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 <u>filter out nonsense</u>
- 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 <u>abstractly describing</u> programs/ functions/terms/variables
- Trivial example:
 - sortBy :: (a -> a -> Bool) -> [a] -> [a]
 - Provides <u>some</u> 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 <u>any type at all</u>
 - Find some type for a program (if it has one)
 - Find the "<u>best</u>" type for a program (if it might have more than one type)

Type syntax

- Concretely:
 - $t ::= a | T | t t | t -> t | [t] | (t_1,...,t_n)$
- C.f. haskell-src:
 - data HsType = HsTyFun HsType HsType | HsTyTuple [HsType] | HsTyApp HsType HsType | HsTyVar HsName | HsTyCon HsQName
- Type desugaring:
 [†] => []†

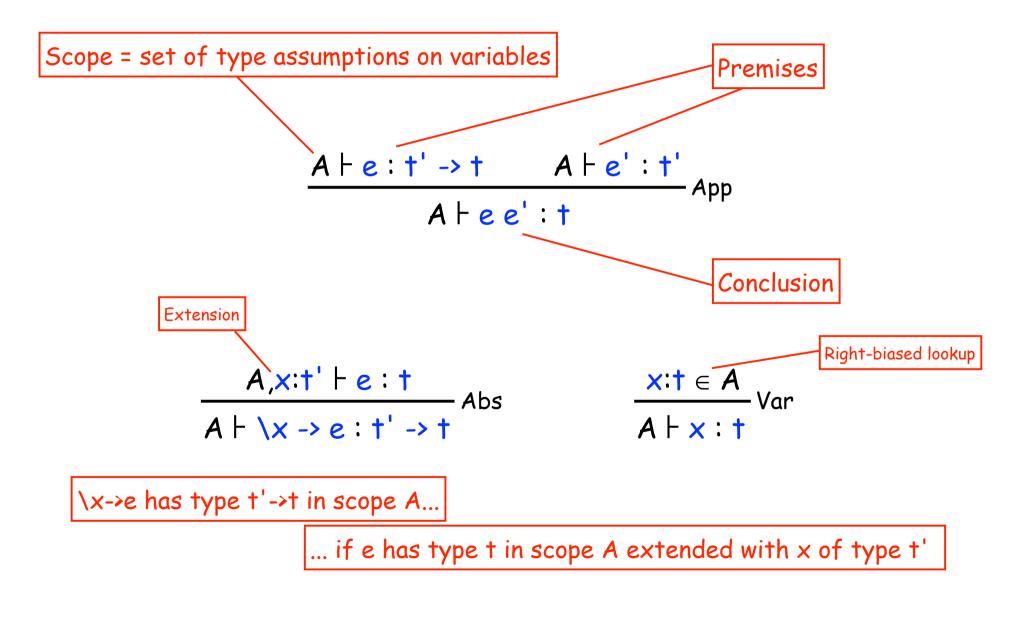
• I.e., a list or tuple constructor is just a T

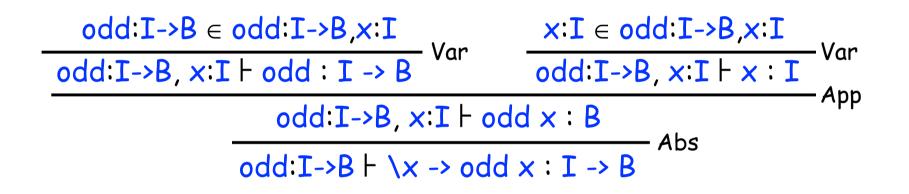
Correct types

- 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





Cannot be derived: odd: I->B + odd odd : B

Datatypes and case

 $\frac{A \vdash e : T ts}{A \vdash case e of \{ K_i \times s_i \rightarrow e_i \} : t} Case$

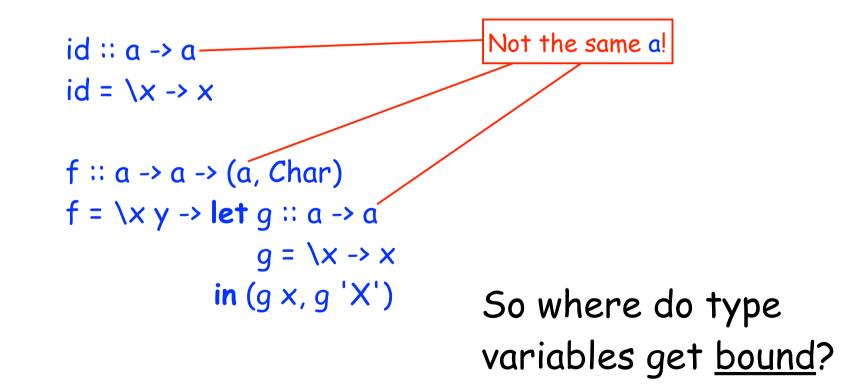
where data T as = K_i ts_i \in top-decls

