Hindley Milner, conclusion; Haskell

Announcements

- Quiz 6
- HW6?
- Please sign up for papers

Outline

- Hindley Milner type inference, conclusion
  - Observations and examples

- Haskell: records, type classes and monads

Hindley Milner Observations

- Generates the most general type (principal type) for each term/subterm
- Type system is sound

- Complexity of Algorithm W
  - PSPACE-Hard
  - Because of nested let blocks

Hindley Milner Recap

let x = E_1 in E_2

1. Calculate type **T_{E1}** for **E_1** in **$\Gamma;x:t_x$** using simple type inference. **T_{E1}** is **principal type** of **E_1**
2. Generalize free type variables in **T_{E1}** to get type scheme for **T_{E1}** (be mindful of caveat!)
3. Extend environment with **$x:Gen(\Gamma,T_{E1})$** and start typing **E_2**
4. When we encounter **x** in **E_2**, instantiate its type scheme to a fresh monotype

   E.g., **id**'s type scheme is **$\forall t_1.t_1 \to t_1$** so **id** is instantiated to **$u_k \to u_k$** at **(id 1)**

Hindley Milner Limitations

- Only let-bound constructs can be polymorphic and instantiated differently

$t_{twice} = (t_x \to t_x) \to t_x \to t_x$

$Gen([\,], (t_x \to t_x) \to t_x \to t_x) = \underline{\forall t_x . (t_x \to t_x) \to t_x \to t_x}$

$twice = \lambda f x . f(f x)$

let twice f x = f (f x)

in twice twice succ 4 // let-bound polymorphism

$(u_1 \to u_1) \to (u_1 \to u_1) \quad (u_2 \to u_2) \to u_2 \to u_2 \qquad [u_2 \to u_2 / u_1]$

$((u_2 \to u_2) \to (u_2 \to u_2)) \to ((u_2 \to u_2) \to (u_2 \to u_2)) \qquad (u_2 \to u_2) \to u_2 \to u_2$

$(u_2 \to u_2) \to u_2 \to u_2$ // type of twice twice

$(u_2 \to u_2) \to u_2 \to u_2 \qquad int \to int \qquad [int / u_2]$

$int \to int$ // twice twice succ

Hindley Milner Limitations

- Only let-bound constructs can be polymorphic and instantiated differently

```
let twice f x = f (f x)
    foo g = g g succ 4 // lambda-bound
in foo twice
```

$t_g = t_g \to t$

Hindley Milner Limitations

- Another example:

(λx. x (λy. y) (x 1)) (λz. z)

vs.

```
let x = (λz. z)
in
   x (λy. y) (x 1)
```

Outline

- Hindley Milner type inference, conclusion
  - Observations and examples
- Haskell: records, type classes and monads

Haskell Records

```
{- Constraint environment. -}
type Constraints = [(Type, Type)]
data ConstraintEnv = CEnv
      {
      constraints :: Constraints
      , var :: Int
      , tenv :: TEnv
      }
cenv = CEnv { constraints=[], var=0, tenv=[] } ;; new environment
… constraints cenv … var cenv … tenv cenv … ;; field accessors
```

the new type

constructor

inferTypes cenv (ELambda v body)

let

(cenv', tv) = ~~fresh~~ new TVar cenv

(cenv'', te) = newTvar cenv'

cenv''' ~~[illegible]~~ augment cenv'' with v:tv

(cenv'''', tbody) = inferTypes cenv''' body

cenv^v ← add constraint te = tv → tbody

in (cenv^v, te)

Haskell Type Classes

- Not to be confused with Java classes/interfaces
- Let us define a type class containing the arithmetic and comparison operators:

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

Read: A type **a** is an instance of the type class **Num** if it provides “overloaded” definitions of operators **==**, **+**, …

Read: **Int** and **Float** are instances of **Num**

Generic Functions with Type Class

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

- One view of type classes: predicates
  - **(Num a)** is a predicate in type definitions
  - Constrains the specific types we can instantiate a generic function with
- A type class has associated laws

Type Class Hierarchy

```
class Eq a where
   (==), (/=)  :: a -> a -> Bool

class (Eq a) => Ord a where
   (<), (<=), (>), (>=)  :: a -> a -> Bool
   min, max              :: a -> a -> a
```

- Each type class corresponds to one concept
- Class constraints give rise to a hierarchy
- **Eq** is a superclass of **Ord**
  - **Ord** inherits specification of **(==)** and **(/=)**
  - Notion of “true subtyping”

Monad Quote

- “A monad is just a monoid in the category of endofunctors, what's the problem?”

