Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Constraint-based Type Error Diagnosis (Tutorial)

Jurriaan Hage

Department of Information and Computing Sciences, Universiteit Utrecht
J.Hage@uu.nl

February 10, 2017

About me

- Assistant professor in Utrecht, Software Technology
- Topics of interest:
  - Static analysis of functional languages
    - Non-standard/type and effect systems
  - On and off: program plagiarism detection, object-sensitive analysis, soft typing of dynamic languages, and switching classes
  - PhD students active in legacy system modernization, and testing
  - Type error diagnosis (for functional languages/EDSLs)

Credits

The following people have contributed to this talk:

- Alejandro Serrano Mena, current PhD student
- Bastiaan Heeren, PhD student between 2000-2004
- Patrick Bahr, visiting postdoc in 2014
- Atze Dijkstra, implementor of UHC
- Many master students
- Many people contributed to Helium

I. Introduction and Motivation

Static type systems

- Statically typed languages come equiped with an intrinsic type system, preventing some structurally correct programs from being compiled
- "well-typed programs can't go wrong"
- type incorrect programs $\Rightarrow$ the need for diagnosis
- When type checking we typically assume various simple local properties to have been checked:
  - syntactic correctness
  - well-scopedness
  - definedness of variables
- Which properties it enforces, depends intimately on the language
  - Cf. does every function have the right number of arguments in C vs. Haskell

What is type error diagnosis?

- Type error diagnosis is the problem of communicating to the programmer that and/or why a program is not type correct
- This may involve information
  - that a program is type incorrect
  - which inconsistency was detected
  - which parts of the program contributed to the inconsistency
  - how the inconsistency may be fixed
- Traditionally, functional languages have more room for inconsistencies ⇒ at least some attention was paid to type error diagnosis

Languages follow Lehmann's sixth law

- Java has seen the introduction of parametric polymorphism (and type errors suffered)
- Java has seen the introduction of anonymous functions (I have not dared look)
- Languages like Scala embrace multiple paradigms
- Odersky's "type wall": unless complicated type system features are balanced by better diagnosis, programmers will flock to dynamic languages
- In terms of maintainability of (sizable) programs, dynamic languages do not seem to scale well
- New trends: dynamic languages becoming more static
- Again, diagnosis rears its ugly (time-consuming) head

Some simple Haskell

$$reverse = foldr\ (flip\ (:))\ [\,]$$
$$palindrome\ xs = reverse\ xs == xs$$

Is this program well typed?

Some simple Haskell

$$reverse = foldr\ (flip\ (:))\ [\,]$$
$$palindrome\ xs = reverse\ xs == xs$$

Is this program well typed?

```
Occurs check: cannot construct the infinite type: t ~ [[t]]
  Expected type: [t]
  Actual type: [[[t]]]
In the second argument of '(==)', namely 'xs'
In the expression: reverse xs == xs
```

What is wrong?


```
Occurs check: cannot construct the infinite type: t ~ [[t]]
  Expected type: [t]
  Actual type: [[[t]]]
In the second argument of '(==)', namely 'xs'
In the expression: reverse xs == xs
```

What is wrong?


```
Occurs check: cannot construct the infinite type: t ~ [[t]]
  Expected type: [t]
  Actual type: [[[t]]]
In the second argument of '(==)', namely 'xs'
In the expression: reverse xs == xs
```

- It does not point to the source of the error → not precise
- It's intimidating → not succint
- It shows an artifact of the implementation → mechanical
  - "Occurs check" is part of the unification algorithm
- Generally, message not very helpful
- Anyone know the likely fix?

What is wrong?


```
Occurs check: cannot construct the infinite type: t ~ [[t]]
  Expected type: [t]
  Actual type: [[[t]]]
In the second argument of '(==)', namely 'xs'
In the expression: reverse xs == xs
```

- It does not point to the source of the error → not precise
- It's intimidating → not succint
- It shows an artifact of the implementation → mechanical
  - "Occurs check" is part of the unification algorithm
- Generally, message not very helpful
- Anyone know the likely fix? *foldr* should be *foldl*

Unresolved top-level overloading

$$
\begin{aligned}
&xxxx = xs : [4,5,6] \\
&\quad \textbf{where}\ len = length\ xs \\
&\qquad xs = [1,2,3]
\end{aligned}
$$

Unresolved top-level overloading

$$
\begin{aligned}
&xxxx = xs : [4,5,6] \\
&\quad \textbf{where}\ len = length\ xs \\
&\qquad xs = [1,2,3]
\end{aligned}
$$

The Hugs message (GHC's message is just more verbose)

```
ERROR "Main.hs":1 - Unresolved top-level overloading
*** Binding             : xxxx
*** Outstanding context : (Num [b], Num b)
```

- Type classes make the type error message hard to understand
- The location of the mistake is rather vague
- No suggestions how to fix the program

