Type Error Diagnosis in Helium

Jurriaan Hage
joint work with Bastiaan Heeren (slides are mostly his too)

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

September 16, 2013

Some things about me

- PhD at Leiden under Grzegorz Rozenberg on algorithms and combinatorics of graphs and groups (switching classes)
- Commercial educator at Leiden during 1999-2000
- Software technology with Doaitse Swierstra and Johan Jeuring since Nov 2000.

Some things about me

- PhD at Leiden under Grzegorz Rozenberg on algorithms and combinatorics of graphs and groups (switching classes)
- Commercial educator at Leiden during 1999-2000
- Software technology with Doaitse Swierstra and Johan Jeuring since Nov 2000.
- They (and Wouter Swierstra) all say “Hi!”.

My current projects

- Type and effect systems
  - PhD on higher-ranked polyvariance
- Continuous testing of Internet applications
  - Flash, in our case
- Automated support for migration of Cobol/JCL legacy systems to service architecture
- Hobby: plagiarism detection for C#, Java and Haskell
- Also low key: object-sensitive analysis of PHP, soft typing of dynamic languages.
- Type error diagnosis for functional languages/EDSLs
  - Proposal currently under appraisal

Blatant advertisement

- Next year, I chair
  - PEPM 2014, San Diego, co-located with POPL
  - TFP 2014, somewhere in the Netherlands
- Start saving up papers to submit!

1. The Helium Type Inferencer

Part I: Constraint-based type inference

- Introduction (includes time travel)
- Bottom-up typing rules
- Equality constraints
- Polymorphism and instance constraints
- Constraint solving
- Summary

Example 1

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

Is this program well typed?

Example 1

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

Is this program well typed?

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

Student FP: "What did I do wrong?"

- 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

Example 2

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

Is this program well typed?

Example 2


```
pExpr = pAndPrioExpr
    <|> sem_Expr_Lam
        <$ pKey "\\"
        <*> pFoldr1 (sem_LamIds_Cons, sem_LamIds_Nil) pVarid
        <*> pKey "->" <*> pExpr
```

Is this program well typed?

```
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])]
```

Example 2


```
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])]
```

Student: "Why is my parser not accepted by the compiler?"

- Message is really big, and thus not very helpful
- You have to discover why the types don't match yourself
- It happens to be a common mistake, and easy to fix

Example 3

$$main :: (Bool \to a) \to (a, a, a)$$
$$main = \lambda f \to (f\ True, f\ False, f\ [\,])$$

Is this program well typed?

Example 3

$$main :: (Bool \to a) \to (a, a, a)$$
$$main = \lambda f \to (f\ True, f\ False, f\ [\,])$$

Is this program well typed?

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

Student Type Systems: "Why is `f False` reported?"

- 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 a *left-to-right* bias

What is Helium?

- Original idea by Arjan van IJzendoorn
- Haskell 98 without class and instance definitions
- Particular attention paid to type error diagnosis
  - Mostly based on Bastiaan Heeren's PhD thesis
- Maintained by Bastiaan Heeren and myself
- Has been dormant for some time, but is now being readied for Hackage
  - A few parts have already made it unto Hackage
- More (sometimes outdated) details on the Helium website
  - `http://www.cs.uu.nl/wiki/bin/view/Helium/WebHome`
- At the basis of the Helium innovations lies a constraint based type inference process.

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

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

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

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

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

Constraint-based type inference

- 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

- 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.
  $\mathcal{S} \text{ satisfies } (\tau_1 \equiv \tau_2) \quad =_{\mathsf{def}} \quad \mathcal{S}\tau_1 = \mathcal{S}\tau_2$
- 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

$$\{x : \beta\},\ \emptyset\ \vdash_{\text{BU}}\ x : \beta \qquad [\text{VAR}]_{\text{BU}}$$

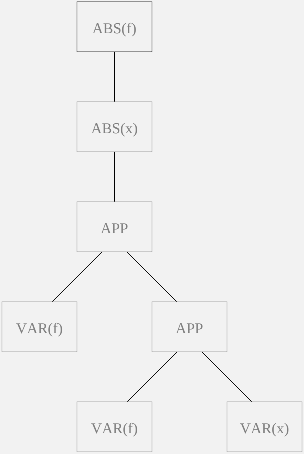
$$\frac{\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\text{BU}}\ e_1 : \tau_1 \qquad \mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\text{BU}}\ e_2 : \tau_2}{\mathcal{A}_1 \cup \mathcal{A}_2,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \to \beta\}\ \vdash_{\text{BU}}\ e_1\ e_2 : \beta} \qquad [\text{APP}]_{\text{BU}}$$

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

- A judgement ($\mathcal{A},\ \mathcal{C}\ \vdash_{\text{BU}}\ e : \tau$) 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

