold <nil=()>} \quad \text{<cons=r> \Rightarrow fold <cons={hd=(#hd r) + (#hd r), tl=double (#tl r)}>}
    \]

    ...

### Folding and unfolding

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

    \[
    \text{int_list} = L \langle \text{nil: unit, cons: (hd: int, tl: L)} \rangle
    \]

    \[
    \langle \text{nil=}() \Rightarrow \langle \text{nil=}(), \text{ cons=}() \Rightarrow \text{int_list} \rangle
    \]

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

### Typing of fold and unfold

- Evaluation ignores fold & unfold

### Using recursive values and types

- double: double all elems of an int_list

  \[
  \text{int_list} = mX \langle \text{nil: unit, cons: (hd: int, tl: L)} \rangle
  \]

  - double \( = \) fix \( \langle \text{double: (int_list: int_list)}. \text{1st:int_list.}

    \text{case (unfold lst) of}
  
    \[
    \langle \text{nil=}x \Rightarrow \text{fold <nil=}() \rangle
    \]

    \[
    \langle \text{cons=}r \Rightarrow \text{fold <cons=} (hd=double (#hd r), tl=double (#tl r)) \rangle
    \]
References and mutable state

- Syntax:
  \[
  \begin{align*}
  t &::= \ldots \mid \text{ref} \\
  e &::= \ldots \mid \text{ref} \ e \mid e_1 := e_2 \\
  v &::= \ldots \mid \text{ref} \ v
  \end{align*}
  \]

- Typing:
  \[
  \begin{array}{c}
  \frac{\vdash \ e : t}{\vdash (\text{ref} \ e) : t\text{-ref}} \quad \text{[T-REF]} \\
  \frac{\vdash \ e : t\text{-ref}}{\vdash (! \ e) : t} \quad \text{[T-DEREF]} \\
  \frac{\vdash \ e_1 : t\text{-ref} \quad \vdash \ e_2 : t}{\vdash (e_1 := e_2) : \text{unit}} \quad \text{[T-ASSIGN]}
  \end{array}
  \]

Evaluation of references

\[
\begin{align*}
\frac{e \Downarrow v}{(\text{ref} \ e) \Downarrow (\text{ref} \ v)} \quad \text{[E-REF]} \\
\frac{e \Downarrow (\text{ref} \ v)}{(! \ e) \Downarrow v} \quad \text{[E-DEREF]} \\
\frac{(e_1 := e_2) \Downarrow \text{unit}}{(e_1 := e_2) \Downarrow \text{unit}} \quad \text{[E-ASSIGN]}
\end{align*}
\]

- But where'd the assignment go?

Example

\[
\begin{align*}
(\text{let } r = \text{ref} \ 0 \ \text{in} \ (\text{let } x = (r := 2) \ \text{in} \ (！r)))
\end{align*}
\]

Stores

- Introduce a store \( s \) 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 semantics with stores

\[
\begin{align*}
\frac{\langle \nu_0 \rangle \Downarrow \langle \nu_0 \rangle}{\langle e_0, s \rangle \Downarrow \langle l, \nu_0 \rangle} \quad \text{[E-VAL]} \\
\langle e_0, s \rangle \Downarrow \langle (l_1, e_0), \nu_0' \rangle \\
\langle e_0, s \rangle \Downarrow \langle (l_2, e), \nu_0'' \rangle \\
\langle \nu_0'' \rangle \Downarrow \langle v, \nu_0''' \rangle
\end{align*}
\]

Semantics of references

- Add locations \( l \) as a new kind of value (not "ref \( v \)"
  \[
  \begin{align*}
  \frac{\langle e_0, s \rangle \Downarrow \langle l_1, \nu_0 \rangle \quad \dom(s) = \{ e[v] \} }{\langle \text{ref} \ e_0, s \rangle \Downarrow \langle l_1, \nu_0 \rangle} \quad \text{[E-REF]} \\
  \frac{\langle e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle \quad \nu = s[l]}{\langle \text{ref} \ e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle} \quad \text{[E-DEREF]} \\
  \frac{\langle e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle \quad \langle e_0, s \rangle \Downarrow \langle l_3, \nu_0' \rangle \quad s'' = s[v]}{\langle e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle} \quad \text{[E-ASSIGN]}
  \end{align*}
  \]

- New semantics
  \[
  \begin{align*}
  \frac{\langle e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle \quad \dom(s) = \{ e[v'] \} }{\langle \text{ref} \ e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle} \quad \text{[E-REF]} \\
  \frac{\langle e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle \quad \nu = s[l]}{\langle \text{ref} \ e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle} \quad \text{[E-DEREF]} \\
  \frac{\langle e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle \quad \langle e_0, s \rangle \Downarrow \langle l_3, \nu_0' \rangle \quad s'' = s[v]}{\langle e_0, s \rangle \Downarrow \langle l_2, \nu_0 \rangle} \quad \text{[E-ASSIGN]}
  \end{align*}
  \]
Example again

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

Summary, so far

- Now have also seen simply typed $\lambda$-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 $\lambda$-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 essence of polymorphic types in our calculus
  - So we'll really understand it

Polymorphic $\lambda$-calculus

- Also known as **System F**
- Extend type syntax with a forall type
  
  $\epsilon ::= \ldots \mid \forall X. \epsilon \mid X$

- Now can write down the types of polymorphic values
  
  $id = \forall T. T \to T$
  
  $map = \forall a \cdot \forall b. (\forall a \cdot b) \mapsto (\forall a \cdot list) \mapsto (\forall b \cdot list)$
  