Very old school parser combinators

$$\begin{array}{l} pExpr = pAndPrioExpr \\ \quad <\!|\!> sem\_Expr\_Lam \\ \qquad <\$\ pKey\ \texttt{"\textbackslash\textbackslash"} \\ \qquad <\!*\!>\ pFoldr1\ (sem\_LamIds\_Cons, sem\_LamIds\_Nil)\ pVarid \\ \qquad <\!*\!>\ pKey\ \texttt{"->"} \\ \qquad <\!*\!>\ pExpr \end{array}$$

gives

```
ERROR "BigTypeError.hs":1 - Type error in application
*** Expression     : sem_Expr_Lam <$ pKey "\\" <*> pFoldr1 (sem_LamIds_Cons,sem_
LamIds_Nil) pVarid <*> pKey "->"
*** Term           : sem_Expr_Lam <$ pKey "\\" <*> pFoldr1 (sem_LamIds_Cons,sem_
LamIds_Nil) pVarid
*** Type           : [Token] -> [((Type -> Int -> [([Char],(Type,Int,Int))] -> I
nt -> Int -> [(Int,(Bool,Int))] -> (PP_Doc,Type,a,b,[c] -> [Level],[S] -> [S]))
-> Type -> d -> [([Char],(Type,Int,Int))] -> Int -> Int -> e -> (PP_Doc,Type,a,b
,f -> f,[S] -> [S]),[Token])]
*** Does not match : [Token] -> [([Char] -> Type -> d -> [([Char],(Type,Int,Int)
)] -> Int -> Int -> e -> (PP_Doc,Type,a,b,f -> f,[S] -> [S]),[Token])]
```

Order is arbitrary (in Hugs)

$$yyyy :: (Bool \text{ -> } a) \text{ -> } (a, a, a)$$
$$yyyy = \backslash f \text{ -> } (f\ True, f\ False, f\ [\,])$$

What's wrong with this program?

Order is arbitrary (in Hugs)

$$yyyy :: (Bool \rightarrow a) \rightarrow (a, a, a)$$
$$yyyy = \backslash f \rightarrow (f\ True, f\ False, f\ [\,])$$

What's wrong with this program?

```
ERROR "Main.hs":2 - Type error in application
*** Expression     : f False
*** Term           : False
*** Type           : Bool
*** Does not match : [a]
```

- There is a lot of evidence that `f False` is well typed
- The type signature is not taken into account
- The type inference process suffers from (**right-to-left**) bias

Order is arbitrary (in GHC)

$$zzzz = \backslash f \rightarrow (f\ [\,], f\ \mathit{True}, f\ \mathit{False})$$

```
Ov.hs:8:23:
    Couldn't match expected type '[t2]' with actual type 'Bool'
    Relevant bindings include
      f :: [t2] -> t (bound at Ov.hs:8:9)
      zzzz :: ([t2] -> t) -> (t, t, t) (bound at Ov.hs:8:1)
    In the first argument of 'f', namely 'True'
    In the expression: f True
```

- No signature to take into account
- Both $f\ \mathit{True}$ and $f\ \mathit{False}$ are found to be in error
- The type inference process suffers from (**left-to-right**) bias

Good Error Reporting Manifesto

From Improved Type Error Reporting by Yang, Trinder and Wells

1. Correct detection and correct reporting
2. Precise: the smallest possible location
3. Succint: maximize useful and minimize non-useful info
4. Does not depend on implementation, i.e., amechanical
5. Source-based: not based on internal syntax
6. Unbiased
7. Comprehensive: enough to reason about the error

II. Constraint-based Type Inference

Hindley-Milner (intuitive summary)

- Consider the expression $\backslash\, x \mathrel{-\!\!>} x + 2$.
- Hindley-Milner will
  - introduce a fresh α for x
  - look at the body $x + 2$: unify the arguments of $+$ with their formal types (here all Int)
  - α becomes Int, and the whole expression has type $Int \mathrel{-\!\!>} Int$

Adding let-polymorphism to the mix

- Consider

  $$\textbf{let } y = \backslash z \text{ -> } z$$
  $$\textbf{in } \backslash x \text{ -> } y\ x + 2$$

- For z, α_1 is introduced, so that the body of y has type α_1
- Since α_1 does not show up in any other type (it is free) we may generalize over α_1 so that $y :: \forall\ \beta\ .\ \beta \text{ -> } \beta$
- Visit the body, introducing α for x, and instantiating β in y to, say, α_2 to give $\alpha_2 \text{ -> } \alpha_2$
- Unifying α with α_2 will identify the two, (arbitrarily) leading to $x :: \alpha$ and the instance of $y :: \alpha \text{ -> } \alpha$
- Then we perform the unifications of the previous slide