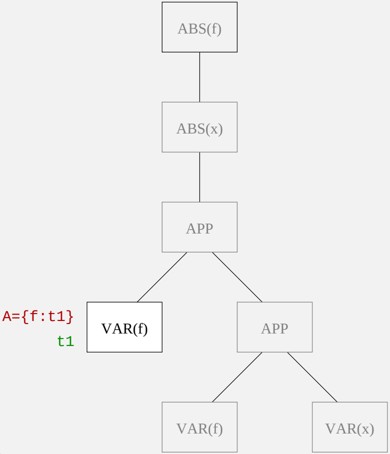
$$twice = \lambda f \rightarrow \lambda x \rightarrow f\ (f\ x)$$



Constraints

Example

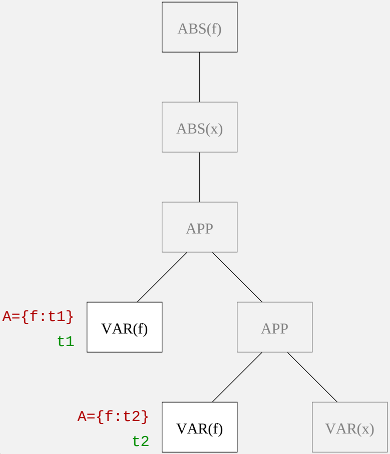
$$twice = \lambda f \rightarrow \lambda x \rightarrow f\ (f\ x)$$



Constraints

Example

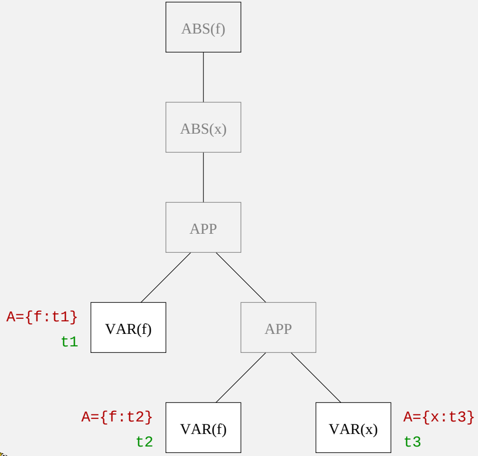
$$twice = \lambda f \rightarrow \lambda x \rightarrow f\ (f\ x)$$



Constraints

Example

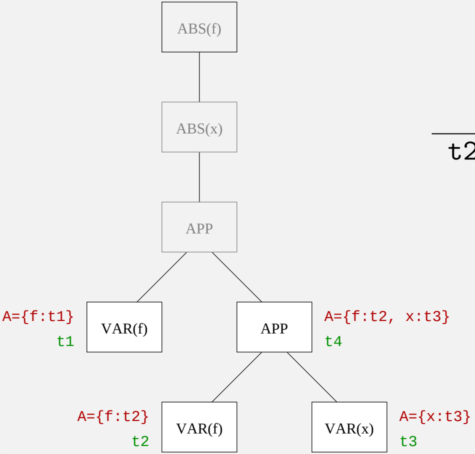
$$twice = \lambda f \rightarrow \lambda x \rightarrow f\ (f\ x)$$



Constraints

Example

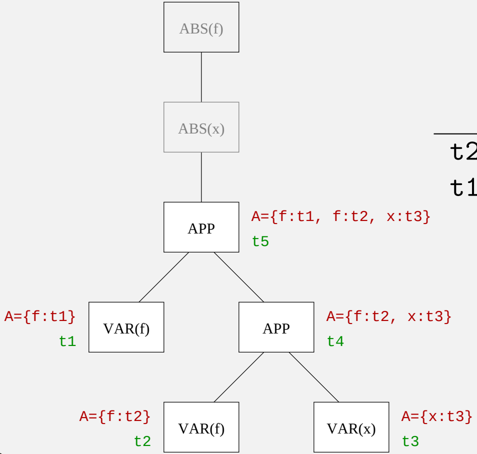
$$twice = \lambda f \rightarrow \lambda x \rightarrow f\ (f\ x)$$



Constraints
t2 $\equiv$ t3 -> t4

Example

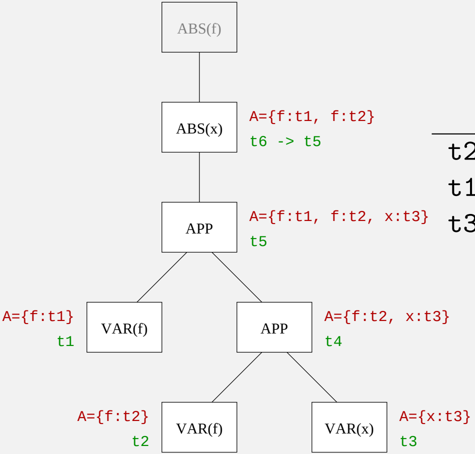
$$twice = \lambda f \to \lambda x \to f\ (f\ x)$$



