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

April 7, 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 ⇒ 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

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

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

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

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)

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

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 True* and *f 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 $\backslash x \rightarrow x + 2$.
- 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* $\rightarrow$ *Int*

Adding let-polymorphism to the mix

- Consider

  let $y = \backslash\, z \rightarrow z$
  in $\backslash\, x \rightarrow y\ x + 2$

- 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 $y :: \forall\, \beta\ .\ \beta \rightarrow \beta$
- Visit the body, introducing α for x, and instantiating β in y to, say, α_2 to give $\alpha_2 \rightarrow \alpha_2$
- Unifying α with α_2 will identify the two, (arbitrarily) leading to $x :: \alpha$ and the instance of $y :: \alpha \rightarrow \alpha$
- Then we perform the unifications of the previous slide

The polymorphic lamdba-calculus

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

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

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

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

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

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., $\alpha == \mathit{Int}$
   - 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 $\mathit{unify}(\tau_1, \tau_2)$, 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 $\tau_1 \rightarrow \tau_1 \equiv \tau_2 \rightarrow \mathit{Int}$

Bottom-up typing rules

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

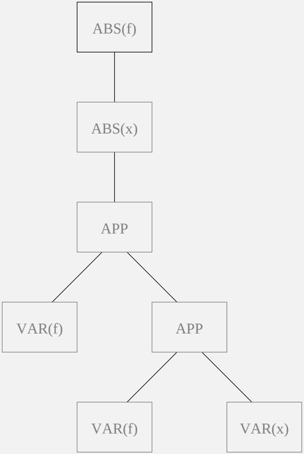
$$\frac{\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\text{BU}}\ e_1 : \tau_1 \qquad \mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\text{BU}}\ e_2 : \tau_2}{\mathcal{A}_1 \cup \mathcal{A}_2,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \to \beta\}\ \vdash_{\text{BU}}\ e_1\ e_2 : \beta} \qquad [\text{APP}]_{\text{BU}}$$

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

- 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

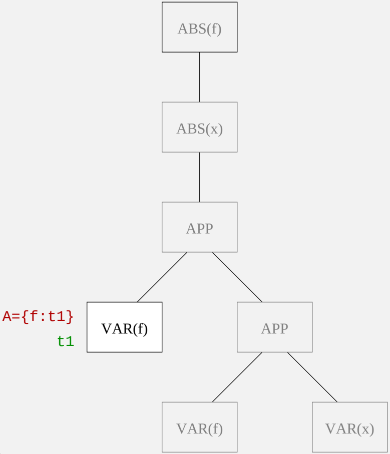
$$twice = \backslash f \to \backslash x \to f\ (f\ x)$$



Constraints

Example

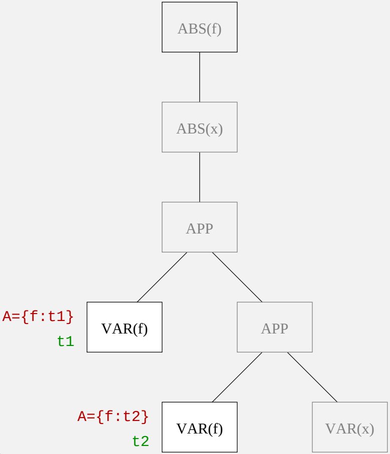
$$twice = \backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$$



Constraints

Example

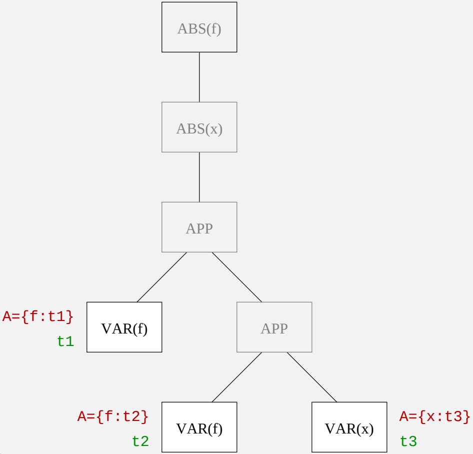
$$twice = \backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$$



Constraints

Example

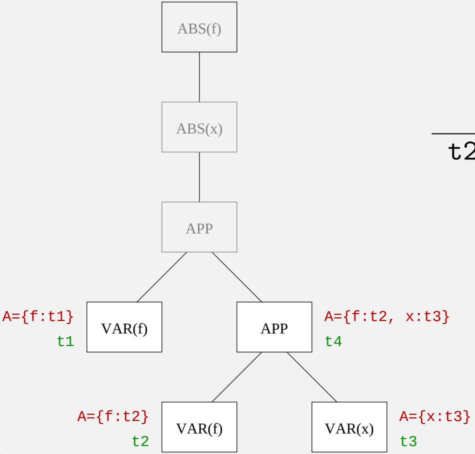
$$twice = \backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$$



Constraints

Example

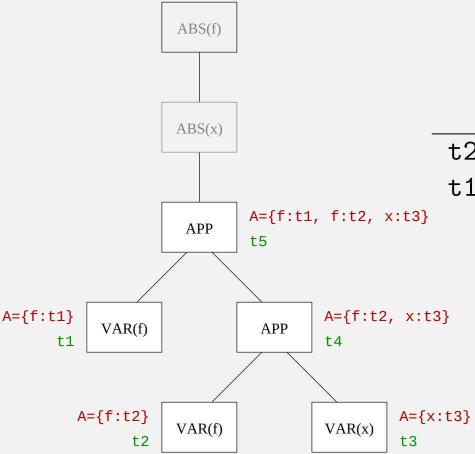
twice = \ *f* -> \ *x* -> *f* (*f* *x*)



