Compiling functional languages

http://www.cse.chalmers.se/edu/year/2011/course/CompFun/

Lecture 4
Type inference

Johan Nordlander

Types

- A means of classifying programs/functions/terms/variables, in order to filter out nonsense
- Trivial example:
  - 3.2/x is a Float if x is a Float
  - 3.2/x is nonsense if x is a String
- Typical limitation:
  - 3.2/x is still a Float, even if x might be 0.0

Types

- A means of abstractly describing programs/functions/terms/variables
- Trivial example:
  - `sortBy :: (a -> a -> Bool) -> [a] -> [a]`
  - Provides some useful info about a term whose implementation is hidden
- Typical limitation:
  - `sortBy` could still be defined as (`\p xs -> []`) ...

Compilation issues

- Type-checking:
  - Find out if a program has a given type
- Type-inference:
  - Find out if a program has any type at all
  - Find some type for a program (if it has one)
  - Find the "best" type for a program (if it might have more than one type)

Type syntax

- Concretely:

  $t \;::=\; a \mid T \mid t\ t \mid t \rightarrow t \mid [t] \mid (t_1,\ldots,t_n)$

- C.f. haskell-src:

```
data HsType   = HsTyFun HsType HsType
              | HsTyTuple [HsType]
              | HsTyApp HsType HsType
              | HsTyVar HsName
              | HsTyCon HsQName
```

- Type desugaring:

  $[t] \Rightarrow [\,]\ t \qquad (t_1,\ldots,t_n) \Rightarrow (,\ldots,)\ t_1 \ldots t_n$

- I.e., a list or tuple constructor is just a T

Correct types

- Under what conditions does term e have type t?
- Usually described in informal language reports...
- Must be matched by type-checking/inference algorithms implemented in compilers
- Advantage of the functional language heritage:

A rich tradition of defining <u>type correctness</u> formally using <u>logical inference systems</u>!

Simple type correctness

Scope = set of type assumptions on variables

Premises

$$\frac{A \vdash e : t' \mathbin{\text{->}} t \qquad A \vdash e' : t'}{A \vdash e\ e' : t}\ \text{App}$$

Conclusion

Extension

$$\frac{A, x{:}t' \vdash e : t}{A \vdash \backslash x \mathbin{\text{->}} e : t' \mathbin{\text{->}} t}\ \text{Abs}$$

Right-biased lookup

$$\frac{x{:}t \in A}{A \vdash x : t}\ \text{Var}$$

\x->e has type t'->t in scope A...

... if e has type t in scope A extended with x of type t'

Example

$$\dfrac{\dfrac{\dfrac{odd{:}I\text{->}B \in odd{:}I\text{->}B, x{:}I}{odd{:}I\text{->}B,\ x{:}I \vdash odd : I \text{ -> } B}\ \text{Var} \qquad \dfrac{x{:}I \in odd{:}I\text{->}B, x{:}I}{odd{:}I\text{->}B,\ x{:}I \vdash x : I}\ \text{Var}}{odd{:}I\text{->}B,\ x{:}I \vdash odd\ x : B}\ \text{App}}{odd{:}I\text{->}B \vdash \backslash x \text{ -> } odd\ x : I \text{ -> } B}\ \text{Abs}$$

Cannot be derived: $odd{:}I\text{->}B \vdash odd\ odd : B$

Datatypes and case

$$\frac{A \vdash e : T\ ts \qquad\qquad A, xs_i{:}[ts/as]ts_i \vdash e_i : t}{A \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\ K_i\ xs_i \to e_i\ \}: t}\ \mathit{Case}$$

where **data** $T\ as = K_i\ ts_i \in$ top-decls

$$\frac{A \vdash es : [ts/as]ts_j}{A \vdash K_j\ es : T\ ts}\ \mathit{Con}$$

where **data** $T\ as = K_i\ ts_i \in$ top-decls

Polymorphism

Type variables have scope as well:

```
id :: a -> a
id = \x -> x

f :: a -> a -> (a, Char)
f = \x y -> let g :: a -> a
                g = \x -> x
            in (g x, g 'X')
```

Not the same a!

