Computational Logic

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 11

Overview

- `CoreFP.hs` – A Module for Core FP Expressions
- `Unification.hs` – Implementing Unification
- `Typing.hs` – Implementing Type Inference

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Grammar of Core FP

e	$::=$	$x \mid e\ e \mid \lambda x.\, e$	λ-terms
	$\mid$	c	constant (for primitives)
	$\mid$	**let** $x = e$ **in** e	let binding
	$\mid$	**if** e **then** e **else** e	conditional branching

A Type for Core FP Expressions

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

Parsing Core FP Expressions from Strings

```
CoreFP.fromString :: String -> Exp
```

Grammar of Types

$$\tau ::= \alpha \mid \tau \to \tau \mid C(\tau, \ldots, \tau)$$

A Type for Types

```
data Type = TVar Int          – for easy renaming of type variables
          | TCon Id [Type]    – type constructors and function type
  deriving Eq                 – type equality can be checked
```

Syntactic Sugar for Function Types

- user-defined right-associative infix operator with precedence 8

```
infixr 8 ~>

(~>) :: Type -> Type -> Type
s ~> t = TCon "Fun" [s, t]
```


Typing Environments

```
type Env = [(String, Type)] -- constants/variables -> types
```

The Free Variables of a Type

```
tvars :: Type -> [Int]
tvars (TVar x)    = [x]
tvars (TCon g ts) = foldr (union . tvars) [] ts
```

"Extracting" Maybes with Default

- `Data.Maybe.fromMaybe :: a -> Maybe a -> a`
- satisfies equations

$$\texttt{fromMaybe}\ x\ \texttt{Nothing} = x$$
$$\texttt{fromMaybe}\ x\ (\texttt{Just}\ y) = y$$

Type Substitutions

```
type TSub = [(Int, Type)]
```

Applying Substitutions to Types

```
tsub :: TSub -> Type -> Type
tsub s (x@(TVar i)) = fromMaybe x $ lookup i s
tsub s (TCon g ts)  = TCon g (map (tsub s) ts)
```

Composition

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


An Environment for Primitives

```
tint, tbool :: Type
tint  = TCon "Int" []
tbool = TCon "Bool" []

tpair :: Type -> Type -> Type
tpair s t  = TCon "Pair" [s, t]

-- assumption: involved type variables are non-negative
primitives :: Env
primitives = [
  ("True", tbool), ("False", tbool),
  ("<", tint ~> tint ~> tbool),
  ("+", tint ~> tint ~> tint),
  ("0", tint),
  ("Pair", a0 ~> a1 ~> tpair a0 a1), ...]
  where a0 = TVar 0
        a1 = TVar 1
```

Interface

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

Applying Type Substitutions to Unification Problems

```
tsubUP :: TSub -> UP -> UP
tsubUP s = map (\(l, r) -> (tsub s l, tsub s r))
```


Unification

```
unify :: UP -> TSub
unify eqs = go [] eqs
  where
    go mgu []       = mgu
    go mgu (eq:eqs) =
      go (mgu `tcomp` mgu') (eqs' ++ tsubUP mgu' eqs)
      where
        (eqs', mgu') = step eq
    step :: (Type, Type) -> (UP, TSub)
    ...
```

Removal of Trivial Equations

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

```
step (s, t) | s == t = ([], [])
```


Variable Elimination

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}}\ (\mathsf{v}_1) \qquad \frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}}\ (\mathsf{v}_2)$$

```
step (TVar x, t) = singletonSub x t
step (t, TVar x) = singletonSub x t
  where
    notUnif = error "not unifiable"
    singletonSub x t =
      if x `elem` tvars t then notUnif else ([], [(x, t)])
```

Decomposition

$$\frac{E_1; C(\tau_1, \ldots, \tau_n) \approx C(\tau'_1, \ldots, \tau'_n); E_2}{E_1; \tau_1 \approx \tau'_1; \ldots; \tau_n \approx \tau'_n; E_2}\ (\mathsf{d}_1) \qquad \frac{E_1; \tau_1 \to \tau_2 \approx \tau'_1 \to \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2}\ (\mathsf{d}_2)$$

```
step (TCon g ss, TCon f ts) |
  g == f && length ss == length ts = (zip ss ts, [])
```


Interface

- input: type inference problem `type IP = (Env, Exp, Type)`
- computing typing constraints: `toUp :: IP -> UP`

Auxiliary Functions

```
maxIdx (TVar i)    = i
maxIdx (TCon _ ts) = maximum (0 : map maxIdx ts)

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

typeOf name env =
  fromMaybe (error ("no type for '" ++ name ++ "'")) $
    lookup name env
```

Computing Typing Constraints

```
toUp :: IP -> UP
toUp (env, e, tau) = go (maxIdx tau + 1) [(env, e, tau)]
  where
    go i []       = []
    go i (ip:ips) = up ++ go i' (ips' ++ ips)
      where
        (i', up, ips') = step i ip
        step :: Int -> IP -> (Int, UP, [IP])
        ...
```

Consistency with the Environment (Polymorphic Constants)

$$\frac{E, e :: \tau' \rhd e :: \tau}{\tau' \approx \tau}\ (\text{con})$$

```
step i (env, Var x, t) = (i, [(typeOf x env, t)], [])
step i (env, Con c, t) = (maxIdx t' + 1, [(t', t)], [])
  where t' = incBy i $ typeOf c env
```


Application

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

```
step i (env, App e1 e2, tau) = (i+1, [],
  [(env, e1, TVar i ~> tau),
   (env, e2, TVar i)])
```

Abstraction

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

```
step i (env, Abs x e, tau) = (i+2,
  [(tau, TVar i ~> TVar (i+1))],
  [((x, TVar i) : env, e, TVar (i+1))])
```


Let-Bindings

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

```
step i (env, Let x e1 e2, tau) = (i+1, [],
  [(env, e1, TVar i),
   ((x, TVar i) : env, e2, tau)])
```

Conditional Branching

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

```
step i (env, Ite e1 e2 e3, tau) = (i, [],
  [(env, e1, tbool),
   (env, e2, tau),
   (env, e3, tau)])
```


Type Inference

- most general type with respect to given environment and constraint

```
mgt :: IP -> Type
                    -- "seq" forces evaluation of "mgu"
mgt ip@(_, _, t) = mgu `seq` tsub mgu t
  where
    mgu = unify $ toUp ip
```

- most general type of expression

```
infer :: String -> Type
infer s = mgt (primitives, e, TVar 0)
  where
    e = fromString s
```

Definitions

```
concat []     = []                 length []     = 0
concat (x:xs) = x ++ concat xs     length (x:xs) = 1 + length xs

map f []      = []                 sum []        = 0
map f (x:xs)  = f x : map f xs     sum (x:xs)    = x + sum xs
```

Exercises (for January 26th)

1. Which of the above functions are tail recursive, which guardedly recursive, and which neither.
2. Evaluate `concat $ map (\x -> [x,x]) [1,2,3]` by equational reasoning.
3. Prove `length` $(\texttt{concat}\ xs) = \texttt{sum}\ (\texttt{map length}\ xs)$ for all xs.
4. Implement a tail recursive function computing `(length . concat)`.
5. Prove the typing judgment $P \vdash (\lambda xy.\, x)$ True $:: \alpha_0 \to$ Bool.
6. Use type inference to compute the most general type of fst (Pair False 0) with respect to the primitive environment P.