Hindley Milner Type Inference, Haskell Type Classes, Monads

Announcements
- HW6 not up on Submitty yet
- Presentation guidelines and papers 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.

Outline
- Parametric polymorphism
- Hindley Milner type inference. Algorithm W
- Haskell type classes
- Monads (for imperative features)

Motivating Example
- A sound type system rejects some programs that don’t get stuck
- Canonical example:
  ```
  let f = \x.x
  in
  if (f true) then (f 1) else 1
  ```
- Term does not get “stuck”
- Term is NOT TYPABLE in the simply typed lambda calculus. But it is typable in Hindley Milner!

Different Styles of (Parametric) Polymorphism
- Impredicative polymorphism (System F)
  ```
  \( \tau ::= b | \tau_1 \rightarrow \tau_2 | T | \forall \tau. \tau \)  
  \( E ::= x | \lambda x : \tau.E | E_1 E_2 | \Lambda T.E | E \[ \tau \] \)
  ```
- Very powerful
  - Can type \( \lambda x. x \)
  - Still cannot type `fix`
- Type inference is undecidable!

Different Styles of Polymorphism
- Predicative polymorphism
  ```
  \( \tau ::= b | \tau_1 \rightarrow \tau_2 | T \)
  \( \sigma ::= \tau | \forall \tau. \sigma | \sigma_1 \rightarrow \sigma_2 \)
  \( E ::= x | \lambda x : \sigma.E | E_1 E_2 | \Lambda T.E | E \[ \tau \] \)
  ```
- Still very powerful
  - But cannot instantiate with a polymorphic type
- Type inference is still undecidable!
Different Styles of Polymorphism

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

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

Towards Hindley Milner

- Let f = \lambda x. x
  - Constraints
    \[ t_f : \tau \rightarrow \tau_1 \]
    \[ t_f = \text{bool} \rightarrow \tau_1 \]
    \[ t_f = \text{int} \rightarrow \tau_3 \]

- Doesn’t unify!

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):
  - \( \tau ::= b | t_1 \rightarrow t_2 \) (\( t \) is a type variable)
  - E.g., int, bool, Int->bool, t₁->int, t₁->t₁, etc.
- Type schemes (aka polymorphic types):
  - \( \sigma ::= \tau | \forall \tau. \sigma \) \( t \) is a "free" type variable as it isn't bound under \( \forall \)
  - Note: all quantifiers appear in the beginning, \( \tau \) cannot contain schemes
- Type environment now

\( \Gamma ::= \text{Identifiers} \rightarrow \text{Type schemes} \)

Instanciations

- Type scheme \( \sigma = \forall \tau_1 \ldots \tau_n. \tau \) 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. \( \text{Domain}(S) \supseteq \text{BV}(\sigma) \)
  - \( \tau' \) 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., \( \sigma = \forall \tau_1. \tau_1 \rightarrow \tau_2 \)
- \( [\tau_1/t_1] \tau_1 \rightarrow \tau_2 = \tau_1 \rightarrow \tau_2 \) is a generic instance of \( \sigma \)
- \( [\text{int}/t_1] \tau_1 \rightarrow \tau_2 = \text{int} \rightarrow \tau_2 \) is a non-generic instance of \( \sigma \)

Generalization (aka Closing)

- We can generalize a type \( \tau \) as follows
  \[ \text{Gen}(\Gamma, \tau) = \forall t_1, \ldots, t_n, \tau' \]
  where \( (t_1, \ldots, t_n) = \text{FV}(\tau) = \text{FV}(\Gamma) \)
- Generalization introduces polymorphism
- Quantify type variables that are free in \( \tau \) but are not free in the type environment \( \Gamma \)
  - E.g., \( \text{Gen}([\lambda], t_1 \rightarrow t_2) \) yields \( \forall t_1, t_1 \rightarrow t_2 \)
  - E.g., \( \text{Gen}([\lambda x], t_1 \rightarrow t_2) \) yields \( \forall t_1, t_1 \rightarrow t_2 \)

Generalization, Examples

\[ \text{Gen}(\Gamma, \tau) = \forall t_1, \ldots, t_n, \tau' \]

\[ \text{Gen}(\Gamma, \lambda x. x) = \forall t_1, \lambda x : t_1. t_1 \]

\[ \text{Gen}(\Gamma, \text{if } (f \text{ true}) \text{ then } (f \text{ true} ) \text{ else } 1) = \forall t_1, \text{if } (f \text{ true}) \text{ then } (f \text{ true} ) \text{ else } 1 \]

\[ \text{Gen}(\Gamma, \text{Let } x = E_1 \text{ in } E_2) = \forall t_1, \text{Let } x = E_1 \text{ in } E_2 \]

\[ \text{Gen}(\Gamma, \text{let } x = E_1 \text{ in } E_2) = \forall t_1, \text{let } x = E_1 \text{ in } E_2 \]