  $nil = \forall T. T \to list$

Values of polymorphic type

- Introduce explicit notation for values of polymorphic type and their instantiations
  - A polymorphic value: $L X. e$
    - $L X. e$ is a function that, given a type $\tau$, gives back $e$ with $\tau$ substituted for $X$
    - Use such values by instantiating them: $e[\tau]$
    - $e[\tau]$ is like function application
  - Syntax:
    
    $e ::= \ldots \mid L X. e \mid e[\tau]$
    
    $v ::= \ldots \mid L X. e$

An example

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

$id = \forall a. \forall x : a \mapsto a$

$id[int] 3 fi b

$3$

$id[bool] fi b$

$1 x:bool. x$
Another example

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

doTwice
   " 'a. (f: 'a->'a) -> 'a->'a "

   doTwice [int] succ 3 fi
      (1:int) -> int. f (f x) succ 3 fi
      succ (succ 3) fi

   (int * int)
   succ (succ 3) fi
   3

Yet another example

map " 'a. 'b. fix (1:map:('a->'b)->'a list: 'b list.
   f:('a->'b). 1: 'a list: 'a list.
   fold (case (unfold list) of
   <nil=> nil => <nil=> nil (nil => <nil=> nil)
   <cons=> cons => <cons=> 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] fi
   * [false,true,false]

   ML infers what the 'l, 'T and ['a] should be

A final example

(* fun cool f = (f 3, f true) *)

cool
   " 1:f:('a:['a->'a]). 1: ('a->'a) -> ('a->'a) -> ('a->'a)

   cool id
      (id: ('a->'a)) -> ('a->'a) -> ('a->'a) -> ('a->'a)
      id

   Note: L. inside of l and fi
   Can't write this in ML; not "prenex" form

Evaluation and typing rules

Evaluation:
  e \Downarrow (L X. e) i \Downarrow v

Typing:
  Γ, X: type ⊢ e : X [T-POLY]
  Γ ⊢ (L X. e) : X [T-INST]
  Γ ⊢ (e[i]) : [X[i]: c]

Different kinds of functions

- 1X. e is a function from values to values
- L X. e is a function from types to values
- What about functions from types to types?
  - Type constructors like fi, 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

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. fi, assoc_list)
- A "meta-type" is called a kind
Kinds

- A type describes a set of values or value constructors (a.k.a. functions) with a common structure
  \( t ::= \text{int} | t_1 \text{fi} t_2 | \ldots \)

- A kind describes a set of types or type constructors with a common structure
  \( k ::= \text{type} | k_1 \Rightarrow k_2 | \ldots \)

- Write \( t :: k \) to say that a type \( t \) has kind \( k \)

```
int :: type
int :: type
list :: type
assoc_list :: type
assoc_list string int :: type
```

Kinded polymorphic \( \lambda \)-calculus

- Also called System F
- Full syntax:
  \[ k ::= \text{type} | k_1 \Rightarrow k_2 \]
  \[ t ::= \text{int} | t_1 \text{fi} t_2 | X :: k \Rightarrow k_2 | \text{assoc_list} :: k \Rightarrow k_2 \]

- Functions and applications at both the value and the type level
- Arrows at both the type and kind level

Examples

```
pair "
  1':type. 1':type. \{first='a, second='b\} :: type

pair int bool "
  \{first=int, second=bool\} :: \{first=5, second=true\} : \text{pair int bool}

swap "
  \text{assoc_list} :: k \Rightarrow k 
  \text{swap} : \text{assoc_list string int} :: k 
```

Expression typing rules

```
\[ \frac{G \vdash t_1 :: \text{type}}{G, x : t_1 \vdash e :: t_2} \quad \text{T-ABS} \]
\[ \frac{G \vdash (LX::k. e) :: \text{type}}{G \vdash e :: \text{forall} k t} \quad \text{T-POLY} \]
\[ \frac{G \vdash e :: \text{forall} k t \quad G \vdash e_i :: t}{G \vdash e[\text{forall} k t] :: t} \quad \text{T-INST} \]
```

(T-VAR and T-APP unchanged)

Type kinding rules

```
\[ \frac{G \vdash t :: \text{type}}{G \vdash \text{int} :: \text{type} \quad \text{K-INT}} \]
\[ \frac{G \vdash \text{int} :: \text{type} \quad G \vdash t_1 :: \text{type}}{G \vdash t_1 \text{fi} t_2 :: \text{type} \quad \text{K-ARROW}} \]
\[ \frac{G, X::k :: \text{type}}{G \vdash \text{forall} k X :: \text{type} \quad \text{K-FORALL}} \]
\[ \frac{G, X::k :: \text{type}}{G \vdash X :: \text{forall} k \text{t} \quad \text{K-VAR}} \]
\[ \frac{G \vdash \text{forall} k X :: \text{type} \quad G \vdash t :: \text{type}}{G \vdash (\text{forall} k X :: \text{type}) :: \text{type} \quad \text{K-ABS}} \]
```

(T-APP and T-APP unchanged)

Summary

- Saw ever more powerful static type systems for the \( \lambda \)-calculus
  - Simply typed \( \lambda \)-calculus
  - Polymorphic \( \lambda \)-calculus, a.k.a. System F
  - Kinded poly. \( \lambda \)-calculus, a.k.a. System F
- 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
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