So where do type variables get <u>bound</u>?

The Hindley/Milner approximation

- Type variables are universally quantified at the outermost type expressions only
- Implicitly present in Haskell/ML/etc:

```
id :: ∀a . a -> a
id = \x -> x

f :: ∀a . a -> a -> (a, Char)
f = \x y -> let g :: ∀a . a -> a
                g = \x -> x
            in (g x, g 'A')
```

A phrase of the form

$\forall a_1 \ldots \forall a_n . t$

is called a type scheme, which is not a type itself

Hindley/Milner polymorphism

- Types:

  $t \quad ::= \quad a \mid T \mid t\ t \mid t \to t$

- Type schemes:

  $\sigma \quad ::= \quad \forall a\,.\,\sigma \mid t$

- Assumptions:

  $A \quad ::= \quad x_1 : \sigma_1\,,\ \ldots\,,\ x_n : \sigma_n$

Free type variables:

$fv(\forall a.\sigma) = fv(\sigma) \setminus \{a\}$

Judgements:

$A \vdash e : \sigma$

Note that all types count as schemes, but not vice versa

Also note that a variable stands for a type, not a scheme

Hindley/Milner polymorphism

$$\frac{A \vdash e : t' \to t \qquad A \vdash e' : t'}{A \vdash e\ e' : t}\ \text{App} \qquad\qquad \frac{A, x{:}t' \vdash e : t}{A \vdash \backslash x \to e : t' \to t}\ \text{Abs}$$

$$\frac{x{:}\sigma \in A}{A \vdash x : \sigma}\ \text{Var} \qquad\qquad \frac{A \vdash e : \sigma \qquad A, x{:}\sigma \vdash e' : t}{A \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : t}\ \text{Let}$$

merge

merge

$$\frac{A \vdash e : \forall a.\sigma}{A \vdash e : [t/a]\sigma}\ \text{Inst} \qquad\qquad \frac{A \vdash e : \sigma \qquad a \notin fv(A)}{A \vdash e : \forall a.\sigma}\ \text{Gen}$$

Hindley/Milner polymorphism

$$\frac{A \vdash e : t' \to t \qquad A \vdash e' : t'}{A \vdash e\ e' : t}\ \text{App}$$

$$\frac{A, x{:}t' \vdash e : t}{A \vdash \backslash x \to e : t' \to t}\ \text{Abs}$$

$$\frac{x{:}\forall as.t \in A}{A \vdash x : [ts/as]t}\ \text{Var}$$

$$\frac{A \vdash e : t \qquad A, x{:}\sigma \vdash e' : t'}{A \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : t'}\ \text{Let}$$

where $\sigma = gen(t,A) = \forall\ fv(t) \backslash fv(A)\ .\ t$

(Type schemes: $\sigma\ ::=\ \forall as\ .\ t$)

Example

$$
\dfrac{
\dfrac{\dfrac{x{:}a \in x{:}a}{x{:}a \vdash x : a}\,\text{Var}}{\vdash \backslash x\text{->}x : a\text{->}a}\,\text{Abs}
\qquad
\dfrac{
\dfrac{f{:}\forall a.a\text{->}a \in A}{A \vdash f : [B\text{->}B/a]a\text{->}a}\,\text{Var}
\qquad
\dfrac{f{:}\forall a.a\text{->}a \in A}{A \vdash f : [B/a]a\text{->}a}\,\text{Var}
}{f{:}\forall a.a\text{->}a \vdash f\ f : B\text{->}B}\,\text{App}
}{\vdash \mathbf{let}\ f = \backslash x\text{->}x\ \mathbf{in}\ f\ f : B\text{->}B}\,\text{Let}
$$

where $A = f{:}\forall a.a\text{->}a$

Example

Cannot be derived:

