Announcements
- HW6 on Submitt
- Presentation guidelines up on Schedule page
  - 1. Select available paper/slot from list (first-come-first-serve)
  - 2. If available, I'll assign and update, otherwise goto 1
- Sorry, I’m a bit behind with paper selection
- No class on Monday

Outline
- Polymorphism
- Hindley Milner type inference. Algorithm W
- Monads

Motivating Example
- A sound type system rejects some programs that don’t get stuck
- Canonical example:
  
  \[
  \text{let } f = \lambda x . x \\
  \text{in} \\
  \text{if } (f \text{ true}) \text{ then } (f \text{ 1}) \text{ else 1}
  \]

  - Term does not get “stuck”
  - Term is NOT TYPABLE in System F₁. But it is typable in Hindley Milner!

Different Styles of (Parametric) Polymorphism
- Impredicative polymorphism (System F)
  \[
  \begin{align*}
  \tau &::= b \mid \tau_1 \to \tau_2 \mid T \mid \forall \tau.\tau \\
  E &::= x \mid \lambda x : \tau . E \mid E_1 E_2 \mid \Lambda T . E \mid E [\tau]
  \end{align*}
  \]

  - Can instantiate with polimorphic type!

  - Very powerful
    - Can type self application \( \lambda x . x \)
    - Still cannot type \( \text{fix} \)

  - Type inference is undecidable!

- Predicative polymorphism
  \[
  \begin{align*}
  \sigma &::= \tau \mid \forall \tau.\sigma \mid \sigma_1 \to \sigma_2 \\
  E &::= x \mid \lambda x : \sigma . E \mid E_1 E_2 \mid \Lambda T . E \mid E [\tau]
  \end{align*}
  \]

  - Still very powerful
    - Restricts System F by disallowing instantiation of
      with a polymorphic type: \( E [\tau] \) but not \( E [\sigma] \)

  - Type inference is still undecidable!
Different Styles of Polymorphism

- Prenex polymorphism
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall \tau. \sigma \]
  \[ E ::= x \mid \lambda \cdot \tau. E \mid E_1 E_2 \mid \Lambda T. E \mid E[\tau] \]

  - Now type inference is decidable
  - But polymorphism is limited
    - We cannot pass polymorphic functions
    - E.g., we cannot pass a sort function as argument

Towards Hindley Milner

Solution:

- Generalize the type variable in type of \( f \)
  \[ t_f : t_1 \rightarrow t_1 \] becomes \[ t_f : \forall \tau. T \rightarrow T \]

  - Different uses of generalized type variables are instantiated differently
    - E.g., (\( f \ true \)) instantiates \( t_f \) into \( bool \rightarrow bool \)
    - E.g., (\( f \ 1 \)) instantiates \( t_f \) into \( int \rightarrow int \)

  - When can we generalize?

Expression Syntax (to study Hindley Milner)

- Expressions:
  \[ E ::= c \mid x \mid \lambda x. E_1 \mid E_1 E_2 \mid \text{let} x = E_1 \text{ in } E_2 \]

  - There are no types in the syntax
  - The type of each sub-expression is derived by the Hindley Milner type inference algorithm
**Type Syntax**
(to study Hindley Milner)

- Types (aka monotypes):
  - \( \tau ::= b \mid \tau \rightarrow \tau \mid \tau \) is a type variable
  - E.g., \( \text{int, bool, int}\rightarrow\text{bool, int}\rightarrow\text{int, etc.} \)

- Type schemes (aka polymorphic types):
  - \( \sigma ::= \tau \mid \forall \tau. \sigma \)
  - E.g., \( \forall \tau. \tau\rightarrow\tau \rightarrow\tau \rightarrow\tau \)
  - Note: all quantifiers appear in the beginning, \( \tau \) cannot contain schemes

- Type environment now

Gamma ::= Identifiers \( \rightarrow \) Type schemes

**Instantiations**

- Type scheme \( \sigma = \forall \tau_1...\tau_n.\tau \) can be instantiated into a type \( \tau' \) by substituting types for the bound variables (BV) under the universal quantifier \( \forall \)
  - \( \tau' = S \tau \)
    - \( S \) is a substitution s.t. Domain(\( S \)) \( \supseteq \) BV(\( \sigma \))
  - \( \tau' \) is said to be an instance of \( \sigma (\sigma > \tau') \)
  - \( \tau' \) is said to be a generic instance when \( S \) maps some type variables to new type variables
  - E.g., \( \sigma = \forall \tau_1.\tau_1\rightarrow\tau_2 \)
    - \( [\tau_1/\tau_1] : t_1\rightarrow t_2 = t_1\rightarrow t_2 \) is a generic instance of \( \sigma \)
    - \( [\text{int/}\tau_1] : t_1\rightarrow t_2 = \text{int}\rightarrow t_2 \) is a non-generic instance of \( \sigma \)