Constraints
t2 ≡ t3 -> t4

Example

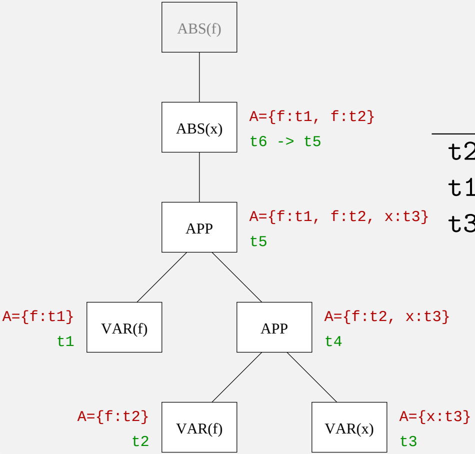
$$twice = \backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$$



Constraints		
t2	≡	t3 -> t4
t1	≡	t4 -> t5

Example

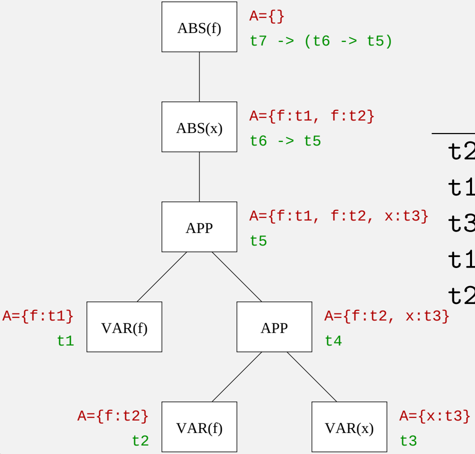
$$twice = \backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$$



Constraints		
t2	≡	t3 -> t4
t1	≡	t4 -> t5
t3	≡	t6

Example

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



Constraints		
t2	≡	t3 -> t4
t1	≡	t4 -> t5
t3	≡	t6
t1	≡	t7
t2	≡	t7

Example

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

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

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

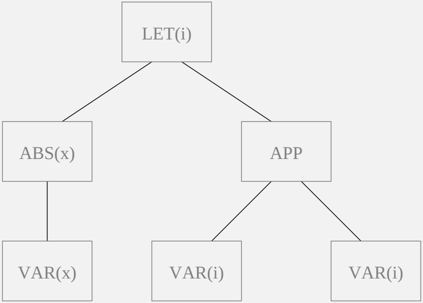
- Example:
  - `[t1 := t2, t4 := t5 -> t5]` satisfies `t4` $\leqslant_\emptyset$ `t1 -> t2`