Incorrect generalization:

$$\dfrac{\dfrac{\dfrac{x:a \in x:a}{x:a \vdash x : a}\,\text{Var} \qquad \dfrac{\dfrac{f:\forall a.a \in \ldots}{\ldots \vdash f : [B\text{->}B/a]a}\,\text{Var} \quad \dfrac{f:\forall a.a \in \ldots}{\ldots \vdash f : [B/a]a}\,\text{Var}}{f:\forall a.a,\ x:a \vdash f\ f : B}\,\text{App}}{x:a \vdash \textbf{let}\ f = x\ \textbf{in}\ f\ f : B}\,\text{Let}}{\vdash \backslash x\ \text{->}\ \textbf{let}\ f = x\ \textbf{in}\ f\ f : B}\,\text{Abs}$$

$$\forall a.a \neq gen(a, (x:a)) = \forall\ fv(a) \setminus fv(x:a)\ .\ a = \forall\ [\,]\ .\ a = a$$

Example

Cannot be derived:

$$\dfrac{\dfrac{\dfrac{f{:}\forall a.a\to a \in \ldots}{\ldots \vdash f : [b\to b/a]a\to a}\,\text{Var} \qquad \dfrac{f{:}\forall a.a\to a \in \ldots}{\ldots \vdash f : [b/a]a\to a}\,\text{Var}}{f{:}\forall a.a\to a \vdash f\ f : b\to b}\,\text{App}}{\vdash \backslash f \to f\ f : (\forall a.a\to a) \to b\to b}\,\text{Abs}$$

Must be a <u>type</u>!

Neither a type, nor a scheme!

Why schemes ≠ types?

- Because of decidability of type inference!
- The type inference problem:

  Given an e and an A, find the most general t such that $A \vdash e : t$

- Amounts to traversing e, guessing yet unknown types, and adjusting the guesses when needed
- Guessing a type: invent a fresh type variable
- Adjusting a guess: unification
- (Guessing and adjusting type schemes: undecidable!)

Hindley/Milner polymorphism

Adjust guesses to make equal

$$\frac{A \vdash e : t' \to t \qquad A \vdash e' : t'}{A \vdash e\ e' : t}\ \text{App}$$

Guess

$$\frac{A, x{:}t' \vdash e : t}{A \vdash \backslash x \to e : t' \to t}\ \text{Abs}$$

$$\frac{x{:}\forall as.t \in A}{A \vdash x : [ts/as]t}\ \text{Var}$$

Guess

$$\frac{A \vdash e : t \qquad A, x{:}\sigma \vdash e' : t'}{A \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : t'}\ \text{Let}$$

where $\sigma = gen(t, A) = \forall\ fv(t) \backslash fv(A)\ .\ t$

Hindley/Milner type inference sketch

Make equal...

$$\frac{A \vdash^{w} e : t \qquad A \vdash^{w} e' : t' \qquad t \sim t' \to a}{A \vdash^{w} e\ e' : a}\ \text{App}$$

where a is new

Guess

$$\frac{A, x{:}a \vdash^{w} e : t}{A \vdash^{w} \backslash x \to e : a \to t}\ \text{Abs}$$

where a is new

Guess

$$\frac{x{:}\forall as.t \in A}{A \vdash^{w} x : [bs/as]t}\ \text{Var}$$

where bs are new

$$\frac{A \vdash^{w} e : t \qquad A, x{:}\sigma \vdash^{w} e' : t'}{A \vdash^{w} \textbf{let}\ x = e\ \textbf{in}\ e' : t'}\ \text{Let}$$

where $\sigma = gen(t, A) = \forall\ fv(t) \backslash fv(A)\ .\ t$

Unification

- Unification is the process of finding a substitution that solves an equation
- E.g. a -> Int = Bool -> b is solved by [Bool/a, Int/b]
- Robinson's algorithm from 1965 finds a <u>most general solution</u> to an equation, if a solution exists at all

Robinson's unification algorithm

General form: $t_1 \overset{\theta}{\sim} t_2$

Output (a substitution)