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

Example

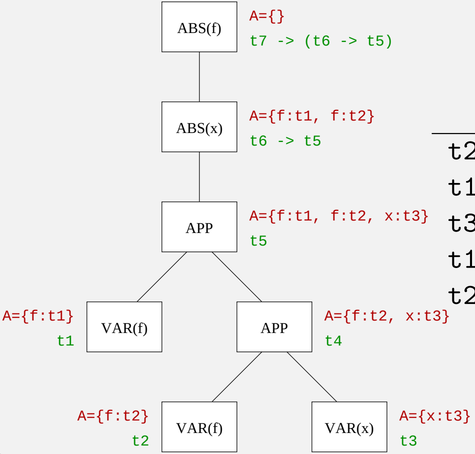
$$twice = \lambda f \to \lambda x \to f\ (f\ x)$$



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

Example

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



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

Example

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

- $\mathcal{C} = \begin{cases} \texttt{t2} & \equiv & \texttt{t3 -> t4} \\ \texttt{t1} & \equiv & \texttt{t4 -> t5} \\ \texttt{t3} & \equiv & \texttt{t6} \\ \texttt{t1} & \equiv & \texttt{t7} \\ \texttt{t2} & \equiv & \texttt{t7} \end{cases}$
- $\mathcal{S} = \begin{cases} \texttt{t1,t2,t7} & := & \texttt{t6 -> t6} \\ \texttt{t3,t4,t5} & := & \texttt{t6} \end{cases}$
- $\mathcal{S}$ 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 $\mathcal{S}$:

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

- 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 $\mathcal{S}$:

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

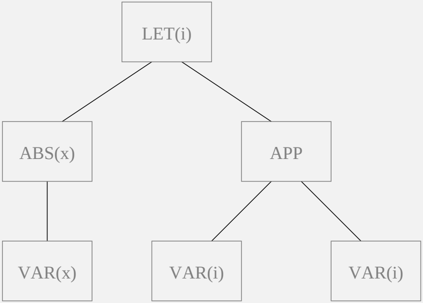
- Example:
  - $[$`t1 := t2, t4 := t5 -> t5`$]$ satisfies `t4` $\leqslant_\emptyset$ `t1 -> t2`