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

Example

identity = **let** *i* = \ *x* –> *x* **in** *i i*

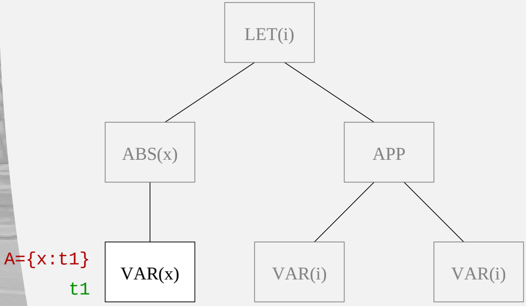
Constraints



Example

identity = **let** *i* = \ *x* −> *x* **in** *i i*

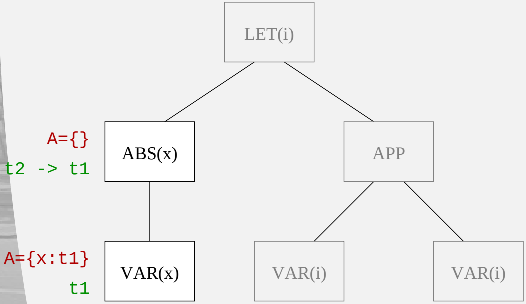
Constraints



Example

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

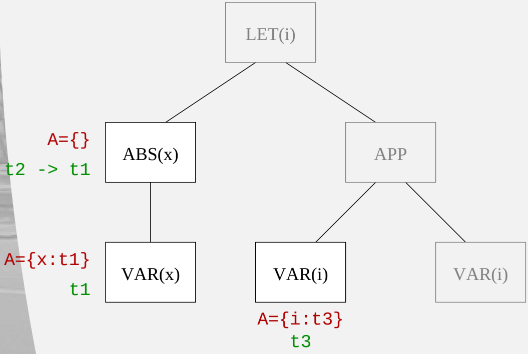
Constraints		
t1	≡	t2



Example

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

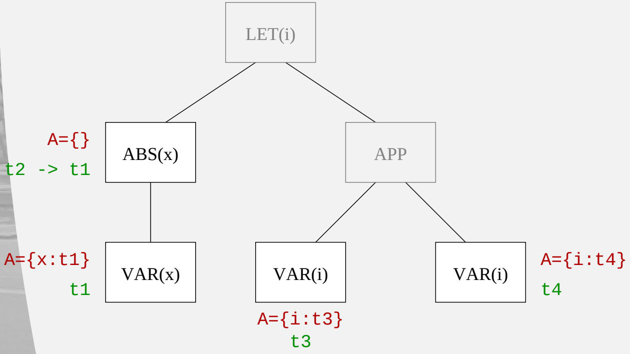
Constraints		
t1	≡	t2



Example

identity = **let** *i* = \ *x* -> *x* **in** *i i*

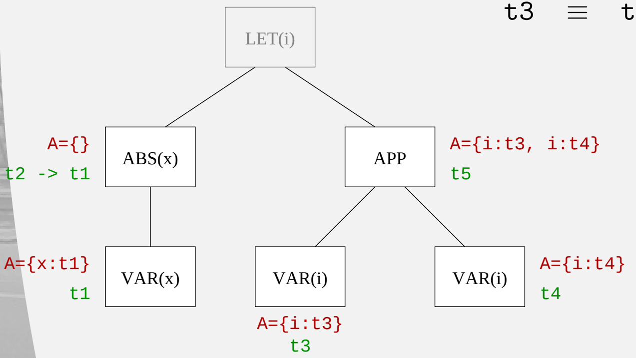
Constraints		
t1	≡	t2



Example

identity = **let** *i* = \ *x* −> *x* **in** *i i*

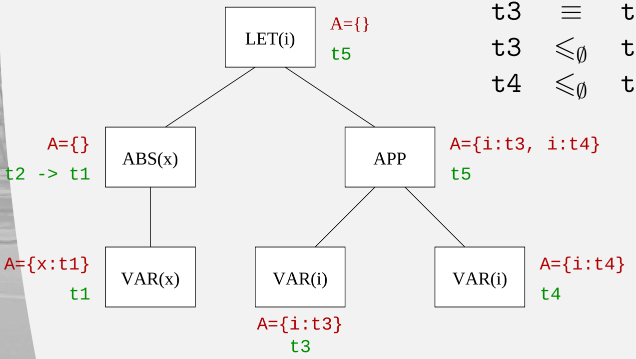
Constraints		
t1	≡	t2
t3	≡	t4 -> t5



Example

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

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



Example

$$\textit{identity} = \textbf{let } i = \backslash x \to x \textbf{ in } i\ i$$

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

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

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:

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

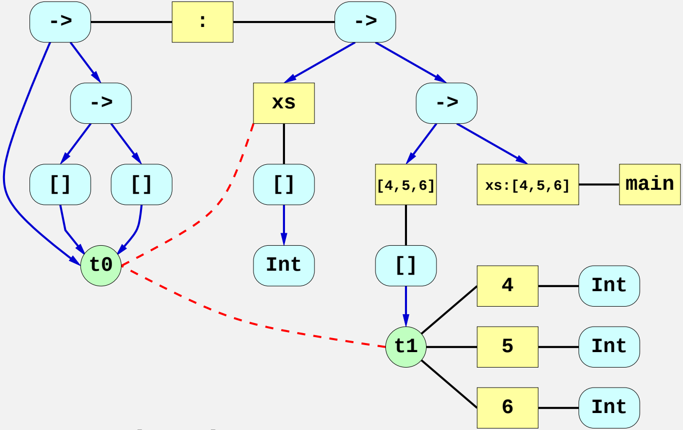
- c_1 is generated by the conditional, but associated with the boolean subexpression.
- Example strategy: left-to-right, bottom-up for then and else part, push down *Bool* (do c_1 before $\mathcal{T}_{\mathcal{C}1}$).

Global constraint solver

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 $xs : [4, 5, 6]$)



$main = xs : [4, 5, 6]$
  where $len = length\ xs$
    $xs = [1, 2, 3]$

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 reported
   , 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* `""` (*token* `"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.

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

- 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.

Who's to blame?

$$wrongxxx :: (Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int$$
$$wrongxxx\ f\ x\ y = \textbf{if}\ f\ (x + y)\ \textbf{then}\ x * y\ \textbf{else}\ x + y$$

Running `helium -d Constraintnr.hs` gets you (a.o.), after some early filters:

```
cnr   edge          ratio   info
-----------------------------------------------------
#12*  (35-97)       100%    {conditional}
#1*   (26-80)       100%    {explicitly typed binding}
#2*   (28-31)       100%    {pattern of function binding}
#5*   (31-36)       100%    {variable}
#11*  (36-96)       100%    {application}
```

The error path

- $wrongxxx :: (Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int$
  $wrongxxx\ \overline{f}^{v28}\ x\ y = \textbf{if}\ \overline{f}^{v36}\ \overline{x + y}^{v37}$
  $\textbf{then}\ x * y\ \textbf{else}\ x + y$

- The error path goes from the explicit type for f as part of *wrongxxx*'s type signature, to the mismatch of the result type of f with the *Bool* the conditional expects:

  $\# 1\ v26 := Inst\ ((Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int)$
  $\# 2\ v28 == v31$
  $\# 5\ v31 == v36$
  $\# 11\ v36 == v37 \rightarrow v35$
  $\# 12\ v35 == Bool$

- The constraint $v26 == v28 \rightarrow v29 \rightarrow v30 \rightarrow v27$ was exonerated earlier.

Blaming a constraint

$$wrongxxx :: (Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int$$

$$wrongxxx\ \overline{f}^{v28}\ x\ y = \textbf{if}\ \overline{f}^{v36}\ \overline{x + y}^{v37}$$
$$\textbf{then}\ x * y\ \textbf{else}\ x + y$$

Run `helium --select-cnr=12 ...` to blame $v35 == Bool$:

```
(9,21): Type error in conditional
 expression       : if f (x + y) then x * y else x + y
 term             : f (x + y)
   type           : Int
   does not match : Bool
```

$v35$ denotes the return type of f, the $Bool$ is the one from the type rule for conditionals.

Blaming a constraint

$$wrongxxx :: (Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int$$

$$wrongxxx\ \overline{f}^{v28}\ x\ y = \textbf{if}\ \overline{f}^{v36}\ \overline{x + y}^{v37}$$
$$\textbf{then}\ x * y\ \textbf{else}\ x + y$$

Constraint #11: $v36 == v37 \rightarrow v35$

```
(20,21): Type error in application
 expression       : f (x + y)
 term             : f
   type           : Int -> Int
   does not match : Int -> Bool
```

Blaming a constraint

$$wrongxxx :: (Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int$$

$$wrongxxx\ \overline{f}^{v28}\ x\ y = \textbf{if}\ \overline{f}^{v36}\ \overline{x + y}^{v37}\ \textbf{then}\ x * y\ \textbf{else}\ x + y$$

Constraint #5: $v31 == v36$

```
(9,21): Type error in variable
 expression        : f
   type            : Int -> Int
   expected type   : Int -> Bool
```