Inputs (two types)

$t \overset{[t/a]}{\sim} a$ if $a \notin fv(t)$ $\qquad$ $a \overset{[t/a]}{\sim} t$ if $a \notin fv(t)$ $\qquad$ $T \overset{[]}{\sim} T$

$$\frac{t_1 \overset{\theta_1}{\sim} t_3 \qquad \theta_1 t_2 \overset{\theta_2}{\sim} \theta_1 t_4}{t_1\ t_2 \overset{\theta_2\theta_1}{\sim} t_3\ t_4} \qquad\qquad \frac{t_1 \overset{\theta_1}{\sim} t_3 \qquad \theta_1 t_2 \overset{\theta_2}{\sim} \theta_1 t_4}{t_1 \to t_2 \overset{\theta_2\theta_1}{\sim} t_3 \to t_4}$$

Composition of substitutions: $(\theta_2\theta_1)a = \theta_2(\theta_1 a)$

The algorithm fails if no rule is applicable

Full Hindley/Milner type inference (algorithm W)

General form:

Input (an environment and an expression)

$$\theta A \vdash^{W} e : t$$

Output (a type and a substitution on the environment)

(W for "well-typed" (Milner))

The algorithm fails if any internal unification attempt fails

Full Hindley/Milner type inference (algorithm W)

$$\frac{\theta_1 A \vdash^W e : t \quad \theta_2(\theta_1 A) \vdash^W e' : t' \quad \theta_2 t \overset{\theta_3}{\sim} t' \text{->} a}{(\theta_3\theta_2\theta_1)A \vdash^W e\ e' : \theta_3\, a}\ \text{App}$$

where a is new

$$\frac{\theta(A, x{:}a) \vdash^W e : t}{\theta A \vdash^W \backslash x \text{ -> } e : \theta a \text{ -> } t}\ \text{Abs}$$

where a is new

$$\frac{x{:}\forall as.t \in A}{[\,]A \vdash^W x : [bs/as]t}\ \text{Var}$$

where bs are new

$$\frac{\theta_1 A \vdash^W e : t \qquad \theta_2(\theta_1 A, x{:}\sigma) \vdash^W e' : t'}{(\theta_2\theta_1)A \vdash^W \textbf{let}\ x = e\ \textbf{in}\ e' : t'}\ \text{Let}$$

where $\sigma = gen(t, \theta_1 A) = \forall\ fv(t)\backslash fv(\theta_1 A)\ .\ t$

Properties of algorithm W

- (Soundness)
  - If $\theta A \vdash^{W} e : t$ succeeds then $\theta A \vdash e : t$
- (Completeness)
  - If $\theta A \vdash e : t$ then $\theta' A \vdash^{W} e : t'$ succeeds such that for some θ'':
    - $t = \theta'' t'$
    - $\theta = \theta'' \theta'$
- Note: on the top-level, $fv(A) = []$, so $\theta A = A$ for all θ

Datatypes and case

$$\frac{\theta_1 A \vdash^{w} e{:}t \qquad \theta_2(\theta_1 A, xs_i{:}[bs/as]ts_i) \vdash^{w} e_i{:}t_i \qquad \theta_2(T\ bs) \to t_i \overset{\theta_3}{\sim} \theta_2 t \to a}{(\theta_3\theta_2\theta_1)A \vdash^{w} \textbf{case}\ e\ \textbf{of}\ \{\ K_i\ xs_i \to e_i\ \} : \theta_3\ a}\ \mathit{Case}$$

where **data** $T\ as = K_i\ ts_i \in$ top-decls and a, bs are new

(Details of unification and substitution threading <u>for all i</u> left as an exercise!)

$$\frac{\theta_1 A \vdash^{w} es : ts \qquad [bs/as]ts_j \overset{\theta_2}{\sim} ts}{(\theta_2\theta_1)A \vdash^{w} K_j\ es : \theta_2(T\ bs)}\ \mathit{Con}$$

where **data** $T\ as = K_i\ ts_i \in$ top-decls and bs are new

Recursion

Note different assumptions!