The polymorphic lamdba-calculus

$$\frac{\tau \prec \Gamma(x)}{\Gamma \ \vdash_{\text{HM}} \ x : \tau}$$

$$\frac{\Gamma \ \vdash_{\text{HM}} \ e_1 : \tau_1 \to \tau_2 \qquad\qquad \Gamma \ \vdash_{\text{HM}} \ e_2 : \tau_1}{\Gamma \ \vdash_{\text{HM}} \ e_1 \ e_2 : \tau_2}$$

$$\frac{\Gamma \backslash x \cup \{x{:}\tau_1\} \ \vdash_{\text{HM}} \ e : \tau_2}{\Gamma \ \vdash_{\text{HM}} \ \lambda x \to e : (\tau_1 \to \tau_2)}$$

$$\frac{\Gamma \ \vdash_{\text{HM}} \ e_1 : \tau_1 \qquad\qquad \Gamma \backslash x \cup \{x : \mathit{generalize}(\Gamma, \tau_1)\} \ \vdash_{\text{HM}} \ e_2 : \tau_2}{\Gamma \ \vdash_{\text{HM}} \ \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

- Algorithm $\mathcal{W}$ is a (deterministic) implementation of these typing rules.

Characteristics of Algorithm $\mathcal{W}$

- Can infer most general types for the let-polymorphic lambda-calculus
- Can deal with user-provided type information
- For extensions like higher-ranked types, type signatures must be provided
- Binding group analysis may need to be performed (always messy)
- Minor disadvantage: let-polymorphism does not integrate that well with some advanced type system features.
- Major disadvantage: algorithmic bias

What bias?

- Unifications are performed in a fixed order
- Order may be changed: many alternative implementations of HM exist
- Order of unification is unimportant for the resulting types,
- but it is important if you blame the first unification that is inconsistent with the foregoing.

How to cope

1. Investigate families of implementations (=solving orders) algorithm W, M, G, H,...
   - But which one to use when?

How to cope

1. Investigate families of implementations (=solving orders) algorithm W, M, G, H,...
   - But which one to use when?
2. Take a constraint-based approach, separating the unifications (=constraints) from the order in which they are solved.
   - generate and collect the constraints that describe the unifications that were to be performed, e.g., $\alpha == Int$
   - choose the order to solve them in some way that may be determined by the programmer, or by the program
   - Or even better: consider constraints a set at the time to identify situations that are known to often cause mistakes and suggest fixes

Constraint-based type inference

- Popular approach (see Pottier et al., Wells et al., OutsideIn(X), Pavlinovic et al.)
- A basic operation for type inference is unification. Property: let S be *unify*(τ_1, τ_2), then $S\tau_1 = S\tau_2$

We can view unification of two types as a constraint.

Constraint-based type inference

- Popular approach (see Pottier et al., Wells et al., OutsideIn(X), Pavlinovic et al.)
- A basic operation for type inference is unification.
  Property: let S be *unify*(τ_1, τ_2), then $S\tau_1 = S\tau_2$

We can view unification of two types as a constraint.

- An equality constraint imposes two types to be equivalent.
  Syntax: $\tau_1 \equiv \tau_2$
- We define satisfaction of an equality constraint as follows.
  $\mathcal{S}$ satisfies $(\tau_1 \equiv \tau_2) \quad =_{\mathsf{def}} \quad \mathcal{S}\tau_1 = \mathcal{S}\tau_2$
- Example:
  - $[\tau_1 := \mathit{Int}, \tau_2 := \mathit{Int}]$ satisfies $\tau_1 \to \tau_1 \equiv \tau_2 \to \mathit{Int}$

Bottom-up typing rules

$$\{x : \beta\},\ \emptyset\ \vdash_{\mathrm{BU}}\ x : \beta \qquad [\textsc{Var}]_{\mathrm{BU}}$$

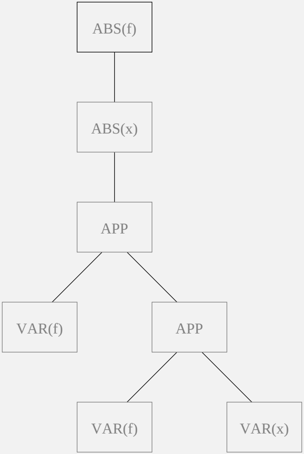
$$\frac{\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\mathrm{BU}}\ e_1 : \tau_1 \qquad \mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\mathrm{BU}}\ e_2 : \tau_2}{\mathcal{A}_1 \cup \mathcal{A}_2,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \to \beta\}\ \vdash_{\mathrm{BU}}\ e_1\ e_2 : \beta} \qquad [\textsc{App}]_{\mathrm{BU}}$$

$$\frac{\mathcal{A},\ \mathcal{C}\ \vdash_{\mathrm{BU}}\ e : \tau}{\mathcal{A}\backslash x,\ \mathcal{C} \cup \{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\}\ \vdash_{\mathrm{BU}}\ \lambda x \to e : (\beta \to \tau)} \qquad [\textsc{Abs}]_{\mathrm{BU}}$$

- A judgement ($\mathcal{A},\ \mathcal{C}\ \vdash_{\mathrm{BU}}\ e : \tau$) consists of the following.
  - $\mathcal{A}$: assumption set (contains assigned types for the free variables)
  - $\mathcal{C}$: constraint set
  - e: expression
  - τ: asssigned type (variable)