Blaming a constraint

$$wrongxxx :: (Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int$$

$$wrongxxx\ \overline{f}^{v28}\ x\ y = \textbf{if}\ \overline{f}^{v36}\ \overline{x + y}^{v37}\ \textbf{then}\ x * y\ \textbf{else}\ x + y$$

Constraint #2: $v28 == v31$

```
(9,10): Type error in pattern of function binding
 pattern             : f
   type              : Int -> Bool
   does not match : Int -> Int
```

Blaming a constraint

$$wrongxxx :: (Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int$$

$$wrongxxx\ \overline{f}^{v28}\ x\ y = \textbf{if}\ \overline{f}^{v36}\ \overline{x + y}^{v37}\ \textbf{then}\ x * y\ \textbf{else}\ x + y$$

Constraint #1:

$$v26 := Inst\ ((Int \rightarrow Int) \rightarrow Int \rightarrow Int \rightarrow Int)$$

```
(9,1): Type error in explicitly typed binding
 definition          : wrongxxx
   inferred type     : (a    -> Bool) -> a   -> a   -> a
   declared type     : (Int -> Int ) -> Int -> Int -> Int
```

v26 denotes the type inferred for *wrongxxx*'s implementation.
Not all knowledge about *a* has been used.

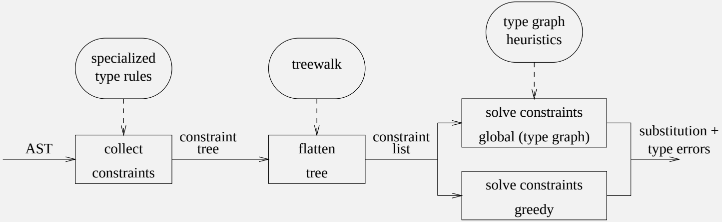
The next logical step...

- Put control over the order of constraint solving in the hands of the programmer
- Associate your own error message with a given constraint
- $\Rightarrow$ domain-specific type error diagnosis

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



IV. Domain Specific Type Error Diagnosis

What is a DSL?

- Walid Taha:
  - the domain is well-defined and central
  - the notation is clear,
  - the informal meaning is clear,
  - the formal meaning is clear and implemented.

What is a DSL?

- Walid Taha:
  - the domain is well-defined and central
  - the notation is clear,
  - the informal meaning is clear,
  - the formal meaning is clear and implemented.
- Missing is:
  - and an implementation of the DSL can communicate with the programmer about the program in terms of the domain
- "domain-abstractions should not leak"

Embedded Domain Specific Languages

- Embedded (internal à la Fowler) Domain Specific Languages are achieved by encoding the DSL syntax inside that of a host language.
- Some (arguable) 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
- According to Hudak "the ultimate abstraction"

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
- Optimisation and error diagnosis are also costly in a non-embedded setting, but there we have more control.
- Can we achieve this control for error diagnosis?

Parser combinators

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

```
pExpr = pAndPrioExpr
    <|> sem_Expr_Lam    -- a function of two arguments
      ⟨$ pKey "\\"
      ⟨*⟩pFoldr1 (sem_LamIds_Cons, sem_LamIds_Nil) pVarid
      ⟨*⟩pKey "->"
      ⟨*⟩pExpr
```

A small mistake

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

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 solution in a nutshell

1. Bring the type inference mechanism under control
   - by phrasing the type inference process as a constraint solving problem (see earlier)
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 (see earlier)
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 in Helium?

- 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

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

- 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.

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

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

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 (pre-*Applicative*), for instance <\$>.

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

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

Specializing a type rule (1/3) §IV

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 (pre-*Applicative*), for instance <$>.

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

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)

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

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, $\langle \$ \rangle$.

$$\frac{x : \tau_1 \qquad y : \tau_2}{x \mathbin{<\$>} y : \tau_3} \qquad \begin{cases} \tau_1 & \equiv \quad a \to b \\ \tau_2 & \equiv \quad \mathit{Parser}\ s\ a \\ \tau_3 & \equiv \quad \mathit{Parser}\ s\ b \end{cases}$$

Specializing a type rule (2/3) §IV

- 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, ⟨\$⟩.

$$\frac{x : \tau_1 \qquad y : \tau_2}{x <\$> y : \tau_3} \qquad \begin{cases} \tau_1 & \equiv & a \to b \\ \tau_2 & \equiv & Parser\ s\ a \\ \tau_3 & \equiv & Parser\ s\ b \end{cases}$$

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

$$\frac{x : \tau_1 \qquad y : \tau_2}{x <\$> y : \tau_3} \qquad \begin{cases} \tau_1 & \equiv & a_1 \to b_1 & s_1 & \equiv & s_2 \\ \tau_2 & \equiv & Parser\ s_1\ a_2 & a_1 & \equiv & a_2 \\ \tau_3 & \equiv & Parser\ s_2\ b_2 & b_1 & \equiv & b_2 \end{cases}$$

Specializing a type rule (3/3)

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

```
  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

$$\mathit{test} :: \mathit{Parser}\ \mathit{Char}\ \mathit{String}$$
$$\mathit{test} = \mathit{map}\ \mathit{toUpper} \langle\$\rangle \texttt{"hello, world!"}$$

This results in the following type error message:

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

Example

$$\mathit{test} :: \mathit{Parser}\ \mathit{Char}\ \mathit{String}$$
$$\mathit{test} = \mathit{map}\ \mathit{toUpper} \langle\$\rangle \texttt{"hello, world!"}$$

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
```

Other facilities

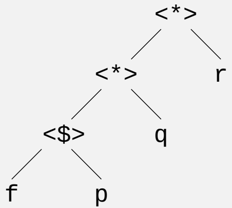

```
   x :: t1;    y :: t2;
----------------------------
  x <$> y :: Parser s b;

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 ...
```

- Interpolate constraints into the rule (cf. *Parser s b*): no effort for default behaviour
- Control over solving order wrt. subexpressions
- Automatic check for soundness and completeness

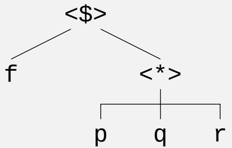
AST versus conceptual structure

$f \langle\$\rangle p \langle * \rangle q \langle * \rangle r$

- 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

Deciding on the order of constraint solving

Consider an expression of the form $f \langle \$ \rangle p1 \langle * \rangle p2 \langle * \rangle \ldots \langle * \rangle pn$, where the *pi* are parsers, and *f* an n-ary function that defines the semantics.