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

Example

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

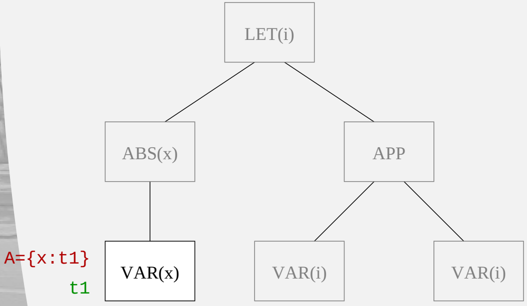
Constraints



Example

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

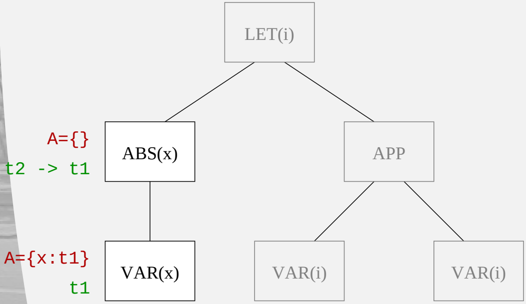
Constraints



Example

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

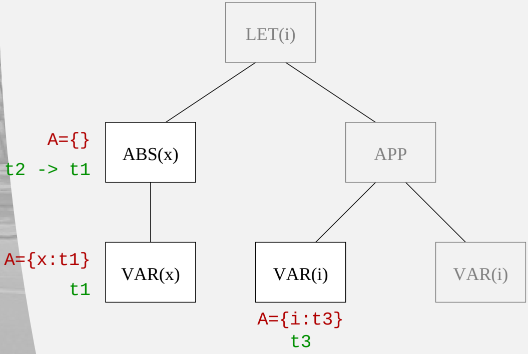
Constraints		
t1	≡	t2



Example

$$identity = \textbf{let } i = \lambda x \rightarrow x \textbf{ in } i \; i$$

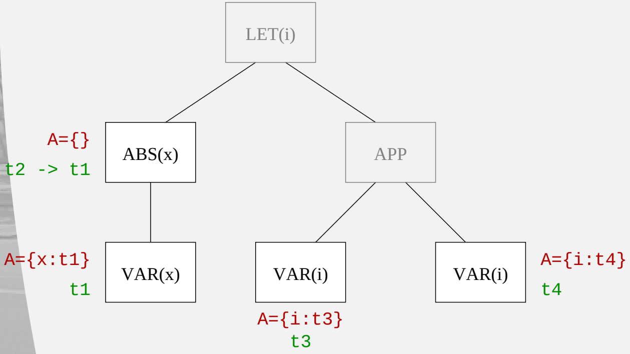
Constraints		
t1	≡	t2



Example

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

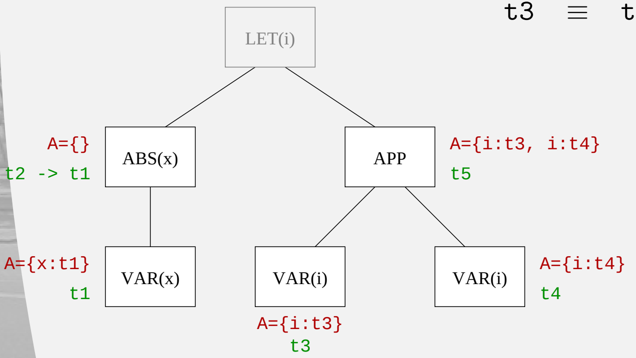
Constraints		
t1	≡	t2



Example

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

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

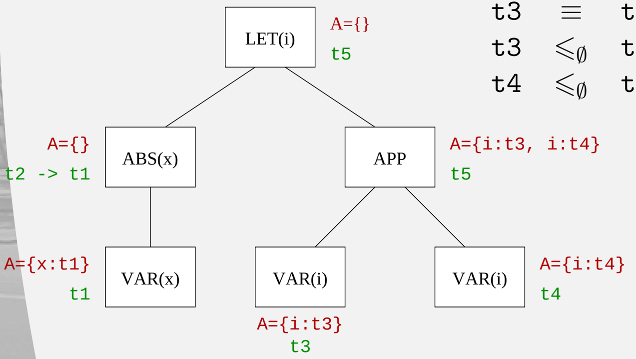


Example

$$\mathit{identity} = \mathbf{let}\ i = \lambda x \rightarrow x\ \mathbf{in}\ i\ i$$

Constraints

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



Example

$$identity = \mathbf{let}\ i = \lambda x \to x\ \mathbf{in}\ i\ i$$

- $\mathcal{C} = \begin{cases} \texttt{t1} & \equiv & \texttt{t2} \\ \texttt{t3} & \equiv & \texttt{t4 -> t5} \\ \texttt{t3} & \leqslant_\emptyset & \texttt{t2 -> t1} \\ \texttt{t4} & \leqslant_\emptyset & \texttt{t2 -> t1} \end{cases}$
- $\mathcal{S} = \begin{cases} \texttt{t1} & := & \texttt{t2} \\ \texttt{t3} & := & \texttt{(t6 -> t6) -> t6 -> t6} \\ \texttt{t4,t5} & := & \texttt{t6 -> t6} \end{cases}$
- $\mathcal{S}$ 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`.

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:

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

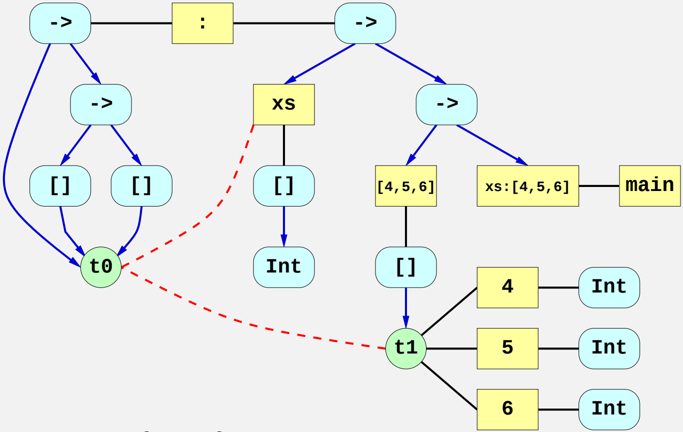
- c_1 is generated by the conditional, but associated with the boolean subexpression.
- Example strategy: left-to-right, bottom-up for then and else part, push down Bool (do c_1 before $\mathcal{T}_{\mathcal{C}1}$).

Global constraint solver

Type graphs allow us to solve the collected type constraints in a more global way.

- 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 slower

Type graphs



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

Type graph heuristics

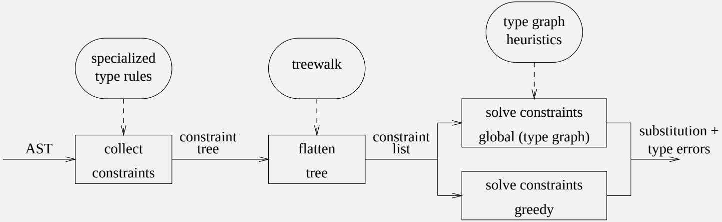
If a type graph contains an inconsistency, then heuristics help to choose which location is reported as type incorrect.