Example

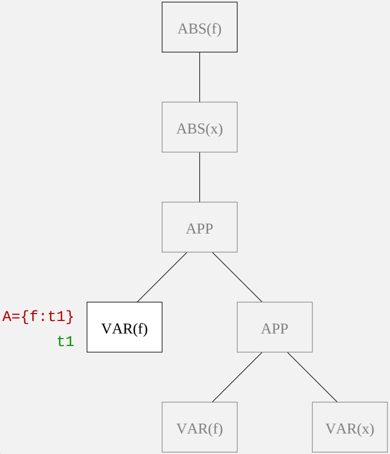
$$twice = \backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$$



Constraints

Example

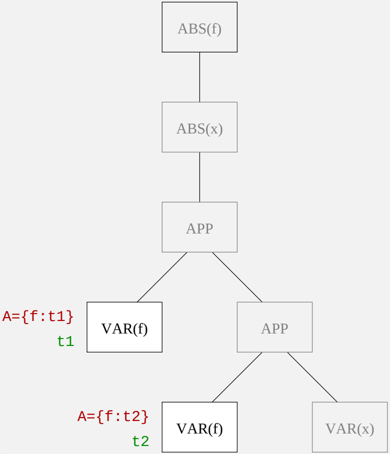
$$twice = \backslash f \;\text{->}\; \backslash x \;\text{->}\; f\ (f\ x)$$



Constraints

Example

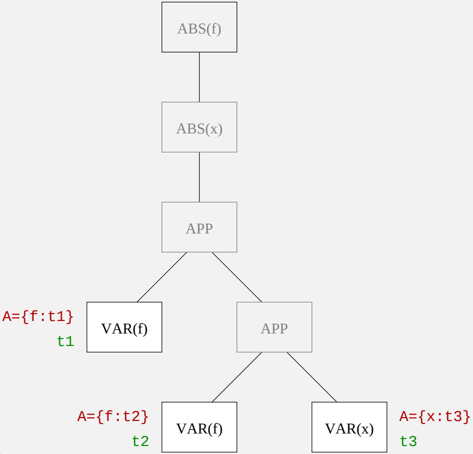
$$twice = \backslash f \;\text{->}\; \backslash x \;\text{->}\; f\ (f\ x)$$



Constraints

Example

$$twice = \backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$$



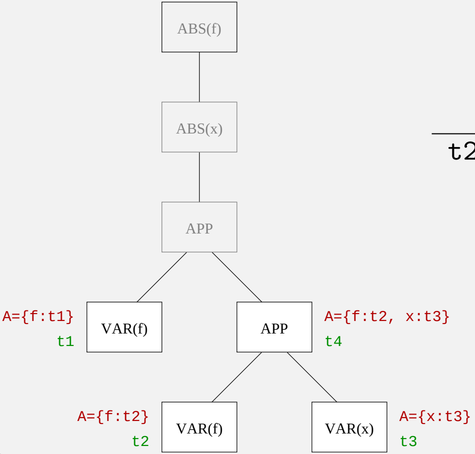
Constraints

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Example

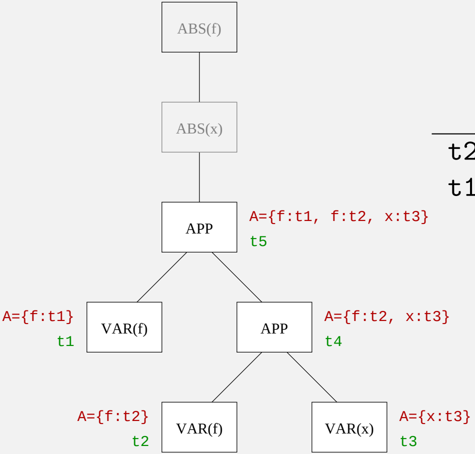
$$twice = \backslash f \text{ -> } \backslash x \text{ -> } f\ (f\ x)$$



Constraints		
`t2`	$\equiv$	`t3 -> t4`

Example

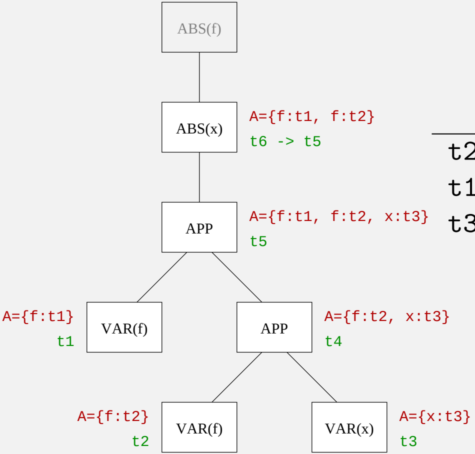
$$twice = \backslash f \;\text{->}\; \backslash x \;\text{->}\; f\ (f\ x)$$



Constraints		
`t2`	$\equiv$	`t3 -> t4`
`t1`	$\equiv$	`t4 -> t5`

Example

$$twice = \backslash f \; \texttt{->} \; \backslash x \; \texttt{->} \; f \; (f \; x)$$