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

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

Polymorphism

Type variables have scope as well:



The Hindley/Milner approximation

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

```
id :: \forall a . a \rightarrow a

id = \backslash x \rightarrow x

f :: \forall a . a \rightarrow a \rightarrow (a, Char)

f = \backslash x y \rightarrow let g :: \forall a . a \rightarrow a

g = \backslash x \rightarrow x

in (g x, g A')
```

```
A phrase of the form
```

```
\forall a_1 \dots \forall a_n . \dagger
```

```
is called a <u>type scheme,</u>
which is <u>not a type</u> itself
```

- Types:
 - t ::= a | T | tt | t-> t
- Type schemes: $\sigma ::= \forall a, \sigma \mid t$
- Assumptions:

A ::= $x_1 : \sigma_1 , \dots , x_n : \sigma_n$

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

$$\frac{A \vdash e: t' \rightarrow t}{A \vdash e': t} A \vdash e': t} A \downarrow PP \qquad \qquad \frac{A, x: t' \vdash e: t}{A \vdash x \rightarrow e: t' \rightarrow t} Abs$$

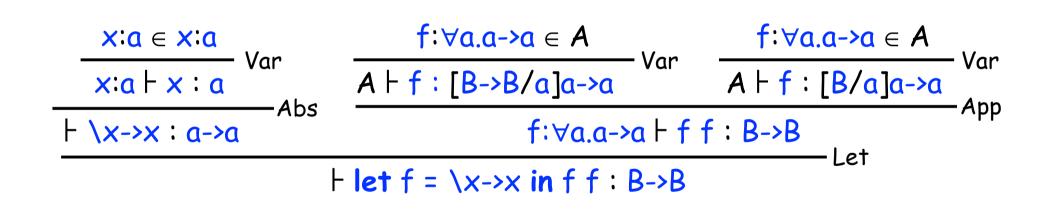
$$\frac{x:\sigma \in A}{A \vdash x:\sigma} \quad Var \qquad A \vdash e:\sigma \qquad A, x:\sigma \vdash e':t \\ A \vdash let x = e \text{ in } e':t \\ A \vdash let x = e \text{ in } e':t \\ merge \qquad \frac{A \vdash e: \forall a.\sigma}{A \vdash e: [t/a]\sigma} \quad Inst \qquad \frac{A \vdash e:\sigma \qquad a \notin fv(A)}{A \vdash e: \forall a.\sigma} Gen$$

$$\frac{A \vdash e: t' \rightarrow t}{A \vdash e': t} A pp \qquad \qquad \frac{A, x: t' \vdash e: t}{A \vdash x \rightarrow e: t' \rightarrow t} Abs$$

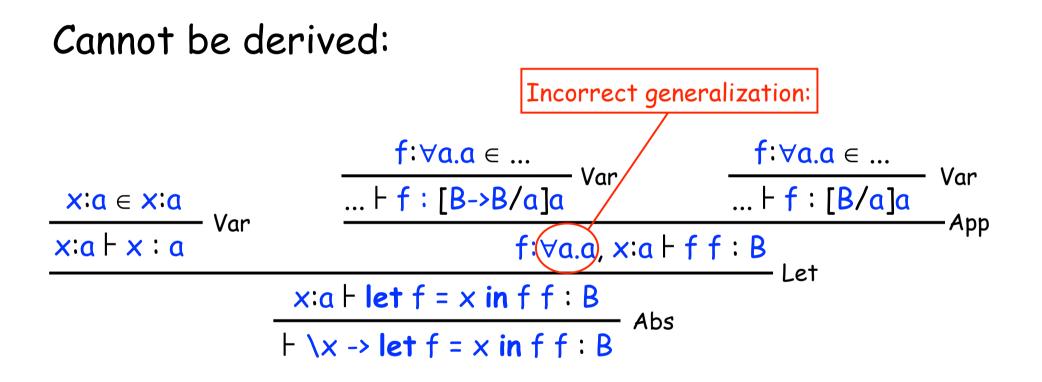
$$\frac{x:\forall as.t \in A}{A \vdash x: [ts/as]t} \quad Var \qquad \frac{A \vdash e:t \qquad A, x:\sigma \vdash e':t'}{A \vdash let x = e in e':t'} Let$$

