Review

- Type inference
  - For lambda calculus (variables, anonymous functions, applications)
  - Hindley-Milner (HM) inference algorithm:
    - Constraint generation
    - Constraint solving

- Today:
  - Finish constraint solving
  - Study lambda calculus itself

- Next week: on to mutability and object-oriented language principles

Lambda calculus

- The lambda calculus is a subset of ML that contains only three kinds of expressions:
  - Variables
  - Function application
  - Anonymous functions

  - No integers, no pattern matching, etc.
  - The “assembly language” of functional programming
    - Every ML program can be compiled down to just the lambda calculus
  - Doing type inference for all of ML is conceptually no different than doing it for the lambda calculus

Constraint generation

- Input: a lambda calculus expression e
  - Assume that every anonymous function in e has a different variable name as its argument
    - Easy to ensure that holds, thanks to lexical scope
  - Output: a set of constraints

Example:
- Input: `fn x => (fn y => x)`
- Output: `{T = R -> X, X = S -> Y, Y = R}`

Algorithm for constraint solving

- What is a solution to a set of constraints?
  - A way of substituting for variables to make equalities hold
    - e.g., given constraint
      - `X -> int -> int = int -> Y`
      - substituting `int -> int` for `Y` and `int` for `X` makes equality hold
    - So to solve a set of constraints, need an algorithm to compute a substitution that makes all the equalities hold
  - Well-known algorithm for this comes from AI: Robinson’s unification algorithm

Substitutions

- A type substitution (or simply substitution) is a map from a type-variable to a type
  - e.g., `Y = int -> int` is the substitution that maps `Y` to `int -> int`
- A substitution `S` can be applied to a type, producing a new type
  - `S(X) = if S = (X = t) then t else X`
  - `S(t1 -> t2) = S(t1) -> S(t2)`
- A substitution can be applied to a constraint, producing a new constraint
  - `S(t = t') = S(t) = S(t')`
- A substitution can be applied to a set of constraints, producing a new set of constraints; just apply `S` individually to each element of set
  - A more ML way of saying that: `S(C) = map(S, C)`
- Given two substitutions `S` and `S'`, write `S o S'` for their composition
  - `(S o S')(t) = S(S'(t))`
Toward real HM: Let-polymorphism

- One major piece of HM missing (actually, stems from language feature missing from lambda calculus) let expressions.

```plaintext
let
val double = fn f => fn x => f (f x)
val a = double (fn x => x+1) 1
val b = double (fn x => not x) false
in...
end
```

- As we’ve defined it so far, HM would infer type X->X for f in `double`
- Use of `double` in binding of a would produce constrains X = int
- Use of `double` in binding of b would produce constrains X = bool
- Unification would fail: program wouldn’t be typeable! 😊
- HM solves this by allowing each use of a polymorphic function have its own instantiation of the type variables.

Unifiers

- A substitution S unifies a constraint t1 - t2 if S(t1) = S(t2)
  - e.g., (Y = int -> int) o (X = int) unifies
    X -> X => int -> int => Y
- A substitution S unifies a set C of constraints if S unifies every constraint in
  C, in which case S is a unifier for C
- A solution to a set of constraints is a unifier for that set
  - If expression e generates set C of constraints, and if C has no unifier, then e is not typeable
  - The unification algorithm finds a unifier, if possible, for a set of constraints

Unification algorithm

- Fact: unification always terminates
- Fact: produces a result if and only if a unifier actually exists
  - i.e., if and only if set of constraints has some solution
- Fact: result produced is the most general unifier
  - if S = unify(C) and S’ unifies C, then there is some S’’ such that S’’ = S’ o S
  - So any other unifier makes more substitutions, which means it’s a less general type

HM type inference

(e.g., HM-like)

To infer type of e:
- Assign type variables to all variables and subexpressions in e
  - Type variable assigned to e is U(e)
- Generate constraints C from those preliminary types
- Solve constraints for substitution S
  - S(U(e)) is the inferred type of e

Fact: type inferred is “most lenient” (aka principal) type
  - any other type that e could have would be the same as some additional substitution applied to inferred type

Toward full SML: other features

- Most other language features actually not that complicated, except...
- Records
  - Requires a more complicated kind of constraint system
- Mutability
  - Some programs that might have been well-typed are instead rejected because there’s a chance that mutability could ruin strong typing
  - Called the value restriction
Toward real HM: Type checking + inference

- Actual HM algorithm does checking and inference simultaneously
  - "mashup" of the ideas we've now seen
  - In some ways, simpler!
    - Don't need the U and D functions
    - Don't have to introduce as many type variables
  - You'll implement it for homework 4

Complexity of HM

- HM is very efficient in practice (effectively linear in size of program)
  - But in theory...doubly exponential! \(2^{2^{\text{size of program}}})
- For fun, try this code out in REPL:

```plaintext
val h = true
val f0 = fn x => x+1
val f = fn x => if b then f0 else fn y => x y
val f = fn x => if b then f0 else fn y => x y
val f = fn x => if b then f0 else fn y => x y
(* keep repeating that line *)
```

Lambda calculus

- Helps you understand programming
  - The core of every functional language
- Helps you understand computation
  - Every "algorithmically computable function" can be expressed in lambda calculus (Church's Thesis)
- Helps you understand reasoning
  - Modern mathematical logic and automated reasoning based on ideas from lambda calculus
  - There is no guaranteed-to-terminate algorithm that can correctly determine whether a first-order logic formula is provable (Church's Theorem)

Alonzo Church

- Born in Washington, DC
- Turing's PhD advisor
- Invented lambda calculus

(1903-1995)