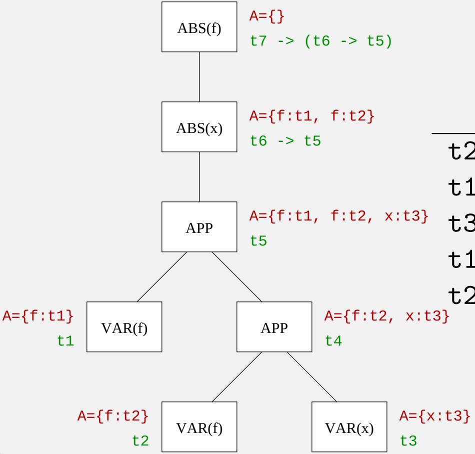


Constraints

t2	$\equiv$	t3 -> t4
t1	$\equiv$	t4 -> t5
t3	$\equiv$	t6

Example

$$twice = \backslash f \;\text{->}\; \backslash x \;\text{->}\; f\ (f\ x)$$



Constraints

t2	$\equiv$	t3 -> t4
t1	$\equiv$	t4 -> t5
t3	$\equiv$	t6
t1	$\equiv$	t7
t2	$\equiv$	t7

Example

$$twice = \backslash f \ \texttt{->} \ \backslash x \ \texttt{->} \ f\ (f\ x)$$

- $\mathcal{C} = \begin{cases} \texttt{t2} & \equiv & \texttt{t3 -> t4} \\ \texttt{t1} & \equiv & \texttt{t4 -> t5} \\ \texttt{t3} & \equiv & \texttt{t6} \\ \texttt{t1} & \equiv & \texttt{t7} \\ \texttt{t2} & \equiv & \texttt{t7} \end{cases}$
- $\mathcal{S} = \begin{cases} \texttt{t1,t2,t7} & := & \texttt{t6 -> t6} \\ \texttt{t3,t4,t5} & := & \texttt{t6} \end{cases}$
- $\mathcal{S}$ satisfies $\mathcal{C}$ (moreover, $\mathcal{S}$ is a minimal substitution that satisfies $\mathcal{C}$). As a result, we have inferred the type

$$\mathcal{S}(\texttt{t7 -> t6 -> t5}) = \texttt{(t6 -> t6) -> t6 -> t6}$$

for `twice`.

Constraints and polymorphism

- Syntax of an instance constraint:

$$\tau_1 \leqslant_M \tau$$

- Semantics with respect to a substitution $\mathcal{S}$:

$$\mathcal{S} \text{ satisfies } (\tau_1 \leqslant_M \tau_2) \quad =_{\mathsf{def}} \quad \mathcal{S}\tau_1 \prec \textit{generalize}(\mathcal{S}M, \mathcal{S}\tau_2)$$

- Example:
  - $[$`t1 := t2, t4 := t5 -> t5`$]$ satisfies `t4` $\leqslant_\emptyset$ `t1 -> t2`

Constraints and polymorphism

- Syntax of an instance constraint:

$$\tau_1 \leqslant_M \tau$$

- Semantics with respect to a substitution $\mathcal{S}$:

$$\mathcal{S} \text{ satisfies } (\tau_1 \leqslant_M \tau_2) \quad =_{\mathsf{def}} \quad \mathcal{S}\tau_1 \prec \mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau_2)$$

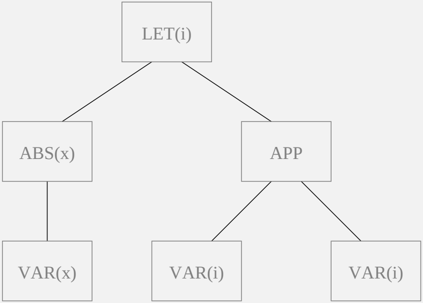
- Example:
  - $[$`t1 := t2, t4 := t5 -> t5`$]$ satisfies `t4` $\leqslant_\emptyset$ `t1 -> t2`

$$\frac{\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\text{BU}}\ e_1 : \tau_1 \qquad\qquad \mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\text{BU}}\ e_2 : \tau_2}{\begin{array}{c}\mathcal{A}_1 \cup \mathcal{A}_2 \backslash x,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \leqslant_M \tau_1 \mid x : \tau' \in \mathcal{A}_2\} \\ \vdash_{\text{BU}}\ \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2\end{array}} \quad [\textsc{Let}]_{\text{BU}}$$

