Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS) Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

January 12, 2011

Today's Topics

- A Module for Core FP Expressions
- Implementing Unification
- Implementing Type Inference

A Module for Core FP Expressions

Grammar of Core FP

$$
\begin{array}{rcll}
e & \stackrel{\text{def}}{=} & x \mid e\ e \mid \lambda x.\, e & \lambda\text{-calculus} \\
 & \mid & c & \text{constant (for primitives)} \\
 & \mid & \textbf{let}\ x = e\ \textbf{in}\ e & \text{let binding} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \text{conditional}
\end{array}
$$

Grammar of Core FP

$$
\begin{array}{rcll}
e & \stackrel{\text{def}}{=} & x \mid e\ e \mid \lambda x.\, e & \lambda\text{-calculus} \\
 & \mid & c & \text{constant (for primitives)} \\
 & \mid & \textbf{let}\ x = e\ \textbf{in}\ e & \text{let binding} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \text{conditional}
\end{array}
$$

A Type for Core FP Expressions

```
type Id   =  String
data Exp  =
  Var Id | Con Id |
  App Exp Exp     |
  Abs Id Exp      |
  Let Id Exp Exp  |
  Ite Exp Exp Exp   deriving Eq
```

Grammar of Core FP

$$
\begin{array}{rcll}
e & \stackrel{\text{def}}{=} & x \mid e\ e \mid \lambda x.\, e & \lambda\text{-calculus} \\
 & \mid & c & \text{constant (for primitives)} \\
 & \mid & \textbf{let}\ x = e\ \textbf{in}\ e & \text{let binding} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \text{conditional}
\end{array}
$$

A Type for Core FP Expressions

```
type Id   =  String
data Exp  =
  Var Id | Con Id |
  App Exp Exp     |
  Abs Id Exp      |
  Let Id Exp Exp  |
  Ite Exp Exp Exp   deriving Eq
```

Core FP Expressions from Strings

```
fromString :: String -> Exp
```

Grammar of Types

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \to \tau \mid C(\tau, \ldots, \tau)$$

Grammar of Types

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \to \tau \mid C(\tau, \ldots, \tau)$$

A Type for Types

```
data Type = TVar Int
          | TFun Type Type
          | TCon Id [Type]  deriving Eq
```

Implementing Unification

Recall

$$\frac{E_1; C(\tau_1, \ldots, \tau_n) \approx C(\tau_1', \ldots, \tau_n'); E_2}{E_1; \tau_1 \approx \tau_1'; \ldots; \tau_n \approx \tau_n'; E_2} \ (\mathsf{d}_1)$$

$$\frac{E_1; \tau_1 \to \tau_2 \approx \tau_1' \to \tau_2'; E_2}{E_1; \tau_1 \approx \tau_1'; \tau_2 \approx \tau_2'; E_2} \ (\mathsf{d}_2)$$

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{TV}\mathsf{ar}(\tau)}{(E_1; E_2)\{\alpha/\tau\}} \ (\mathsf{v}_1)$$

$$\frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{TV}\mathsf{ar}(\tau)}{(E_1; E_2)\{\alpha/\tau\}} \ (\mathsf{v}_2)$$

$$\frac{E_1; \tau \approx \tau; E_2}{E_1; E_2} \ (\mathsf{t})$$

Data Structures

- input: unification problem `type UP = [(Type, Type)]`

Data Structures

- input: unification problem `type UP = [(Type, Type)]`
- output: substitution `type TSub = [(Int, Type)]`

Data Structures

- input: unification problem `type UP = [(Type, Type)]`
- output: substitution `type TSub = [(Int, Type)]`
- unification: `unify :: UP -> TSub`

Data Structures

- input: unification problem `type UP = [(Type, Type)]`
- output: substitution `type TSub = [(Int, Type)]`
- unification: `unify :: UP -> TSub`

Applying and Composing Substitutions

```
tsub :: Type -> TSub -> Type
x@(TVar i)   `tsub` s = case lookup i s of
                          Just t -> t
                          _      -> x
(TFun t1 t2) `tsub` s = t1 `tsub` s @-> t2 `tsub` s
(TCon g ts)  `tsub` s = TCon g (map (`tsub` s) ts)

tcomp :: TSub -> TSub -> TSub
s2 `tcomp` s1 =
  map (\(x, t) -> (x, t `tsub` s2)) s1
    ++ filter ((`notElem` dom1) . fst) s2
  where dom1 = map fst s1
```

```haskell
unify :: UP -> TSub
unify eqs = unfy [] eqs
  where
    unfy mgu []          =  mgu
    unfy mgu (eq:eqs)  =  unfy (mgu' `tcomp` mgu)
      (e ++ map (\(l, r) ->
        (l `tsub` mgu', r `tsub` mgu')) eqs)
      where (e, mgu')  =  step eq
    step (s,t) | s == t  =  ([], [])
    step (TVar x, t)     =  singletonSub x t
    step (t, TVar x)     =  singletonSub x t
    step (TFun s t,TFun u v) = ([(s,u),(t,v)],[])
    step (TCon g ss,TCon f ts) | g==f = (zip ss ts,[])
    step _  =  notUnif
    notUnif  =  error "not unifiable"
    singletonSub x t =
      if x `elem` tvars t then notUnif
                          else ([], [(x,t)])
```