A 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.

Assigning phase numbers


```
  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

Phasing by example

- 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
```

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


```
data Expr = Lambda [String] Expr

pExpr
    = pAndPrioExpr
  <|> Lambda <$ pKey "\\"
             <*> many pVarid
             <* pKey "->"
             <* pExpr
```

Extremely concise:

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

V. Towards Haskell 2010

Introducing DOMSTED

DOMain Specific Type Error Diagnosis

- Enable embedded DSL developers to control the error messages produced by the compiler
- Focus on those errors coming from ill-typed expressions
- Target a full-blown type system
  - Haskell 2010 + type classes, functional dependencies, type families, GADTs, kind polymorphism...
  - In the works: higher-rank and impredicative instantiation
- Constraint-based approach to typing

Why Haskell 98 is not complicated enough

Statistics computed some years back:

Extension	# Hackage	# Top 20
FlexibleInstances	332	10
MultiParamTypeClasses	321	9
FlexibleContexts	232	3
ScopedTypeVariables	192	3
ExistentialQuantification	149	6
FunctionalDependencies	139	4
TypeFamilies	114	1
OverlappingInstances	108	3
Rank2Types	100	3
GADTs	88	3
RankNTypes	81	1
UnboxedTuples	20	4
KindSignatures	20	0

What have we accomplished?

- Two-phase specialized type rules (ESOP 2016)
- Alternative to phasing: regular tree expressions (we may revisit this)
- Implementation on top of `OutsideIn(X)` Experiment at `http://cobalt.herokuapp.com/` (under Domain-specific type rules)
- Syntax of type rules still in flux
- Most recently, we infected GHC with our ideas
  - Part of a third talk

Introducing Persistent

`persistent` is a Haskell library for database access

- Example of embodying knowledge of some domain
- Type-safe approach: each entity is assigned a Haskell type
- Strict separation between:
  1. Values which are kept in the database, v
  2. Primary keys to a certain value, $Key\ v$
  3. Combinations of key and value, $Entity\ v$

Persistent includes the function

$$replace :: Key\ v \rightarrow v \rightarrow m\ ()$$

But if you ever write ill-typed code...

replace 1 *alejandro*

```
No instance for (Num (Key Person))
arising from the literal '1'
```

replace (*key banana*) *alejandro*

```
Cannot unify 'Fruit' with 'Person'
```

- The DSL is not transparent when an error occurs
- Implementation details leak in error messages
  - It gets worse as the host language becomes more complex

Our solution: specialized type rules

We use a different approach syntactically compared to Helium.

replace :: *Key v* –> *v* –> *m* ()

Our solution: specialized type rules

We use a different approach syntactically compared to Helium.

$replace :: Key\ v \rightarrow v \rightarrow m\ ()$

Rewrite the type of *replace* slightly...

$$\begin{aligned} replace\ ::\ & key \sim Key\ v \\ ,\ & value \sim v \\ \Rightarrow\ & key \rightarrow value \rightarrow m\ () \end{aligned}$$

Our solution: specialized type rules

We use a different approach syntactically compared to Helium.

$replace :: Key\ v \rightarrow v \rightarrow m\ ()$

Rewrite the type of *replace* slightly...

$$\begin{aligned} replace\ ::\ & key \sim Key\ v \\ ,\ & value \sim v \\ \Rightarrow\ & key \rightarrow value \rightarrow m\ () \end{aligned}$$

And now add custom error messages!

```
replace :: key   ~ Key v
                 error "The first arg. should be a Key"
         , value ~ v
                 error "Key and value do not coincide"
        => key -> value -> m ()
```

Our solution: specialized type rules

Applies whenever an expression that calls *replace* with two arguments is type incorrect:

```
replace #key #value
    :: constraints #key
     ,  #key   ~ Key v
          error { #key.expr "should be a Key."
                  "Did you forget a wrapper?"}
     ,  constraints #value
     ,  #value ~ v
          error {"Key type" v "and value type"
                  #value.ty "do not coincide"},
   => m ()
```

- Ordering for constraint solving
- Mention expressions and types in messages

Specialized errors for generic constructs

Why *map* instead of *fmap*?

- Many different reasons in play
- One of them, better error messages for beginners

Specialized errors for generic constructs

Why *map* instead of *fmap*?

- Many different reasons in play
- One of them, better error messages for beginners

$$
\begin{aligned}
& fmap\ \#fn\ \#lst \\
& when\ \#lst \sim [a] \\
& \quad ::\ constraints\ \#fn \\
& \quad ,\ \ \#fn \sim s \to r \\
& \qquad error\ \{\#fn.\mathrm{expr}\ \texttt{"is not a function"}\} \\
& \quad ,\ constraints\ \#lst \\
& \quad ,\ \ \#lst \sim [b] \\
& \qquad error\ \{\#lst.\mathrm{expr}\ \texttt{"is not a list"}\} \\
& \quad ,\ s \sim b \\
& \qquad error\ \{\texttt{"Domain"}\ s\ \texttt{"and list type"} \\
& \qquad\qquad b\ \texttt{"do not coincide"}\} \\
& \Rightarrow [r]
\end{aligned}
$$

Specialized errors for generic constructs

Haskell supports monad comprehensions

$$sumpos\ x\ y = [a + b \mid a <\!\!- x, a > 0, b <\!\!- y, b > 0]$$
$$:: (MonadPlus\ m, Num\ a, Ord\ a) => m\ a -> m\ a -> m\ a$$

Supersede list comprehensions, why not make them default?