Example

$$identity = \textbf{let}\ i = \backslash\ x \ {-}{>}\ x\ \textbf{in}\ i\ i$$

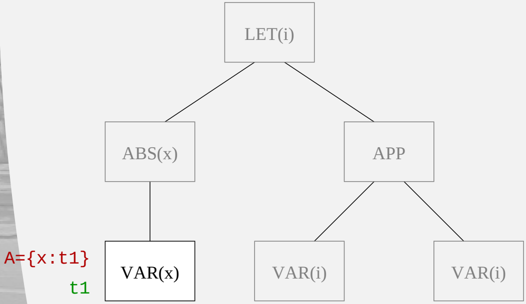
Constraints



Example

$$identity = \textbf{let}\ i = \backslash\ x \ {-}{>}\ x\ \textbf{in}\ i\ i$$

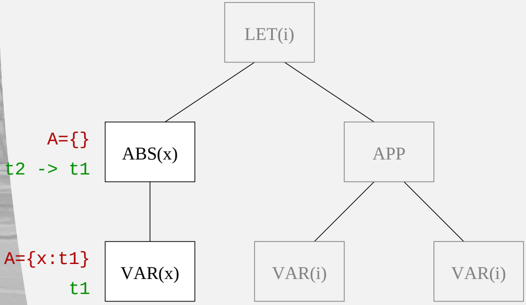
Constraints



Example

$$identity = \textbf{let}\ i = \backslash\ x \ {-}{>}\ x\ \textbf{in}\ i\ i$$

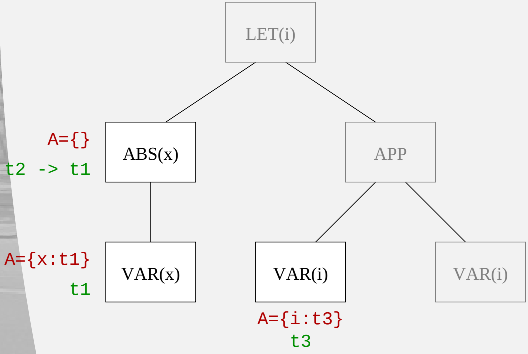
Constraints		
`t1`	$\equiv$	`t2`



Example

$$identity = \textbf{let}\ i = \backslash\, x \rightarrow x\ \textbf{in}\ i\ i$$

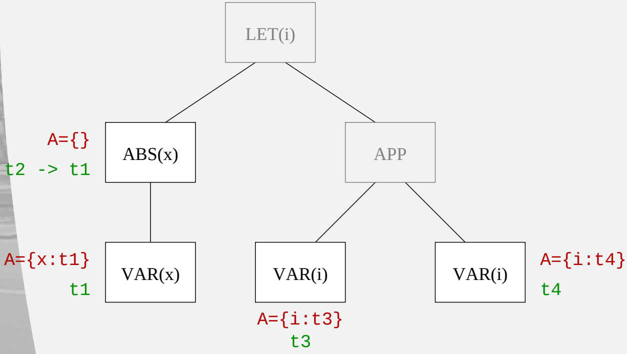
Constraints		
t1	$\equiv$	t2



Example

$$identity = \textbf{let}\ i = \backslash\ x \ \texttt{->}\ x\ \textbf{in}\ i\ i$$

Constraints		
t1	≡	t2

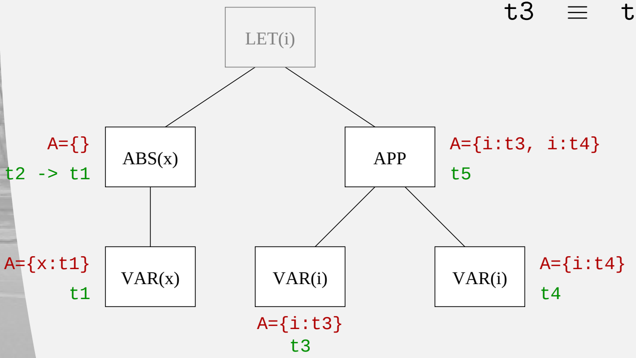


Example

$$identity = \textbf{let}\ i = \backslash\, x \ \text{->}\ x\ \textbf{in}\ i\ i$$

Constraints

t1	$\equiv$	t2
t3	$\equiv$	t4 -> t5

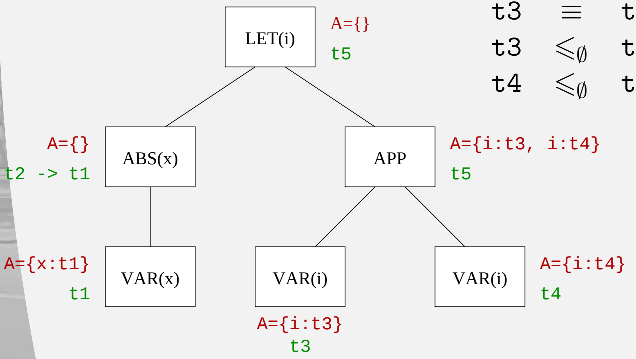


Example

$$identity = \mathbf{let}\ i = \backslash\, x \rightarrow x\ \mathbf{in}\ i\ i$$

Constraints

t1	$\equiv$	t2
t3	$\equiv$	t4 -> t5
t3	$\leqslant_\emptyset$	t2 -> t1
t4	$\leqslant_\emptyset$	t2 -> t1



Example

$$identity = \textbf{let}\ i = \backslash\ x \ \texttt{->}\ x\ \textbf{in}\ i\ i$$