Hindley Milner Typing Rules

\[ \Gamma \vdash \tau : \tau' \quad \Gamma, \tau : \text{Gen}(\Gamma, \tau) \vdash \tau : \tau' \quad (\text{Let}) \]

\[ \Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau' \quad (\text{Var}) \]

- Type of \( x \) as inferred for \( E_1 \) is \( \tau \). Type of \( x \) in \( E_2 \) is the generalized type scheme \( \sigma = \text{Gen}(\Gamma, \tau) \)
- \( x \in \text{FV}(E_2) \) and \( x \) has type \( \tau \) in the environment can be instantiated to \( \tau \)

(Note: remaining rules, c, App, Abs are as in \( F_1 \) )
Next, generalize 

Solve (!)

\[ \lambda = \ldots \]

1. Calculate type \( T_{E_1} \) for \( E_1 \) in \( \Gamma;x:t \) using simple type inference
2. Generalize free type variables in \( T_{E_1} \) to get the type scheme for \( T_{E_1} \) (be mindful of caveat!)
3. Extend environment with \( x:Gen(T_{E_1}) \) and start typing \( E_2 \)
4. Every time we encounter \( x \) in \( E_2 \), instantiate its type scheme using fresh type variables

E.g., \( id \)'s type scheme is \( \forall x.t \rightarrow t \rightarrow t \rightarrow t \); so \( id \) is instantiated to \( u_k \rightarrow u_k \) at \( id 1 \)

---

**Hindley Milner Type Inference, Rough Sketch**

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Calculate type ( T_{E_1} ) for ( E_1 ) in ( \Gamma;x:t ).</td>
</tr>
<tr>
<td>2.</td>
<td>Generalize free type variables in ( T_{E_1} ) to get the type scheme for ( T_{E_1} ).</td>
</tr>
<tr>
<td>3.</td>
<td>Extend environment with ( x:Gen(T_{E_1}) ) and start typing ( E_2 ).</td>
</tr>
<tr>
<td>4.</td>
<td>Every time we encounter ( x ) in ( E_2 ), instantiate its type scheme using fresh type variables.</td>
</tr>
</tbody>
</table>

---

**Strategy 1**

**Example**

\[ \lambda. \; let \; f = \lambda y. \; x \; in \; (f \; true, \; f \; 1) \]

---

**Strategy 2: Algorithm W**

```
def W(\( \Gamma \), \( E \)) = case \( E \) of
  c  -\> if (?\( \).c) then fail
  x  -\> if \( x \) NOT in Domain(\( \Gamma \)) then fail
          else let \( T_x = \Gamma(\( x \)) \)
          in case \( T_x \) of
                   \[ \forall t \rightarrow t \rightarrow t \rightarrow t \] -> \( ([u_1,u_2,u_3,u_4] \rightarrow \) \( t \) \)
                   \( \ldots \) -> \( (\ldots) \rightarrow \) \( t \) \)
          \( \lambda x.\; E_x \) -> let \( (S_{E_x}, T_{E_x}) = W(\( \Gamma \), \( x \)) \)
          in \( (S_{E_x}, S_{\lambda x}(t_3) \rightarrow T_{E_x}) \)

          \ldots
          \ldots
          \ldots
          \ldots
```

---

**Strategy 2: Algorithm W**

```
def W(\( \Gamma \), \( E \)) = case \( E \) of
  if continues from previous slide
  // ...
  E_1, E_2 -> let \( (S_{E_1}, T_{E_1}) = W(\( \Gamma \), \( E_1 \)) \)
           \( (S_{E_2}, T_{E_2}) = W(\( \Gamma \), \( E_2 \)) \)
           \( S = \text{Unify}(S_{E_1} (T_{E_2} \rightarrow T_{E_2}) = T_{E_2} \rightarrow T_{E_2}) \)
           in \( (S \in S_{E_2}, S(T)) \)
  let x = \( E_x \) in \( E_x \rightarrow \) let \( (S_{E_x}, T_{E_x}) = W(\( \Gamma \), \( x \)) \)
           in \( (S_{E_x}, S_{\lambda x}(t_3) \rightarrow T_{E_x}) \) ;
           \( \ldots \) \( \ldots \) \( \ldots \) \( \ldots \) \( \ldots \)
```

---

Spring 18 CSCI 4450/6450, A Milanova (from MIT 2015 Program Analysis OCW)
**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**

- Only let-bound constructs can be polymorphic and instantiated differently
- Quiz example:
  \[
  (\lambda x. x (\lambda y. y) (x 1)) (\lambda z. z)
  \]
  vs.
  \[
  \text{let } x = (\lambda z. z) \\
  \text{in } x (\lambda y. y) (x 1)
  \]
Generic Functions in Haskell

We can generalize a function when a function makes no assumptions about the type:

```haskell```
const :: a -> b -> a
const x y = x
```

```haskell```
apply :: (a -> b) -> a -> b
apply g x = g x
```