- 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 (++)
- All these heuristics are present in the Helium compiler
- We will see more examples in Part II

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



2. Domain Specific Type Error Diagnosis

Embedded Domain Specific Languages

- Embedded (internal) Domain Specific Languages are achieved by encoding the DSL syntax inside that of a host language.
- Many "advantages":
  - familiarity host language syntax
  - escape hatch to the host language
  - existing libraries, compilers, IDE's, etc.
  - combining EDSLs
- At the very least, useful for prototyping DSLs

What host language?

- Some languages provide extensibility as part of their design, e.g., Ruby, Python, Scheme
- Others are rich enough to encode a DSL with relative ease, e.g., Haskell, C++
- In most languages we just have to make do
- In Haskell, EDSLs are simply libraries that provide some form of "fluency"
  - Consisting of domain terms and types, and special operators with particular priority and fixity

Challenges for EDSLs

- How to achieve:
  - domain specific optimisations
  - domain specific error diagnosis
- Optimisations and error diagnosis also take up time in a non-embedded setting, but there we have more control.
- Can we attain this control for error diagnosis?

Our case study

- Parser combinators: an EDSL for describing parsers
- An executable and extensible form of EBNF
  - Concatenation/juxtaposition: $p \mathbin{<\!\!*\!\!>} q$, and $p \mathbin{<\!\!*} q$
  - Choice: $p \mathbin{<\!|\!>} q$
  - Semantics: $f \mathbin{<\!\$\!>} p$ and $f \mathbin{<\!\$} p$
  - Repetition: $many$, $many1$, ...
  - Optional: $option\ p$ **default**
  - Literals: $token$ `"text"`, $pKey$ `"->"`
  - Others introduced as needed, and defined at will

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

My first mistake

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

The error message that results:

```
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])]
```

A closer look at the message


```
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])]
```

- Message is large and looks complicated
- You have to discover why the types don't match yourself
- No mention of "parsers" in the error message
- It happens to be a common mistake, and easy to fix

The problems

Type error messages typically suffer from the following problems.

1. **A fixed type inference process.** The order in which types are inferred strongly influences the reported error site, and there is no way to depart from it.
2. **The size of the mentioned types.** Irrelevant parts are shown, and type synonyms are not always preserved.
3. **The standard format of type error messages.** Domain specific terms are not used.
4. **No anticipation for common mistakes.** Error messages focus on the problem, and not on how to fix it.

The solution in a nutshell

1. Bring the type inference mechanism under control
   - by phrasing the type inference process as a constraint solving problem
2. Provide hooks in the compiler's type inference process to change the process for certain classes of expressions
   - specialize type error messages for a particular domain
   - control the order in which constraints are solved
   - drive heuristics that suggest fixes for often-made mistakes

The solution in a nutshell

1. Bring the type inference mechanism under control
   - by phrasing the type inference process as a constraint solving problem
2. Provide hooks in the compiler's type inference process to change the process for certain classes of expressions
   - specialize type error messages for a particular domain
   - control the order in which constraints are solved
   - drive heuristics that suggest fixes for often-made mistakes

- Changing the type system is forbidden!
  - Only the order of solving, and the provided messages can be changed

How is this organised?

- For a given source module Abc.hs, a DSL designer may supply a file Abc.type containing the directives
- The directives are automatically used when the module is imported
- The compiler will adapt the type error mechanism based on these type inference directives.
- The directives themselves are also a(n external) DSL!

The type inference process

- We piggy-back ride on Haskell's underlying type system
- Type rules for functional languages are often phrased as a set of logical deduction rules
- Inference is then implemented by means of an AST traversal
  - Ad-hoc or using attribute grammars

The rule for type checking applications

$$\frac{\Gamma \vdash_{\text{HM}} f : \tau_a \to \tau_r \qquad \Gamma \vdash_{\text{HM}} e : \tau_a}{\Gamma \vdash_{\text{HM}} f\ e : \tau_r}$$

- Γ is an environment, containing the types of identifiers defined elsewhere
- Rules for variables, anonymous functions and local definitions omitted
- Algorithm $\mathcal{W}$ is a (deterministic) implementation of these typing rules.