- $\mathcal{C} = \begin{cases} \texttt{t1} & \equiv & \texttt{t2} \\ \texttt{t3} & \equiv & \texttt{t4 -> t5} \\ \texttt{t3} & \leqslant_{\emptyset} & \texttt{t2 -> t1} \\ \texttt{t4} & \leqslant_{\emptyset} & \texttt{t2 -> t1} \end{cases}$
- $\mathcal{S} = \begin{cases} \texttt{t1} & := & \texttt{t2} \\ \texttt{t3} & := & \texttt{(t6 -> t6) -> t6 -> t6} \\ \texttt{t4,t5} & := & \texttt{t6 -> t6} \end{cases}$
- $\mathcal{S}$ satisfies $\mathcal{C}$ (moreover, $\mathcal{S}$ is a minimal substitution that satisfies $\mathcal{C}$). As a result, we have inferred the type

$$\mathcal{S}(\texttt{t5}) = \texttt{t6 -> t6}$$

for `identity`.

III. Type Inferencing in Helium

The Helium compiler

- Constraint based approach to type inferencing
- Implements many heuristics, multiple solvers
- Existing algorithms/implementations can be emulated
- ```
  cabal install helium
  cabal install lvmrun
  ```
- Only: Haskell 98 minus type class and instance definitions
- And bias still exists from early binding groups to later ones
  - Others have addressed this issue
  ```

The Helium compiler

- Constraint based approach to type inferencing
- Implements many heuristics, multiple solvers
- Existing algorithms/implementations can be emulated
- ```
  cabal install helium
  cabal install lvmrun
  ```
- Only: Haskell 98 minus type class and instance definitions
- And bias still exists from early binding groups to later ones
  - Others have addressed this issue
- Supports domain specific type error diagnosis
- Details of the type rules: see Bastiaan Heeren's PhD
  ```

Some important compiler flags

- `--overloading` and `--no-overloading`
- `--enable-logging`, `--host` and `--port`
- `--algorithm-w` and `--algorithm-m`
- `--experimental` gives many more flags
  - `--kind-inferencing`
  - `--select-cnr` to select a particular constraint for blame
  - flags for choosing a particular solver
  - many other treewalks for ordering constraints

Constraints generated by Helium

For the program,

$$allinc = \backslash\, xs \;{-}{>}\; map\ (+1)\ xs$$

Helium generates ($-d$ option)

```
v5 := Inst(forall a b. (a -> b) -> [a] -> [b])
v9 := Inst(forall a. Num a => a -> a -> a)
Int == v10   : {literal}
v9 == v8 -> v10 -> v7   : {infix application}
v8 -> v7 == v6   : {left section}
v3 == v11   : {variable}
v5 == v6 -> v11 -> v4   : {application}
v3 -> v4 == v2   : {lambda abstraction}
v2 == v0   : {right-hand side}
v0 == v1   : {right hand side}
s22 := Gen([], v1)   : {Generalize allinc}
```

Greedy constraint solver

Given a set of type constraints, the greedy constraint solver returns a substitution that satisfies these constraints, and a list of constraint that could not be satisfied by the solver. The latter is used to produce type error messages.

- Advantages:
  - Efficient and fast
  - Straightforward implementation
- Disadvantage:
  - The order of the type constraints strongly influences the reported error messages. The type inference process is biased.

Ordering type constraints

- One is free to choose the order in which the constraints should be considered by the greedy constraint solver. (Although there is a restriction for an implicit instance constraint)
- Instead of returning a list of constraints, return a constraint tree that follows the shape of the AST.
- A tree-walk flattens the constraint tree and orders the constraints.
  - $\mathcal{W}$: almost a post-order tree walk
  - $\mathcal{M}$: almost a pre-order tree walk
  - Bottom-up: ...
  - Pushing down type signatures: ...

A realistic type rule

- Some constraints 'belong' to certain subexpressions:

$$
\begin{array}{c}
\mathcal{T}_{\mathcal{C}} = [c_2, c_3] \mathbin{\underline{\Diamond}} \{\!\!\{ c_1 \triangledown \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2}, \mathcal{T}_{\mathcal{C}3} \}\!\!\} \\
c_1 = (\tau_1 \equiv \mathit{Bool}) \quad c_2 = (\tau_2 \equiv \beta) \quad c_3 = (\tau_3 \equiv \beta) \\
\mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash e_1 : \tau_1 \\
\mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash e_2 : \tau_2 \qquad \mathcal{A}_3, \mathcal{T}_{\mathcal{C}3} \vdash e_3 : \tau_3 \\
\hline
\mathcal{A}_1 + \!\!\! + \mathcal{A}_2 + \!\!\! + \mathcal{A}_3, \mathcal{T}_{\mathcal{C}} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \beta
\end{array}
$$