Implementing Type Inference

Recall

$$\frac{E, e :: \tau_0 \rhd e :: \tau_1}{\tau_0 \approx \tau_1}\ (\text{con})$$

$$\frac{E \rhd e_1\ e_2 :: \tau}{E \rhd e_1 :: \alpha \to \tau; E \rhd e_2 :: \alpha}\ (\text{app})$$

$$\frac{E \rhd \lambda x.\, e :: \tau}{E, x :: \alpha_1 \rhd e :: \alpha_2; \tau \approx \alpha_1 \to \alpha_2}\ (\text{abs})$$

$$\frac{E \rhd \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: \tau}{E \rhd e_1 :: \alpha; E, x :: \alpha \rhd e_2 :: \tau}\ (\text{let})$$

$$\frac{E \rhd \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 :: \tau}{E \rhd e_1 :: \text{Bool}; E \rhd e_2 :: \tau; E \rhd e_3 :: \tau}\ (\text{ite})$$

Data Structures

- input: environment `type Env = [(String, Type)]`

Data Structures

- input: environment `type Env = [(String, Type)]`
- output: `type IP = (Env, Env, Exp, Type)`

Data Structures

- input: environment `type Env = [(String, Type)]`
- output: `type IP = (Env, Env, Exp, Type)`
- computing typing constraints: `toUp :: IP -> UP`

Data Structures

- input: environment `type Env = [(String, Type)]`
- output: `type IP = (Env, Env, Exp, Type)`
- computing typing constraints: `toUp :: IP -> UP`

Auxiliary Functions

```
maxIdx (TVar i)      = i
maxIdx (TFun t1 t2) = max (maxIdx t1) (maxIdx t2)
maxIdx (TCon _ ts)  = maximum (0 : map maxIdx ts)

incBy i (TVar j)      = TVar (i + j)
incBy i (TFun t1 t2) = TFun (incBy i t1) (incBy i t2)
incBy i (TCon g ts)  = TCon g (map (incBy i) ts)

typeOf id env =
  case lookup id env of
    Just t  -> t
    Nothing -> error ("no type for '" ++ id ++ "'")
```

```haskell
toUp :: IP -> UP
toUp ip@(_,_,_,t) = step (maxIdx t + 1) [] [ip]
 where
  step i up []                =  up
  step i up ((cs,xs,e,t):ip) =
   case e of
    Var x          -> step i ((typeOf x xs,t):up) ip
    Con c          -> let t' = typeOf c cs in
     step (i + maxIdx t' + 1) ((incBy i t',t):up) ip
    App e1 e2      -> step (i+1) up
     ((cs,xs,e1,TVar i @-> t):(cs,xs,e2,TVar i):ip)
    Abs x e        -> step (i+2)
     ((t,TVar i @-> TVar (i+1)):up)
     ((cs,(x,TVar i):xs,e,TVar (i+1)):ip)
    Let x e1 e2  -> step (i+1) up
     ((cs,xs,e1,TVar i):(cs,(x,TVar i):xs,e2,t):ip)
    Ite e1 e2 e3 -> step i up
     ((cs,xs,e1,tbool):(cs,xs,e2,t):(cs,xs,e3,t):ip)
```

Type Inference

- most general type w.r.t. given environment and constraint

```
mgt :: Env -> Env -> Exp -> Type -> Type
mgt cs xs e t = t `tsub` mgu
  where mgu = unify (toUp (cs, xs, e, t))
```

Type Inference

- most general type w.r.t. given environment and constraint

```
mgt :: Env -> Env -> Exp -> Type -> Type
mgt cs xs e t = t `tsub` mgu
  where mgu = unify (toUp (cs, xs, e, t))
```

- most general type of expression

```
infer :: String -> Type
infer s = mgt primitives [] e (TVar (-1))
  where e = fromString s
```

Exercises (for January 21st)

1. Read the lecture notes about type checking and type inference.
2. Prepare for 2nd Test.