Specializing a type rule (1/3)

Applying the type rule for function application twice in succession results in the following:

$$\frac{\Gamma \vdash_{\mathrm{HM}} op : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \qquad \Gamma \vdash_{\mathrm{HM}} x : \tau_1 \qquad \Gamma \vdash_{\mathrm{HM}} y : \tau_2}{\Gamma \vdash_{\mathrm{HM}} x \text{ `}op\text{` } y : \tau_3}$$

Specializing a type rule (1/3)

Applying the type rule for function application twice in succession results in the following:

$$\frac{\Gamma \ \vdash_{\mathrm{HM}} \ op : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \qquad \Gamma \ \vdash_{\mathrm{HM}} \ x : \tau_1 \qquad \Gamma \ \vdash_{\mathrm{HM}} \ y : \tau_2}{\Gamma \ \vdash_{\mathrm{HM}} \ x \ `op` \ y : \tau_3}$$

Consider one of the parser combinators, for instance $<\$>$.

$$<\$> :: (a \rightarrow b) \rightarrow \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ b$$

We can now create a specialized type rule by filling in this type in the type rule

Applying the type rule for function application twice in succession results in the following:

$$\frac{\Gamma \vdash_{\mathrm{HM}} op : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \qquad \Gamma \vdash_{\mathrm{HM}} x : \tau_1 \qquad \Gamma \vdash_{\mathrm{HM}} y : \tau_2}{\Gamma \vdash_{\mathrm{HM}} x \text{ `}op\text{` } y : \tau_3}$$

Consider one of the parser combinators, for instance $<\$>$.

$$<\$> :: (a \rightarrow b) \rightarrow \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ b$$

We can now create a specialized type rule by filling in this type in the type rule (x and y stand for arbitrary expressions of the given type)

$$\frac{\Gamma \vdash_{\mathrm{HM}} x : a \rightarrow b \qquad \Gamma \vdash_{\mathrm{HM}} y : \mathit{Parser}\ s\ a}{\Gamma \vdash_{\mathrm{HM}} x <\$> y : \mathit{Parser}\ s\ b}$$

Specializing a type rule (2/3)

- Use equality constraints to make the restrictions that are imposed by the type rule explicit.
- Γ is unchanged, and therefore omitted from the rule
- Type rules are invalidated by shadowing, here, $<\$>$.

$$\frac{x : \tau_1 \qquad y : \tau_2}{x <\$> y : \tau_3} \qquad \left\{ \begin{array}{lll} \tau_1 & \equiv & a \to b \\ \tau_2 & \equiv & \textit{Parser}\ s\ a \\ \tau_3 & \equiv & \textit{Parser}\ s\ b \end{array} \right.$$

Specializing a type rule (2/3) §2

- Use equality constraints to make the restrictions that are imposed by the type rule explicit.
- Γ is unchanged, and therefore omitted from the rule
- Type rules are invalidated by shadowing, here, $<\$>$.

$$\frac{x : \tau_1 \qquad y : \tau_2}{x <\$> y : \tau_3} \qquad \left\{ \begin{array}{lll} \tau_1 & \equiv & a \rightarrow b \\ \tau_2 & \equiv & \mathit{Parser}\ s\ a \\ \tau_3 & \equiv & \mathit{Parser}\ s\ b \end{array} \right.$$

Split up the type constraints in "smaller" unification steps.

$$\frac{x : \tau_1 \qquad y : \tau_2}{x <\$> y : \tau_3} \qquad \left\{ \begin{array}{lllllll} \tau_1 & \equiv & a_1 \rightarrow b_1 & & s_1 & \equiv & s_2 \\ \tau_2 & \equiv & \mathit{Parser}\ s_1\ a_2 & & a_1 & \equiv & a_2 \\ \tau_3 & \equiv & \mathit{Parser}\ s_2\ b_2 & & b_1 & \equiv & b_2 \end{array} \right.$$

$$\frac{x : \tau_1 \qquad y : \tau_2}{x \mathbin{<\$>} y : \tau_3} \qquad \left\{ \begin{array}{lllllll} \tau_1 & \equiv & a_1 \to b_1 & & s_1 & \equiv & s_2 \\ \tau_2 & \equiv & \mathit{Parser}\ s_1\ a_2 & & a_1 & \equiv & a_2 \\ \tau_3 & \equiv & \mathit{Parser}\ s_2\ b_2 & & b_1 & \equiv & b_2 \end{array} \right.$$

```
  x :: t1;   y :: t2;
----------------------
    x <$> y :: t3;

t1 == a1 -> b1
t2 == Parser s1 a2
t3 == Parser s2 b2
s1 == s2
a1 == a2
b1 == b2
```

Special type error messages


```
  x :: t1;   y :: t2;
---------------------
    x <$> y :: t3;

t1 == a1 -> b1      : left operand is not a function
t2 == Parser s1 a2 : right operand is not a parser
t3 == Parser s2 b2 : result type is not a parser
s1 == s2 : parser has an incorrect symbol type
a1 == a2 : function cannot be applied to parser's result
b1 == b2 : parser has an incorrect result type
```

- Supply an error message for each type constraint. This message is reported if the corresponding constraint cannot be satisfied.