- One reason is the quality of error messages

Specialized errors for generic constructs

Haskell supports monad comprehensions

$$sumpos\ x\ y = [a + b \mid a <\!\!- x, a > 0, b <\!\!- y, b > 0]$$
$$:: (MonadPlus\ m, Num\ a, Ord\ a) => m\ a -\!\!> m\ a -\!\!> m\ a$$

Supersede list comprehensions, why not make them default?

- One reason is the quality of error messages

Language designers make compromises in order to obtain good error messages for common cases

$\Longrightarrow$ Type-sensitive type rules affect language design

Suggesting reparations

$$select :: [Filter\ v] \rightarrow [SelectOpt\ v] \rightarrow m\ [Entity\ v]$$

$$(==) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

$$(==.) :: PersistField\ t \Rightarrow EntityField\ v\ t \rightarrow t \rightarrow Filter\ v$$

- OK: *select* [*PersonName* ==. "Alejandro"] []
- Wrong: *select* [*PersonName* == "Alejandro"] []

Suggesting reparations

$$select :: [Filter\ v] \rightarrow [SelectOpt\ v] \rightarrow m\ [Entity\ v]$$

$$(==)\ \ :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

$$(==.) :: PersistField\ t \Rightarrow EntityField\ v\ t \rightarrow t \rightarrow Filter\ v$$

- OK: *select* [*PersonName* ==. `"Alejandro"`] []
- Wrong: *select* [*PersonName* == `"Alejandro"`] []

#field == *#value*
when *#field* ∼ *EntityField* *#value* *t*
  :: ⊥ *error* {`"Database field"` *#field*.expr
        `"is being compared using (==)."`
        `"Did you intend to use (==.)?"`}
  ⇒ *Filter* *#value*

How do we achieve this? Stage the solving §V

- In stage 1, we collect and solve a constraint set C (for a given binding group)
- If we have no type error, we extend the environment/substitution and move onto the next binding group.
- Otherwise, we run an algorithm on C to compute a maximal, satisfiable subset of C, and the associated substitution δ.
  - E.g. the subset might tell us *fmap* :: *Functor f* => (*a* -> *b*) -> *f a* -> *f b* that $f = []$
  - In stage 2, solve C again under the assumption of δ and employ the specialized type rules
  - If the substitution says $f = []$ then we can adapt the message accordingly
- Computing a maximal, satisfiable subset is more robust but can be costly

Two stages for one example

$(PersonName^{\,\beta} ==^{\alpha}$ `"Alejandro"` $^{\gamma})^{\,\delta}$

We want the following type rule to be applied:

$$
\begin{array}{l}
\#\,field == \#value \\
when\ \#\,field \sim EntityField\ \#\ value\ t \\
\quad ::\ \ \bot\ error\ \{\ldots\} \\
\quad => Filter\ \#\ value
\end{array}
$$

Two stages for one example

$$((==)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

$$\#\mathit{field} == \#\mathit{value}$$
$$\mathit{when}\ \#\mathit{field} \sim \mathit{EntityField}\ \#\mathit{value}\ t$$

No specialized type rule is applied

$$\alpha \sim \rho \to \rho \to \mathit{Bool} \qquad \alpha \sim \beta \to \gamma \to \delta$$
$$\beta \sim \mathit{EntityField}\ \mathit{Person}\ \mathit{String} \qquad \gamma \sim \mathit{String}$$

Two stages for one example

$$((==)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

$$\#\mathit{field} == \#\mathit{value}$$
$$\mathit{when}\ \#\mathit{field} \sim \mathit{EntityField}\ \#\mathit{value}\ t$$

No specialized type rule is applied

$$\alpha \sim \rho \rightarrow \rho \rightarrow \mathit{Bool} \qquad \alpha \sim \beta \rightarrow \gamma \rightarrow \delta$$
$$\beta \sim \mathit{EntityField}\ \mathit{Person}\ \mathit{String} \qquad \gamma \sim \mathit{String}$$
$$\Downarrow$$

Inconsistent!

Two stages for one example

$$((==)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

$\#\mathit{field} == \#\mathit{value}$
$\mathit{when}\ \#\mathit{field} \sim \mathit{EntityField}\ \#\mathit{value}\ t$

No specialized type rule is applied

$$\alpha \sim \rho \rightarrow \rho \rightarrow \mathit{Bool} \qquad \alpha \sim \beta \rightarrow \gamma \rightarrow \delta$$
$$\beta \sim \mathit{EntityField}\ \mathit{Person}\ \mathit{String} \qquad \gamma \sim \mathit{String}$$
$$\Downarrow$$

Inconsistent!

Prune the constraint set until satisfiability

$$\alpha \sim \beta \rightarrow \gamma \rightarrow \delta \quad \beta \sim \mathit{EntityField}\ \mathit{Person}\ \mathit{String} \quad \gamma \sim \mathit{String}$$

Two stages for one example

$$((==)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

$$\#\mathit{field} == \#\mathit{value}$$
$$\mathit{when}\ \#\mathit{field} \sim \mathit{EntityField}\ \#\mathit{value}\ t$$

No specialized type rule is applied

$$\alpha \sim \rho \to \rho \to \mathit{Bool} \qquad \alpha \sim \beta \to \gamma \to \delta$$
$$\beta \sim \mathit{EntityField\ Person\ String} \qquad \gamma \sim \mathit{String}$$

$$\Downarrow$$

Inconsistent!