- Monad type class and the monad laws
- Maybe monad
- List monad
- IO monad
- State monad

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 → Nothing
    Just m → case (f m) …

(Just expr) >>= oneStep >>=
oneStep >>= ...

- In Haskell, monads model IO and other **imperative** features

An Example: Cloned Sheep

```
type Sheep = …
father :: Sheep → Maybe Sheep
father = ...
mother :: Sheep → Maybe Sheep
mother = …
```

(Note: a sheep has both parents; a cloned sheep has one)

```
maternalGrandfather :: Sheep → Maybe Sheep
maternalGrandfather s = case (mother s) of
                          Nothing → Nothing
                          Just m → father m
```

An Example

```
mothersPaternalGrandfather :: Sheep → Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
                               Nothing → Nothing
                               Just m → case (father m) of
                                      Nothing → Nothing
                                      Just gf → father gf
```

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

The Monad Type Class

- Haskell's Monad type class requires 2 operations, >>= (bind) and return

class Monad **m** where

// >>= (the bind operation) takes a monad
// **m a**, and a function that takes **a** and turns
// it into a monad **m b**, and returns **m b**

(>>=) :: **m a** → (**a** → **m b**) → **m b**

// return encapsulates a value into the monad

return :: **a** → **m a**

The **Maybe** Monad

```
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

- Back to our example:

```
mothersPaternalGrandfather s =
      (return s) >>= mother >>= father >>= father
```

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

The **List** Monad

- The List type constructor is a monad

li **>>=** f = concat (map f li)

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** --- ?

The **List** Monad

```
parents :: Sheep → [Sheep]
parents s = MaybeToList (mother s) ++
                                   MaybeToList (father s)

grandParents :: Sheep → [Sheep]
grandParents s = (parents s) >>= parents
```

The **do** Notation

- **do** notation is syntactic sugar for monadic bind

```
> f x = x+1
> g x = x*5
> [1,2,3] >>= (return . f) >>= (return . g)
```

Or

```
> [1,2,3] >>= \x->[x+1] >>= \y->[y*5]
```

Or, make encapsulated element explicit with **do**

```
> do { v <- [1,2,3]; w <- (\x->[x+1]) v; (\y->[y*5]) w }
```

List Comprehensions

```
> [ x | x <- [1,2,3,4] ]
[1,2,3,4]
> [ x | x <- [1,2,3,4], x `mod` 2 == 0 ]
[2,4]
> [ [x,y] | x <- [1,2,3], y <- [6,5,4] ]
[[1,6],[1,5],[1,4],[2,6],[2,5],[2,4],[3,6],[3,5],[3,4]]
```

List Comprehensions

- List comprehensions are syntactic sugar on top of the **do** notation!

[x | x <- [1,2,3,4]] is syntactic sugar for

do { x <- [1,2,3,4]; return x }

[[x,y] | x <- [1,2,3], y <- [6,5,4]] is syntactic sugar for

do { x <- [1,2,3]; y<-[6,5,4]; return [x,y] }

- Which in turn, we can translate into monadic bind…

So, What is the Point of the Monad…

- Conveniently chains (builds) computation

- Encapsulates “mutable” state. E.g., **IO**:

openFile :: FilePath -> IOMode -> IO Handle

hClose :: Handle -> IO () -- void

hIsEOF :: Handle -> IO Bool

hGetChar :: Handle -> IO Char

These operations break “referential transparency”.
For example, **hGetChar** typically returns different value when called twice in a row!