Example

$$test :: Parser\ Char\ String$$
$$test = map\ toUpper <\$> \texttt{"hello, world!"}$$

This results in the following type error message:

```
Type error: right operand is not a parser
```

Example

$$test :: Parser\ Char\ String$$
$$test = map\ toUpper <\$> \texttt{"hello, world!"}$$

This results in the following type error message:

```
Type error: right operand is not a parser
```

Important context specific information is missing, for instance:

- Inferred types for (sub-)expressions, and intermediate type variables
- Pretty printed expressions from the program
- Position and range information

Error message attributes

The error message attached to a type constraint might now look like:

```
  x :: t1;    y :: t2;
---------------------
    x <$> y :: t3;
...
t2 == Parser s1 a2 :
 @expr.pos@: The right operand of <$> should be a
  expression       : @expr.pp@             parser
  right operand    : @y.pp@
    type           : @t2@
    does not match : Parser @s1@ @a2@
...
```

Example

$test :: Parser\ Char\ String$
$test = map\ toUpper <\$>$ "hello, world!"

This results in the following type error message (including the inserted error message attributes):

```
(2,21): The right operand of <$> should be a parser
 expression          : map toUpper <$> "hello, world!"
 right operand       : "hello, world!"
   type              : String
   does not match    : Parser Char String
```

Implicit constraints

A type constraint can be "moved" from the constraint set to the deduction rule.

```
   x :: t1;    y :: t2;
----------------------------
  x <$> y :: Parser s b;

t1 == a1 -> b      : left operand is not a function
t2 == Parser s a2 : right operand is not a parser
a1 == a2 : function cannot be applied to parser's
                                              result
```

An implicit constraint with a default error message is inserted for the type in the conclusion.

Order of the type constraints

Each meta-variable represents a subtree for which also type constraints are collected. This constraint set can be explicitly mentioned in the type rule.

```
   x :: t1;    y :: t2;
----------------------------
  x <$> y :: Parser s b;

constraints x
t1 == a1 -> b      : left operand is not a function
constraints y
t2 == Parser s a2 : right operand is not a parser
a1 == a2 : function cannot be applied to parser's
                                            result
```

Soundness and completeness

The soundness of a specialized type rule with respect to the default type rules is examined at compile time.

- Because a mistake is easily made
- Invalid type rules are rejected when a Haskell file is compiled
- Type safety can still be guaranteed at run-time
- The type rule may not be too restrictive, so we are also complete
  - This restriction may be dropped

Example


```
  x :: t1;      y :: t2;
-----------------------------
   x <$> y :: Parser s b;

t1 == a1 -> b      : left operand is not a function
t2 == Parser s a2 : right operand is not a parser
```

This specialized type rule is not restrictive enough:

```
The type rule for "x <$> y" is not correct
  the type according to the type rule is
    (a -> b, Parser c d, Parser c b)
  whereas the standard type rules infer the type
    (a -> b, Parser c a, Parser c b)
```

Another example


```
   x :: a -> b;    y :: Parser Char a;
-------------------------------------
      x <$> y :: Parser Char b;
```

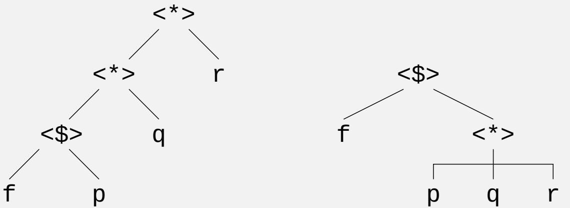
This specialized type rule is too restrictive: there is no reason to demand that we parse streams of characters.

```
The type rule for "x <$> y" is not correct
  the type according to the type rule is
    (a -> b, Parser Char a, Parser Char b)
  whereas the standard type rules infer the type
    (a -> b, Parser c a, Parser c b)
```

AST versus conceptual structure: phasing

$$f <\$> p <\ast> q <\ast> r$$

- Associativity and priorities of the operators chosen to minimize parentheses in a practical situation
- The inferencing process follows the shape of the abstract syntax tree closely
- Conceptual and actual AST shape may be very different



abstract syntax tree

conceptual structure

$$test :: Parser\ Char\ String$$
$$test = (+\!\!+) \mathbin{<\$>} token\ \texttt{"hello world"} \mathbin{<\!\!*\!\!>} symbol\ \texttt{'!'}$$

My four step approach to infer the types:

1. Infer the types of the expressions between the parser combinators.
2. Check if the types inferred for the parser subexpressions are indeed *Parser* types.
3. Verify that the parser types can agree upon a common symbol type.
4. Determine whether the result types of the parser fit the function.

In this case, a type inconsistency is detected in the fourth step.