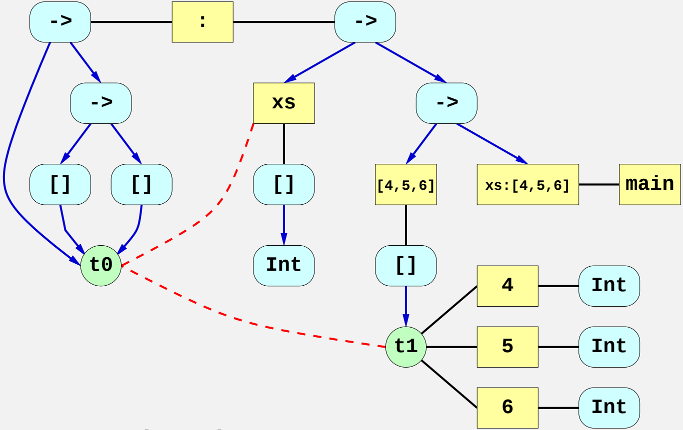
- c_1 is generated by the conditional, but associated with the boolean subexpression.
- Example strategy: left-to-right, bottom-up for then and else part, push down Bool (do c_1 before $\mathcal{T}_{\mathcal{C}1}$).

Global constraint solver

Uses type graphs allow us to solve the collected type constraints in a more global way. These can represent inconsistent sets of constraints.

- Advantages:
  - Global properties can be detected
  - A lot of information is available
  - The type inference process can be unbiased
  - It is easy to include new heuristics to spot common mistakes.
- Disadvantage:
  - Extra overhead makes this solver a bit slower
  - But: only for the first inconsistent binding group!

Type graphs (for $xs : [4,5,6]$)



$$
\begin{aligned}
&main = xs : [4,5,6] \\
&\quad \textbf{where}\ len = length\ xs \\
&\qquad xs = [1,2,3]
\end{aligned}
$$

Type graph heuristics

If a type graph contains an inconsistency, then heuristics help to choose which location is reported as type incorrect.

- Examples:
  - minimal number of type errors
  - count occurrences of clashing type constants ($3\times \mathit{Int}$ versus $1\times \mathit{Bool}$)
  - reporting an expression as type incorrect is preferred over reporting a pattern
  - wrong literal constant (4 versus 4.0)
  - not enough arguments are supplied for a function application
  - permute the elements of a tuple
  - (:) is used instead of (++)

Heuristics in Helium


```
listOfHeuristics options siblings path =
  ...
  [avoidForbiddenConstraints   -- remove constraints that should NEVER be reporte
  , highParticipation 0.95 path
  , phaseFilter                -- phasing from the type inference directives
  ] ++
  [Heuristic (Voting (
    [siblingFunctions siblings
    , siblingLiterals
    , applicationHeuristic
    , variableFunction    -- ApplicationHeuristic without application
    , tupleHeuristic      -- ApplicationHeuristic for tuples
    , fbHasTooManyArguments
    , constraintFromUser path   -- From .type files
    , unaryMinus (Overloading `elem` options)
    ] ++
    [similarNegation | Overloading `notElem` options] ++
    [unifierVertex | UnifierHeuristics `elem` options]))] ++
  [inPredicatePath | Overloading `elem` options] ++
  [avoidApplicationConstraints, avoidNegationConstraints
  , avoidTrustedConstraints, avoidFolkloreConstraints
  , firstComeFirstBlamed   -- Will delete all except the first
  ]
```

The Helium message

$$\begin{aligned} &main = xs : [4,5,6] \\ &\quad \textbf{where } len = length\ xs \\ &\qquad xs = [1,2,3] \end{aligned}$$

```
(2,9): Warning: Definition "len" is not used
(1,11): Type error in constructor
 expression         : :
   type             : a     -> [a  ] -> [a]
   expected type    : [Int] -> [Int] -> b
 probable fix       : use ++ instead
```

Example: permute function arguments

$$test :: Parser\ Char\ String$$
$$test = option\ \texttt{""}\ (token\ \texttt{"hello!"})$$

In Helium:

```
(2,8): Type error in application
 expression        : option "" (token "hello!")
 term              : option
   type            : Parser a b -> b -> Parser a b
   does not match  : String -> Parser Char String -> c
 probable fix      : flip the arguments
```

Limitations of Helium

- The Helium language is relatively small
- A major limitation of the type inference process: consistent binding groups are never blamed.

$$
\begin{aligned}
&myfold\ f\ z\ [\,] = [z] \\
&myfold\ f\ z\ (x : xs) = myfold\ f\ (f\ z\ x)\ xs \\
&rev = myfold\ (flip\ (:))\ [\,] \\
&palin :: Eq\ a \Rightarrow [a] \rightarrow Bool \\
&palin\ xs = rev\ xs == xs
\end{aligned}
$$

- Helium blames $palin$, some other systems can blame $myfold$ instead. Signatures for rev and $myfold$ improve Helium's message.
- Note: we use our intuition of what rev and $palin$ do, a compiler (typically) cannot.

Summary

We have described a *parametric* type inferencer

- Constraint-based: specification and implementation are separated
- Standard algorithms can be simulated by choosing an order for the constraints
- Two implementations are available to solve the constraints
- Type graph heuristics help in reporting the most likely mistake