**Generalization (aka Closing)**

- We can generalize a type \( \tau \) as follows
  - \( \text{Gen(} \Gamma, \tau \text{)} = \forall \tau_1...\tau_n.\tau \)
    - where \( (\tau_1...\tau_n) = \text{FV}(\tau) \rightarrow \text{FV}(\Gamma) \)
  - Generalization introduces polymorphism!
  - Quantify type variables that are free in \( \tau \) but are not free in the type environment \( \Gamma \)
    - E.g., \( \text{Gen}([],\tau_1\rightarrow\tau_2) \) yields \( \forall \tau_1.\tau_1\rightarrow\tau_2 \)
    - E.g., \( \text{Gen}([x:\tau_1],[x:\tau_2]) \) yields \( \forall \tau_1.\forall \tau_2.\tau_1\rightarrow\tau_2 \)

**Generalization, Examples**

- let \( f = \lambda x. x \) in if \((f \text{ true})\) then \((f \text{ 1})\) else \(1\)
  - We’ll infer type for \( \lambda x. x \) using simple type inference: \( \tau_1 \rightarrow \tau_1 \)
  - Then we’ll generalize that type, \( \text{Gen}(\Gamma,\tau_1\rightarrow\tau_1) \):
    - \( \forall \tau_1.\tau_1\rightarrow\tau_1 \)
  - Then we’ll pass the polymorphic type into \( \text{let} \: x = \text{E} \: \text{in} \: \text{E} \) and instantiate for each \( f \) in \( \text{if} (f \text{ true}) \) then \((f \text{ 1})\) else \(1\)
    - E.g., \( [u_2/\tau_1] (\tau_1\rightarrow\tau_1) \) where \( u_2 \) is fresh type variable at \((f 1)\)

**Generalization, Examples**

- \( \text{let} \: f = \lambda t.g \: \text{let} \: g = f \: \text{in} \: g \: x \)
  - \( \text{Gen}(\Gamma,\text{x}:\tau,\text{t}:\tau) \)
  - Why can’t we generalize \( \tau_1 \)?
  - Suppose we can generalize to \( \forall \tau_1 \)
    - Then \( \forall \tau_1 = \tau_1 \) will instantiate at \( \text{g} \: \text{x} \) to some fresh \( u \)
    - Then \( u \) becomes \( \tau_1 = \tau_1 \) thus losing the important connection between \( \tau_1 \) and \( t_1 \)
    - Thus \( \text{let} \: f = \lambda t.g \: \text{let} \: g = f \: \text{in} \: g \: x \) (\( \text{y.y+1} \)) true will type-check (unsound!!!)
  - DO NOT generalize variables that are mentioned in type environment \( \Gamma \)!
Hindley Milner Type Inference, Rough Sketch

let x = E₂ in E₂
1. Calculate type \( T_{E₂} \) for \( E₂ \) in \( \Gamma ; x: t_x \) using simple type inference
2. Generalize free type variables in \( T_{E₂} \) to get the type scheme for \( T_{E₂} \) (be mindful of caveat!)
3. Extend environment with \( x: \text{Gen}(f, T_{E₂}) \) and start typing \( E₂ \)
4. Every time we encounter \( x \) in \( E₂ \), instantiate its type scheme using fresh type variables

E.g., id’s type scheme is \( \forall t_1, t_2 \rightarrow t_1 \) so id is instantiated to \( u_k \rightarrow u_k \) at (id 1)

Hindley Milner Type Inference

- Two ways:
  - Extend Strategy 1 (constraint-based typing)
  - Extend Strategy 2 (Algorithm W)

Example

\( \lambda x. \text{let } f = \lambda y. x \text{ in } (f \text{ true, f 1}) \)

Strategy 1

let \( f = \lambda x. \text{ in if } (f \text{ true}) \text{ else } (f 1) \) end 1