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

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

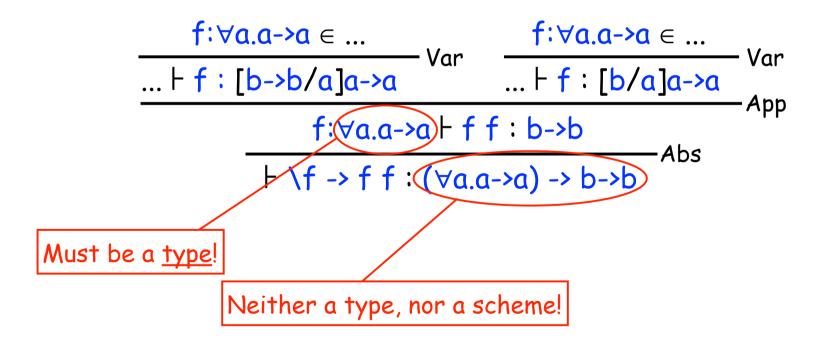


where $A = f: \forall a.a->a$



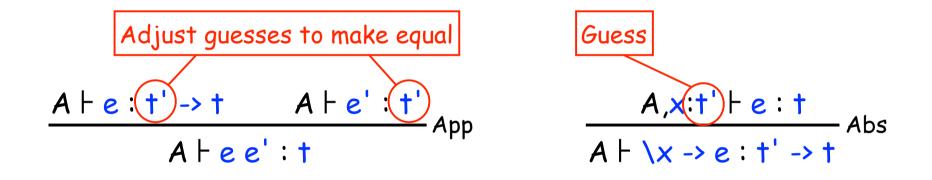
 $\forall a.a \neq gen(a, (x:a)) = \forall fv(a) \setminus fv(x:a) . a = \forall [] . a = a$

Cannot be derived:



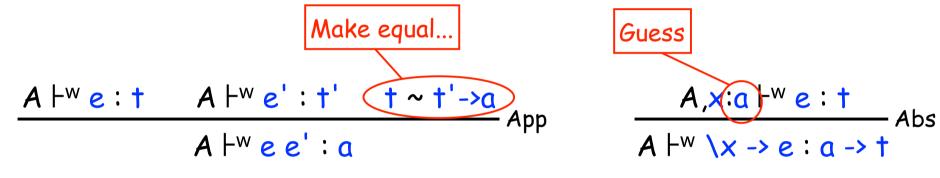
Why schemes ≠ types?

- Because of decidability of type inference!
- The type inference problem:
 Given an e and an A, find the most general t such that A + e : t
- Amounts to <u>traversing</u> e, <u>guessing</u> yet unknown types, and <u>adjusting</u> the guesses when needed
- Guessing a type: invent a <u>fresh type variable</u>
- Adjusting a guess: <u>unification</u>
- (Guessing and adjusting type schemes: undecidable!)



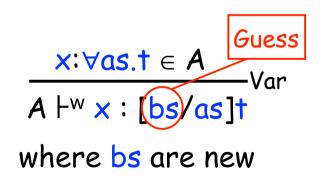
$$\frac{A \vdash e: t}{A \vdash e: t} = \frac{A, x: \sigma \vdash e': t'}{A \vdash e + x = e \text{ in } e': t'}$$
where $\sigma = gen(t, A) = \forall fv(t) \setminus fv(A) \cdot t$