Generic Functions

We can generalize a function when a function makes no assumptions about the type:

```haskell```
const :: a -> b -> a
const x y = x
```

```haskell```
apply :: (a -> b) -> a -> b
apply g x = g x
```

Generic Functions

We can generalize a function when a function makes no assumptions about the type:

```haskell```
const :: a -> b -> a
const x y = x
```

```haskell```
apply :: (a -> b) -> a -> b
apply g x = g x
```

Generic Functions

Can we have sum of parameterized type?

```haskell```
sum :: a -> List a -> a
sum n Nil = n
sum n (Cons x xs) = sum (n + x) xs
```

No. a no longer unconstraint. Type and function definition imply that + is of type a -> a -> a but + is not defined for all types!

Furthermore, + is different on Ints and Strings!

Haskell Type Classes

Define a type class containing the arithmetic operators:

```haskell```
class Num a where
  (==) :: a -> a -> Bool
  (+) :: a -> a -> a
  ...
instance Num Int where
  ...
instance Num Float where
  ...
```

On view of type classes: predicates

(Num a) is a predicate in type definitions

Constraints the types we can instantiate a generic function with appropriate types

A type class has associated laws
Haskell Type Classes

- “forall” generic functions (e.g., \( \text{apply} :: (a \rightarrow b) \rightarrow a \rightarrow b \)) embed parametric polymorphism
- Type classes are introduced to control ad-hoc polymorphism (as known as overloading)
- More differences than similarities with standard object-oriented notion of class

Outline

- Parametric polymorphism
- Hindley Milner type inference
- Haskell type classes
- Monads (for imperative features)

The Monad Type Class

- Haskell’s Monad class requires 2 operations, \( \triangleright= \) (bind) and return

```haskell
class Monad m where
  // \( \triangleright= \) (the bind operation) takes a monad \( m \ a \), and a function that takes \( a \) and turns it into a monad \( m \ b \)
  // \( \triangleright= \) :: \( m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b \)
  // return encapsulates a value into the monad
  return :: a \rightarrow m a
```

The Maybe Monad

```haskell
instance Monad Maybe where
  Nothing \( \triangleright= \) f = Nothing
  (Just x) \( \triangleright= \) f = f x
  return = Just
```

- Cloned Sheep example (from awhile ago):
  mothersPaternalGrandfather s =
  (return s) \( \triangleright= \) mother \( \triangleright= \) father \( \triangleright= \) father

(The Note: if at any point, some function returns Nothing, Nothing gets cleanly propagated.)

The List Monad

- The List type constructor is a monad
  \( li \triangleright= f = \text{concat} \ (\text{map } f \ li) \)
  return \( x = [x] \)

Note: \( \text{concat} :: [\left< a \right>] \rightarrow \left< a \right> \)
  e.g., \( \text{concat} \ [\left< 1,2,3,4,5,6 \right>] \) yields \( [1,2,3,4,5,6] \)

- Use any \( f \) s.t. \( f :: a \rightarrow \left< b \right> \). \( f \) may return a list of 0,1,2,…elements of type \( b \), e.g.,
  - \( f x = [x+1] \)
  - \( [1,2,3] \triangleright= f \) // returns \( [2,3,4] \)

The do Notation (Syntactic Sugar)

```haskell
> f x = x + 1
> g x = x^5
> [1,2,3] \triangleright= (return . f) \triangleright= (return . g)
```

Or

```haskell
> do { x <- [1,2,3]; y <- (return . f) x; (return . g) y }
```

(Note: obeying the Monad laws ensures do-notation behaves consistently with \( \triangleright= \))
So What is the Point of the Monad…

- Conveniently chains computation
- Encapsulates "mutable" state. E.g., IO:
  openFile :: FilePath -> IOMode -> String
  hClose :: Handle -> () -- void
  hasEOF :: Handle -> Bool
  hGetChar :: Handle -> Char

[These operations break "referential transparency". For example, hGetChar typically returns different value when called twice in a row.]

Monadic IO

- IO a: Computation that does some IO producing a value of type a
- Unlike other monads (e.g., Maybe) there is no way to make IO a into an a
- The monad encapsulates "mutable" IO state
  - But there is no "rep exposure" of this state!
  - Access to state is only through well-defined monadic operations (e.g., hGetChar)
- Embedding facilitates well-defined sequence of monadic operations

Monadic IO

openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO () -- void
hasEOF :: Handle -> IO Bool
hGetChar :: Handle -> IO Char

getFileContents :: String -> IO String
getFileContents filename = do
  h <- openFile filename ReadMode
  putStrLn filename
  reversed_cs <- readFileContents h []
  hClose h
  return (reverse reversed_cs)

readFileContents :: Handle -> String -> IO String
readFileContents h rcs = do
  b <- hasEOF h
  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
  writeFile g s

Next Class

- Subtyping
- Pluggable types
- Type-based taint analysis for Android