Prune the constraint set until satisfiability

$$\alpha \sim \beta \to \gamma \to \delta \quad \beta \sim \mathit{EntityField\ Person\ String} \quad \gamma \sim \mathit{String}$$

$$\Downarrow$$

Now the specialized type rule kicks in

$\perp$ `Database field PersonName is being compared using (==).`

Two stages for one example

$$((==)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

$$\#\mathit{field} == \#\mathit{value}$$
$$\mathit{when}\ \#\mathit{field} \sim \mathit{EntityField}\ \#\mathit{value}\ t$$

No specialized type rule is applied

$$\alpha \sim \rho \to \rho \to \mathit{Bool} \qquad \alpha \sim \beta \to \gamma \to \delta$$
$$\beta \sim \mathit{EntityField}\ \mathit{Person}\ \mathit{String} \qquad \gamma \sim \mathit{String}$$

$$\Downarrow$$

Inconsistent!

Prune the constraint set until satisfiability

$$\alpha \sim \beta \to \gamma \to \delta \quad \beta \sim \mathit{EntityField}\ \mathit{Person}\ \mathit{String} \quad \gamma \sim \mathit{String}$$

$$\Downarrow$$

Now the specialized type rule kicks in

$\bot_{\texttt{Database field PersonName is being compared using (==).}}$

$$\Downarrow$$

The desired error message is shown to the user

Soundness and completeness

Specialized type rules should not tamper with the type system

1. Generate a meta-expression which encompasses all possible instantiations of the type rule
2. Gather set of constraints S_{with} using specialized type rules
3. At the same time, recall all type preconditions $\mathcal{P}$
4. Gather set of constraints S_{none} using only default type rules
5. Prove that $\mathcal{P} \wedge S_{with} \implies S_{none}$ (soundness) and/or $\mathcal{P} \wedge S_{none} \implies S_{with}$ (completeness)

VI. Customizing type error diagnosis in GHC

Ondertussen, in GHC...

instance *TypeError* (*Text* "Cannot 'Show' functions." :$$: *Text* "Perhaps a missing argument?") => *Show* (*a* -> *b*) **where** ...

- Leverages type-level programming techniques in GHC (Diatchki, 2015)
- Very restricted:
  - Only available for type class and family resolution
  - May not influence the ordering of constraints
  - Messages cannot depend on who generated the constraint

How far can we take this?

We provide

- control over the content of the type error message
    - the same constraint (to the solver) may result in different messages
- (some) control over the order in which constraints are checked
- Expression level error messages by type level programming
- GHC's abstraction facilities allow for reuse and uniformity
    - A type level embedded DSL for diagnosing embedded DSLs
- integrated as a patch in GHC version 8.1.20161202
- soundness and completeness for free

How much effort is involved?

- We get a lot for a few non-invasive changes to GHC, with *TypeError* and the *Constraint* kind as enablers
- Constraint resolution needs some changes to track messages, and deal with priorities
- A few additions to *TypeLits.hs* in the base library and a new module *TypeErrors.hs* (62 lines) that exposes the API
- One additional compiler pragma `CHECK_ARGS_BEFORE_FN`.
- We employ many language extensions:

  `DataKinds, TypeOperators, TypeFamilies,`
  `ConstraintKinds, FlexibleContexts, PolyKinds,`
  `UndecidableInstances, UndecidableSuperclasses`

  but the EDSL programmer only the first four, the EDSL user none.

A very stupid mistake

$$intid :: Int$$
$$intid = id' \; True$$

A very stupid mistake

$intid :: Int$
$intid = id'\ True$

```
FormatEx.hs:17:9: error:
    * Hi! You must be Donald. Donald, please read
      this error message. It's a great error message.
      The argument and result types of 'id' do not
      coincide: Bool vs. Int
    * In the expression: id' True
      In an equation for 'intid': intid = id' True
```

Our wrapped Donald-aware identity function §VI

```
id' :: CustomErrors
   '[ '[a :≁: b
          :⇛: E.Text "Hi! You must be Donald. "
          :◇: E.Text "Donald, please read this error message."
          :◇: E.Text " It's a great error message."
          :$$:
         E.Text "The argument and result types of 'id'"
          :◇: E.Text " do not coincide: " :◇: VS a b]
    ] => a -> b
id' = id
```

- *E* qualifier to address type level *Text*
- *id′* is a type error aware wrapper for *id*
- *id′* = *id* ensures *id′* is sound
- Completeness can be achieved too, dually
- With `{#- INLINE id' -#}` no run-time overhead

Type errors Class I: type inconsistencies

From the *diagrams* library (Yorgey, 2012/2016)

$$
\begin{aligned}
\mathit{atop} \;::\; & (\mathit{OrderedField}\ n, \mathit{Metric}\ v, \mathit{Semigroup}\ m) \\
=> & \ \mathit{QDiagram}\ b\ v\ n\ m \ -> \\
& \ \mathit{QDiagram}\ b\ v\ n\ m \ -> \\
& \ \mathit{QDiagram}\ b\ v\ n\ m
\end{aligned}
$$

writing *atop True* gives

```
Couldn't match type 'QDiagram b v n m' with type 'Bool'
```

or for *atop cube3d plane2d* might give

```
Couldn't match type 'V2' with type 'V3'
```

Type error Class II: left-undischarged errors §VI

From the *persistent* library (Snoyman, 2012)

$$
\begin{aligned}
\mathit{insertUnique} \;::\; & (\mathit{MonadIO}\ m, \mathit{PersistUniqueWrite}\ \mathit{backend},\\
& \mathit{PersistEntity}\ \mathit{record})\\
\Rightarrow\; & \mathit{record} \rightarrow\\
& \mathit{ReaderT}\ \mathit{backend}\ m\ (\mathit{Maybe}\ (\mathit{Key}\ \mathit{record}))
\end{aligned}
$$

use of *insertUnique* gives rise to type class predicates that may be left undischarged, because the programmer forgot to write a *PersistEntity* instance.