Hindley/Milner type inference sketch



where a is new

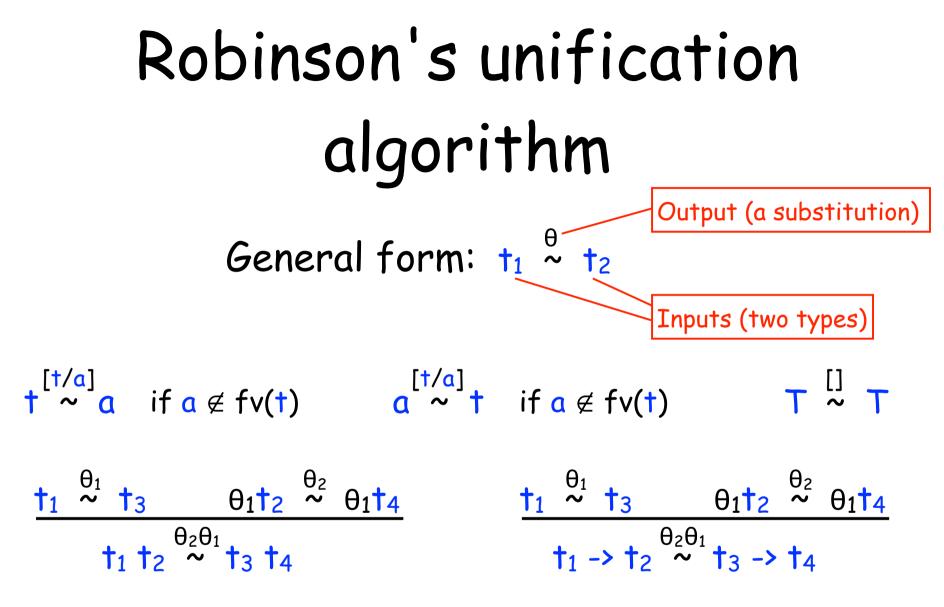
where a is new



$$\frac{A \vdash w e : t}{A \vdash w e : t} = \frac{A, x: \sigma \vdash w e' : t'}{A \vdash w \text{ let } x = e \text{ in } e' : t'}$$
where $\sigma = gen(t, A) = \forall fv(t) \setminus fv(A) . t$

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</u> <u>general solution</u> to an equation, if a solution exists at all

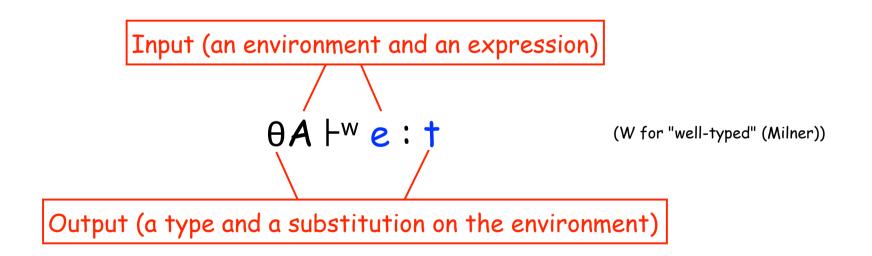


Composition of substitutions: $(\theta_2 \theta_1)a = \theta_2(\theta_1a)$

The algorithm fails if no rule is applicable

Full Hindley/Milner type inference (algorithm W)

General form:



The algorithm fails if any internal unification attempt fails

$$\frac{\theta_1 A \vdash^{w} e : t \quad \theta_2(\theta_1 A) \vdash^{w} e' : t' \quad \theta_2 t \stackrel{\theta_3}{\sim} t' \rightarrow a}{(\theta_3 \theta_2 \theta_1) A \vdash^{w} e e' : \theta_3 a} App \qquad \frac{\theta(A, x:a) \vdash^{w} e : t}{\theta A \vdash^{w} \langle x \rightarrow e : \theta a \rightarrow t} Abs$$
where a is new

x:
$$\forall as.t \in A$$

[]A $\vdash^w x$: [bs/as]t $\theta_1 A \vdash^w e : t$
 $(\theta_2 \theta_1) A \vdash^w let x = e in e' : t'$
 $(\theta_2 \theta_1) A \vdash^w let x = e in e' : t'$
where bs are newLet
 $\psi here \sigma = gen(t, \theta_1 A) = \forall fv(t) \setminus fv(\theta_1 A) . t$