- Hugs reports the following:

```
ERROR "Phase1.hs":4 - Type error in application
Expression: (++) <$> token "hello world" <*>
                     symbol '!'
Term       : (++) <$> token "hello world"
Type       : [Char] -> [([Char] -> [Char],[Char])]
Does not match: [Char] -> [(Char -> [Char],[Char])]
```

- The four step approach might result in:

```
(1,7): The function argument of <$> does not
work on the result types of the parser(s)
 function           : (++)
   type             : [a] -> [a] -> [a]
   does not match : String -> Char -> String
```

Assigning phase numbers (1/2) §2

```
  x :: t1;   y :: t2;
---------------------
    x <$> y :: t3;

phase 6
t2 == Parser s1 a2 : right operand is not a parser
t3 == Parser s2 b2 : result type is not a parser
phase 7
s1 == s2 : parser has an incorrect symbol type
phase 8
t1 == a1 -> b1 : left operand is not a function
a1 == a2 : function can't be applied to parser's result
b1 == b2 : parser has an incorrect result type
```

- All phase i constraints solved before phase $i + 1$
- The default phase number is 5

In a similar way, the constraints can be assigned a lower phase number than the default.

If we assign explicit constraints to phase 4, then the following error is reported:

$test :: Parser\ Char\ String$
$test = map\ toUpper <\$> $ `"hello, world!"`

```
(2,21): Type error in string literal
 expression          : "hello, world!"
   type              : String
   expected type     : Parser Char String
```

Some final words on specialized type rules

- Rules are applied by matching expressions below the line on the AST, and then "replacing" the old constraints and error reporting functions with the new.
- The matched expression can also be something like $f <\$> p <*> q$, where f, p and q are meta-variables and the other two are not.
- Matching rules proceeds top to bottom
- Specialized type rules cannot match across lets and lambda's, but the meta-variables may of course represents ASTs that have these.

Another directive: siblings

- Certain combinators are known to be easily confused:
  - cons (:) and append ($+\!\!+$)
  - <$> and <$
  - (.) and ($+\!\!+$) (PHP programmers)
  - ($+$) and ($+\!\!+$) (Java programmers)
- These combinations can be listed among the specialized type rules.

```
siblings   <$> , <$
siblings   ++ , +, .
```

- The siblings heuristic will try a sibling if an expression with such an operator fails to type check.

Example

$$\textbf{data}\ \mathit{Expr} = \mathit{Lambda}\ [\mathit{String}]\ \mathit{Expr}$$

$$\begin{array}{l} pExpr \\ \quad = pAndPrioExpr \\ \ \langle|\rangle\ Lambda \ \ \langle\$\ \ pKey\ \texttt{"\textbackslash\textbackslash"} \\ \qquad\qquad\qquad \langle*\rangle\ many\ pVarid \\ \qquad\qquad\qquad \langle*\ \ pKey\ \texttt{"->"} \\ \qquad\qquad\qquad \langle*\ \ pExpr \end{array}$$

Extremely concise:

```
(11,13): Type error in the operator <*
  probable fix: use <*> instead
```

Permuting function arguments (1/2)

Supplying the arguments of a function in the wrong order can result in incomprehensible type error messages.

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

```
ERROR "Swapping.hs":2 - Type error in application
*** Expression     : option "" (token "hello!")
*** Term           : ""
*** Type           : String
*** Does not match: [a] ->
                    [([Char] -> [([Char],[Char])],[a])]
```

- Check for permuted function arguments in case of a type error
- There is no need to declare this in a .type file

Permuting function arguments (2/2) §2

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

```
(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
```

Summary

We have shown four techniques to influence the behaviour of constraint-based type inferencers.

	fixed order	size of types	standard format	no anticipation
specialized type rules	$\surd$	$\surd$	$\surd$	$\surd$
phasing	$\surd$	$\times$	$\times$	$\times$
siblings	$\times$	$\times$	$\surd$	$\surd$
permuting	$\times$	$\times$	$\surd$	$\surd$

Concluding remarks

- I have shown what can be achieved in the context of Haskell 98 when it comes to domain specific error diagnosis.
- Implemented in the Helium compiler (www.cs.uu.nl/wiki/bin/view/Helium/WebHome)
- More details in
  Heeren, Hage, Swierstra, Scripting The Type Inference Process (ICFP '03).
- See the paper and a follow-up paper on type classes at PADL '05 for many more details (or read Bastiaan's PhD thesis)

Future Work

- Ongoing: Helium on Hackage
- Scaling up to Haskell 2010 (or later)
- Because many libraries/EDSLs use extensions that we do not yet support
  - existentials
  - GADTs
  - type families
  - rank-n
  - multi-parameter type classes
- Proposal for a PhD to actually perform this work is currently under appraisal

Thank you for your attention