Lecture 22: Polymorphism

CS4400 Programming Languages

Limitations of monomorphic type systems:

- What we've seen so far are *monomorphic* types, meaning a type only represents one type
- What is the problem?
  - identity
  - operations on lists
  - operations on pairs
  - ...
- Consider length of a list – we need:

  ```
  length_int :: [Integer] -> Integer
  length_bool :: [Bool] -> Integer
  ```

Are we done?

What about length of lists of functions from integers to integers?

Length of lists of functions from integers to booleans?

Length of lists of functions from (function from integers to booleans) to integers?

Etc.

- Even simpler: identity

  ```
  id_int :: Integer -> Integer
  id_bool :: Bool -> Bool
  id_intToBool :: (Integer -> Bool) -> (Integer -> Bool)
  ```

- Functions which perform operations such as counting the number of elements of a list, or swapping the elements of a pair, do not depend on the type of the elements in the list or pair

What do we need?

- Back to an untyped language? :-(
- We would really like to specify a type of functions that work for lists (or pairs) containing any type
- In other words, we need a way to say "*for any type t*, list of t"
- In yet other words,

```
length :: forall a. [a] -> Integer
id :: forall a. a -> a
```

- Ingredients:
  a) *Type* variables
  b) Abstracting type variables (quantifying)
- In Haskell (or ML, OCaml, ...), polymorphic types are inferred for you and you (usually) do not need to say that you want a polymorphic function
- Another implementation of the same idea are Java generics

Basics:

- Additional syntax for types

```
type TyVariable = String

data Type = ...
          | TyVar TyVariable
          | TyForall TyVariable Type
```

- For `TyForAll`, we can use a more economical alternative:

```
data Type = ...
          | TyVar TyVariable
          | TyForall [TyVariable] Type
```

We are now able to abstract the type of a function. But how do we actually give the information to the type-abstracted function?

Idea: we pass what type we actually intend to use at runtime.

Consequence: We need type abstraction and type application on the expression level

New syntax:

```haskell
data Expr = ...
          | TAbs TyVariable Expr
          | TApp Expr Type
```

We obtain: polymorphic lambda calculus (with extensions) aka System F.

How do we perform application? Substitute the types

Typing rules:

```
 tenv |- e : t
---------------------------------
 tenv |- TAbs a e : TyForall a t

 tenv |- e : TyForall a t'
----------------------------------
 tenv |- TApp e t : tsubst a t t'
```

Here we use type substitution to substitute types (careful about Forall!)

```haskell
typeOf tenv (TAbs a e) =
  do t <- typeOf tenv e
     return (TyForall a t)
typeOf tenv (TApp e t) =
  do TyForall a t' <- typeOf tenv e
     tsubst a t t'
typeOf tenv (Add e1 e2) =     -- previous cases need to be refactored to use tenv
  do TyInt <- typeOf tenv e1
     TyInt <- typeOf tenv e2
     return TyInt
typeOf tenv (Num _) = return TyInt

tsubst :: TyVariable -> Type -> Type -> Maybe Type
tsubst a s (TyVar b) | a == b    = return s
                     | otherwise = return (TyVar b)
tsubst a s (TyForall b t)
  | a == b = return $ TyForall b t
  | freeInType a s = Nothing
  | otherwise = TyForall b <$> tsubst a s t
tsubst a s (TyArrow t1 t2) =
  do t1' <- tsubst a s t1
     t2' <- tsubst a s t2
     return (TyArrow t1' t2')
```

How do we evaluate type abstraction and type application? We can either use substitution or add another environment

Other kinds of polymorphism

- What we talked about is *parametric polymorphism* – mention let polymorphism
- Other type of polymorphism: *ad-hoc*
  - Allows a polymorphic value to exhibit different behaviors, depending on the actual type
  - *Overloading*: associates a single function symbol with many implementations
  - Compiler (or the runtime system) chooses an appropriate implementation for each application of the function – based on the types of the arguments