Properties of algorithm W

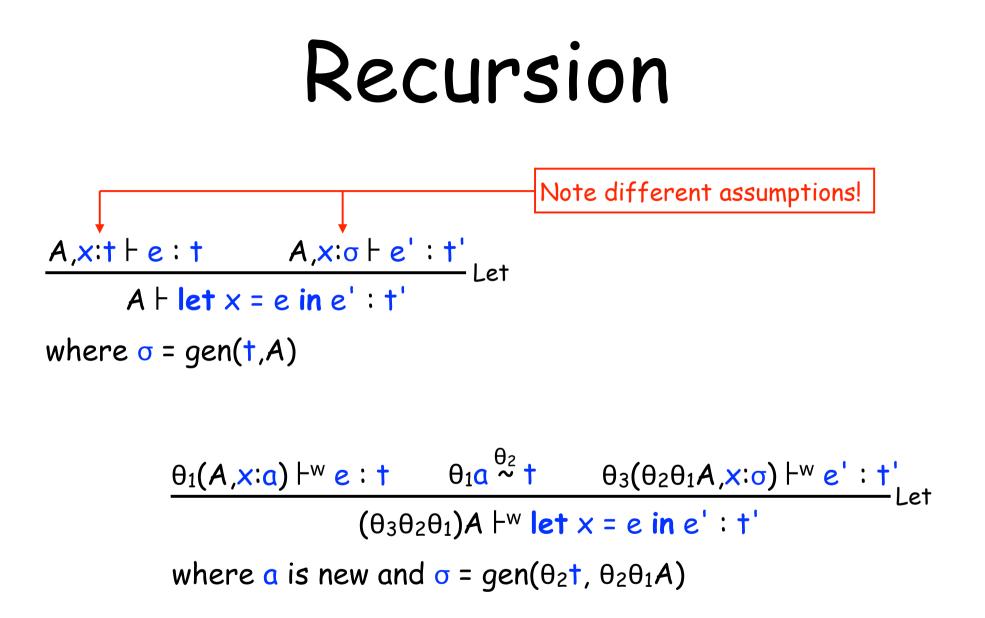
- (Soundness)
 - If $\theta A \vdash w e$: t succeeds then $\theta A \vdash e$: t
- (Completeness)
 - If $\theta A \vdash e : t$ then $\theta' A \vdash w e : t'$ succeeds such that for some θ'' :
 - t = θ"t'
 - θ = θ"θ'
- Note: on the top-level, fv(A) = [], so $\theta A = A$ for all θ

Datatypes and case

 $\begin{array}{lll} \theta_1 A \vdash^w e:t & \theta_2(\theta_1 A, xs_i: [bs/as]ts_i) \vdash^w e_i:t_i & \theta_2(T bs) \rightarrow t_i \stackrel{\theta_3}{\sim} \theta_2 t \rightarrow a \\ & (\theta_3 \theta_2 \theta_1) A \vdash^w \textbf{case e of } \{ K_i \ xs_i \rightarrow e_i \} : \theta_3 a \\ & \text{where data T as = } K_i \ ts_i \ \in \ \text{top-decls and } a, \ bs \ are \ new \\ & (\text{Details of unification and substitution threading } \underbrace{\text{for all } i} \text{ left as an exercise!} \end{array}$

$$\frac{\theta_1 A \vdash^{w} es : ts}{(\theta_2 \theta_1) A \vdash^{w} K_j es : \theta_2(T bs)} Con$$

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



Generalization to mutual recursion straightforward (but space-consuming)

Explicit signatures

 $\frac{A \vdash e: t \quad gen(t,A) \leq \sigma \quad A, x: \sigma \vdash e': t'}{A \vdash let \ x:: \sigma; \ x = e \text{ in } e': t'} \text{ Let}$ where $\forall as.t_a \leq \forall bs.t_b \text{ iff for } all \text{ bs there } exist \text{ as such that } t_a = t_b$ $\underbrace{\theta_1 A \vdash^w e: t \quad gen(t,\theta_1 A) \stackrel{\theta_2}{\leq} \sigma \quad \theta_3(\theta_2 \theta_1 A, x: \sigma) \vdash^w e': t'}_{(\theta_3 \theta_2 \theta_1) A \vdash^w \text{ let } x:: \sigma; \ x = e \text{ in } e': t'} \text{ Let}$

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: $\forall as.t_a \leq \forall bs.t_b \text{ iff } \forall bs . \exists as . t_a = t_b$
- Matching algorithm $\forall as.t_a \stackrel{\theta}{\leq} \forall bs.t_b$ defined as: find the smallest θ such that $\theta(\forall as.t_a) \leq \theta(\forall bs.t_b)$
- Isn't $t_a \stackrel{\theta}{\sim} t_b$ sufficient? No, in addition:
 - θ must not touch bs (dom(θ) \cap bs = \emptyset)
 - θ must not let bs escape (fv($\theta(\forall as.t_a)$) \cap bs = \emptyset and fv($\theta(\forall bs.t_b)$) \cap bs = \emptyset)
- 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 $t_a \stackrel{\theta}{\sim} \phi t_b$

Summary

- The Hindley/Milner stratification:
 - Types (including variables)
 - Type schemes = types with universal quantifiers
- Two similar formal systems
 - Logical proof of type correctness: A + 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!!!