1. let \( \Gamma = [f: t_f] \)
\( t_f = t_f \) now becomes \( t_f \rightarrow t_f \)
Next, generalize \( t_f \) \( \forall t_1, t_2 \rightarrow t_1 \)
\( t_1 = t_2 \) \( \forall t_1, t_2 \rightarrow t_1 \)

\( \lambda x : \Gamma \)
\( t_f \)
\( x \)
\( f \)
\( t_1 \)
\( t_2 \)
\( f \)
true
1

\( \text{Solve } (f) \) constraint \( t_f \rightarrow t_f \rightarrow t_f \)
\( t_1 = t_2 \) now becomes \( t_f \rightarrow t_f \)
Next, generalize \( t_f \) \( \forall t_1, t_2 \rightarrow t_1 \)
\( t_1 = t_2 \) \( \forall t_1, t_2 \rightarrow t_1 \)

\( \lambda x : \Gamma \)
\( t_f \)
\( x \)
\( f \)
true
1

Strategy 2: Algorithm W

\( \text{def } W(\Gamma, E) = \text{ case } E \text{ of } \)
\( \text{if continues from previous slide} \)
\( c \rightarrow (\text{SortOf}(c)) \)
\( x \rightarrow \text{if } (x \text{ NOT in Domain}(\Gamma)) \text{ then fail} \)
else let \( T_x = \Gamma(x) \)
in case \( T_x \) of
\( \forall \Gamma \rightarrow ([], [u_1, u_2, \ldots u_n] \Gamma) \)
\( \rightarrow ([], T_x, \Gamma) \)
\( \lambda x . E_x \rightarrow \text{let } T_{E_x} = \text{TypeOf}(x, T_x) E_x \)
in \( (S_{E_x}, T_x) \rightarrow T_{E_x} \)

\( \text{if } \ldots \)
\( \text{if continues on next slide!} \)

Strategy 2: Algorithm W

\( \text{def } W(\Gamma, E) = \text{ case } E \text{ of } \)
\( \text{if continues from previous slide} \)
\( c \rightarrow (\text{SortOf}(c)) \)
\( x \rightarrow \text{if } (x \text{ NOT in Domain}(\Gamma)) \text{ then fail} \)
else let \( T_x = \Gamma(x) \)
in case \( T_x \) of
\( \forall \Gamma \rightarrow ([], [u_1, u_2, \ldots u_n] \Gamma) \)
\( \rightarrow ([], T_x, \Gamma) \)
\( \lambda x . E_x \rightarrow \text{let } T_{E_x} = \text{TypeOf}(x, T_x) E_x \)
in \( (S_{E_x}, T_x) \rightarrow T_{E_x} \)

\( \text{if } \ldots \)
\( \text{if continues on next slide!} \)
**Strategy 2 Example**

Let $f = \lambda x. x$ in if (f true) then (f 1) else 1

1. let $f = \lambda x. x$ in if (f true) then (f 1) else 1

2. Abs

3. if-then-else

4. App

5. App

No constraint, types Abs immediately: $T_f = \langle t \rangle \rightarrow \langle t \rangle$, $S_f = \emptyset$

Hindley Milner Observations

- Do not generalize over type variables mentioned in type environment (they are used elsewhere)
- `let` is the only way of defining polymorphic constructs
- Generalize the types of `let`-bound identifiers only after processing their definitions

Hindley Milner Limitations

- Quiz example:

$(\lambda x. x (\lambda y. y) (x \ 1)) (\lambda z. z)$

vs.

let $x = (\lambda z. z)$ in $x (\lambda y. y) (x \ 1)$

Example

$\lambda x. let f = \lambda y. x in (f true, f 1)$

Spring 19 CSCI 4450/6450, A Milanova
Monads

A way to cleanly compose computations
- E.g., \( f \) may return a value of type \( a \) or Nothing
- Composing computations becomes tedious:
  - case \( (f \, s) \) of
    - Nothing \( \rightarrow \) Nothing
    - Just \( m \) \( \rightarrow \) case (\( f \, m \)) ...