$$\frac{A, x{:}t \vdash e : t \qquad A, x{:}\sigma \vdash e' : t'}{A \vdash \textbf{let } x = e \textbf{ in } e' : t'}\,\text{Let}$$

where $\sigma = gen(t, A)$

$$\frac{\theta_1(A, x{:}a) \vdash^{w} e : t \qquad \theta_1 a \overset{\theta_2}{\sim} t \qquad \theta_3(\theta_2\theta_1 A, x{:}\sigma) \vdash^{w} e' : t'}{(\theta_3\theta_2\theta_1)A \vdash^{w} \textbf{let } x = e \textbf{ in } e' : t'}\,\text{Let}$$

where a is new and $\sigma = gen(\theta_2 t, \theta_2\theta_1 A)$

Generalization to mutual recursion straightforward (but space-consuming)

Explicit signatures

$$\frac{A \vdash e : t \qquad gen(t, A) \preccurlyeq \sigma \qquad A, x{:}\sigma \vdash e' : t'}{A \vdash \mathbf{let}\ x :: \sigma;\ x = e\ \mathbf{in}\ e' : t'}\ \text{Let}$$

where $\forall as.t_a \preccurlyeq \forall bs.t_b$ iff for <u>all</u> bs there <u>exist</u> as such that $t_a = t_b$

Matching ≈ <u>one-way</u> unification…

$$\frac{\theta_1 A \vdash^{W} e : t \qquad gen(t, \theta_1 A) \overset{\theta_2}{\preccurlyeq} \sigma \qquad \theta_3(\theta_2\theta_1 A, x{:}\sigma) \vdash^{W} e' : t'}{(\theta_3\theta_2\theta_1)A \vdash^{W} \mathbf{let}\ x :: \sigma;\ x = e\ \mathbf{in}\ e' : t'}\ \text{Let}$$

Generalization to (mutual) recursion straightforward, but notice opportunity to use $x{:}\sigma$ as assumption when checking e — Haskell's <u>polymorphic recursion</u>!

Matching

- Def: $\forall as.t_a \preccurlyeq \forall bs.t_b$ iff $\forall bs\,.\,\exists as\,.\,t_a = t_b$
- Matching algorithm $\forall as.t_a \overset{\theta}{\preccurlyeq} \forall bs.t_b$ defined as: find the smallest θ such that $\theta(\forall as.t_a) \preccurlyeq \theta(\forall bs.t_b)$
- Isn't $t_a \overset{\theta}{\sim} t_b$ sufficient? No, in addition:
  - θ must not touch bs ($dom(\theta) \cap bs = \varnothing$)
  - θ must not let bs escape ($fv(\theta(\forall as.t_a)) \cap bs = \varnothing$ and $fv(\theta(\forall bs.t_b)) \cap bs = \varnothing$)
- Can be explicitly checked, of course. Alternatively...

Skolemization

- Method for solving equations under nested $\forall$ and $\exists$
- Note our general problem: $\exists as' \,.\, \forall bs \,.\, \exists as \,.\, t_a = t_b$ where $as' = fv(\forall as.t_a) \cup fv(\forall bs.t_b)$
- Skolemized equivalent: $\exists as' \,.\, \exists as \,.\, t_a = \phi\, t_b$ where ϕ is a substitution that maps each b_i in bs to $T_i\ as'$ and each T_i is a newly invented type constructor
- This problem is efficiently solved by $t_a \overset{\theta}{\sim} \phi\, t_b$

Summary

- The Hindley/Milner stratification:
  - Types (including variables)
  - Type schemes = types with universal quantifiers
- Two similar formal systems
  - Logical proof of type correctness: $A \vdash e : t$
  - Algorithm for inferring m.g. types: $\theta A \vdash^{W} e : t$
- Algorithm W based on unification and fresh names
- Challenge: implement unification, substitution and matching without getting too clever!!!