In Haskell, monads cleanly encapsulate IO and other imperative features

An Example: Cloned Sheep

type Sheep = ...
father :: Sheep \( \rightarrow \) Maybe Sheep
father = ...
mother :: Sheep \( \rightarrow \) Maybe Sheep
mother = ...
(Note: a sheep may have both parents, or just one)
maternalGrandfather :: Sheep \( \rightarrow \) Maybe Sheep
maternalGrandfather \( s \) = case (mother \( s \)) of
  - Nothing \( \rightarrow \) Nothing
  - Just \( m \) \( \rightarrow \) father \( m \)

An Example

mothersPaternalGrandfather :: Sheep \( \rightarrow \) Maybe Sheep
mothersPaternalGrandfather \( s \) = case (mother \( s \)) of
  - Nothing \( \rightarrow \) Nothing
  - Just \( m \) \( \rightarrow \) case (father \( m \)) of
    - Nothing \( \rightarrow \) Nothing
    - Just \( gf \) \( \rightarrow \) father \( gf \)

- Tedious, unreadable, difficult to maintain
- Monads help!

The Monad Type Class

- Haskell’s Monad class requires 2 operations, \( \gg\gg \) (bind) and \texttt{return}

\begin{verbatim}
class Monad m where
  // (the bind operation) takes a monad
  // it into a monad \( m \, b \), and returns \( m \, b \)
  (\( \gg\gg \)) :: m a \( \rightarrow \) (a \( \rightarrow \) m b) \( \rightarrow \) m b
  // return encapsulates a value into the monad
  return :: a \( \rightarrow \) m a
\end{verbatim}
The List Monad

- The List type constructor is a monad
  - `lis >>= f = concat (map f lis)
  
  - `return x = [x]
  
  Note: `concat :: [[a]] -> [a]
  
  e.g., `concat [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]

- Use any f s.t. `f :: a -> [b]. f may return a list of 0,1,2,... elements of type b, e.g.,
  - `f x = [x+1]
  - `[1,2,3] >>= f // yields ?

---

do Notation (Syntactic Sugar)

- `f x = x+1
  
  - `g x = x*5
  
  - `do { x <- [1,2,3]; y <- (return . f) x; (return . g) y }
  
  is syntactic sugar for
  
  - `[1,2,3] >>= (return . f) >>= (return . g)

---

List Comprehensions

- `[ 2*i | i <- [1..] ] yields ?
  
  - `[ (i,j) | i <- [1,2], j <- [1..4] ] yields ?

- `[ (i,j) | i <- [1,2], j <- [1..4] ] is syntactic sugar for
  
  - `do { i <- [1,2]; j <- [1..4]; return (i,j) }

  which in turn is syntactic sugar for?

  - `[1,2] >>= (lx -> [1..4] >>= (y -> return (x,y)))

---

Monad Laws

1. `(return x) >>= f <<= f x

2. `m >>= return <<= m

3. `m >>= f >>= g <<= m >>= (lx -> f x >>= g)

- Adherence to monad laws is a responsibility of the programmer who wrote the Monad instance

- Ensure correctness of do notation!
So What is the Point of the Monad…

- Conveniently chains computation
- Encapsulates “mutable” state. E.g., IO:
  - `openFile :: FilePath -> IOMode -> Handle`
  - `hClose :: Handle -> () -- void`
  - `hIsEOF :: Handle -> Bool`
  - `hGetChar :: Handle -> Char`

These operations break “referentially transparency”. For example, `hGetChar` returns different value when called twice in a row.

The IO Monad

- IO a: Computation that does some IO producing a value of type a. E.g., (IO Char), (IO String)
- Unlike other monads (e.g., Maybe) there is no way to make IO a into an a
- The monad encapsulates “mutable” IO state
- … and, there is no “rep exposure” of this state!
- Access to state is only through well-defined monadic operations (e.g., `hGetChar`)

The IO Monad

- `getFileContents :: String -> IO String`
  - `getFileContents filename = do
     h <- openFile filename
     putStrLn filename
     reversed_cs <- readFileContents h []
     hClose h
     return (reverse reversed_cs)`
- `readFileContents :: Handle -> String -> IO String`
  - `readFileContents h rcs = do
     b <- hIsEOF
     if (b) then return rcs`
  - `else do { c <- hGetChar h; readFileContents h (c:rcs) }`

Other useful functions

- // reads entire file into one string:
  - `readFile :: FilePath -> IO String`
- // writes entire string into a file:
  - `writeFile :: FilePath -> String -> IO ()`
- E.g. `main = do`
  - `[f,g] <- getArgs`
  - `s   <- readFile f`
  - `writeFile g s`