We'd like to get something like:

```
Data type 'Person' is not declared as a Persistent
entity. Hint: entity definition can be automatically
derived. Read more at http://www.yesodweb.com/...
```

Type error Class III: ambiguous type errors

- Defaulting seems to be a more apt solution, or simply adding type annotations
- We wondered: are these ever "domain-specific"? We'd like to hear about it.
- Our work handles Class I and Class II errors

Before we go on: *Constraints*

GHC supports a special kind *Constraint* so that type level programming can be applied to constraints

type $JSONSerializable\ a = (FromJSON\ a, ToJSON\ a)$

and use type families as type-level functions:

$$\textbf{type}\ family\ All\ (c :: k \rightarrow Constraint)\ (xs :: [k])\ \textbf{where}$$
$$All\ c\ [\,] = ()$$
$$All\ c\ (x : xs) = (c\ x, All\ c\ xs)$$

so we can write $All\ Show\ [Int, Bool]$ instead of $(Show\ Int, Show\ Bool)$

This is what opens the door to manipulating constraints and type error messages in a reusable fashion.

The running example

$$
\begin{aligned}
\mathit{atop} \;::\; & (\mathit{OrderedField}\ n, \mathit{Metric}\ v, \mathit{Semigroup}\ m) \\
\Rightarrow\; & \mathit{QDiagram}\ b\ v\ n\ m \rightarrow \\
& \mathit{QDiagram}\ b\ v\ n\ m \rightarrow \\
& \mathit{QDiagram}\ b\ v\ n\ m
\end{aligned}
$$

can also be written as

$$
\begin{aligned}
\mathit{atop} \;::\; & (d_1 \sim \mathit{QDiagram}\ b_1\ v_1\ n_1\ m_1, \\
& \ d_2 \sim \mathit{QDiagram}\ b_2\ v_2\ n_2\ m_2, \\
& \ b_1 \sim b_2, v_1 \sim v_2, n_1 \sim n_2, m_1 \sim m_2, \\
& \ \mathit{OrderedField}\ n_1, \mathit{Metric}\ v_1, \mathit{Semigroup}\ m_1) \\
\Rightarrow\; & d_1 \rightarrow d_2 \rightarrow d_1
\end{aligned}
$$

Failure to satisfy either $b_1 \sim b_2$ or $v_1 \sim v_2$ should lead to different messages.

Approach I: attaching hints to constraints

```
atop    :: (
   (d1 ~ QDiagram b1 v1 n1 m1)
    `IH` (Text "argument #1 to 'atop' must be a diagram"),
   (d2 ~ QDiagram b2 v2 n2 m2)
    `IH` (Text "argument #2 to 'atop' must be a diagram"),
   (b1 ~ b2)
    `IH` (Text "the diagrams must use the same back-end"),
   (v1 ~ v2)
    `IH` (Text "diagrams must live in the same vector space"),
    ... same for n1, n2, m1 and m2
   OrderedField n1, Metric v1, Semigroup m1)
    => d1 -> d2 -> d1
atop = Diagrams.Combinators.atop
```

The constraint solving machinery propagates messages along with the associated type level error message. The *IH* annotations/predicates ensure the message is reported.

Some observations

- Message is attached as a hint if a constraint cannot be satisfied

  example = *atop True* 'c'

  ```
  * Couldn't match type 'QDiagram b v n m' with 'Bool'
    ...
  * In the expression: atop True 'c'
    ...
  * Hint: argument #1 to 'atop' must be a diagram
  ```

- Very simple to implement
- May sometimes give unexpected results (more info in the paper)

Before we go on: a Class II example

We can also associate a hint with a type class predicate so that the hint is shown if that predicate is left undischarged:

```
insertUnique ::
    (  MonadIO m, PersistUniqueWrite backend,
       PersistEntity record `LeftUndischargedHint` (
       Text "Data type '"
         :◇: ShowType record
         :◇: Text "' is not declared as entity."
         :$$: Text "Hint: entity definition can be "
         :◇: "automatically derived."
         :$$: Text "Read more at http://www.yesodweb.com/..."
    )
  => record -> ReaderT backend m (Maybe (Key record))
```

Approach II: controlling the order

- The problem of Approach I arises from the order in which constraints may be solved by the constraint solver
- The solution is to give control over that order to the developer
- The basic combinator we introduce is *IfNot*

  $IfNot\ (c :: Constraint)\ (fail :: Constraint)\ (ok :: Constraint)$

- **IMPORTANT:** the *ok* branch will also be chosen if the constraint c is not yet known to be consistent or not!
- E.g., if $c = \alpha \sim \beta$, we have to wait for more information.
- In other words: *IfNot* does not perform a unification.

Example

$atop ::$
$\quad IfNot\ (d_1 \sim QDiagram\ b_1\ v_1\ n_1\ m_1)$
```
      (TypeError "Arg. #1 to 'atop' must be a diagram")
      (IfNot (d2 ~ QDiagram b2 v2 n2 m2)
        (TypeError "Arg. #2 to 'atop' must be a diagram")
        (IfNot (b1 ~ b2)
          (TypeError "Back-ends do not coincide")
        ....))))
   => d1 -> d2 -> d1
```

- Better syntax later (defined on top of *IfNot*)

Controlling the solving order

- *IfNots* can be nested which induce a preferred solving order
- The constraint solver uses priorities to ensure solving obeys the dictated order (more details in the paper)
- The priorities cannot be generally controlled in relation to the rest of the program: too invasive
- We do offer one pragma: `CHECK_ARGS_BEFORE_FN`.
  - Ensures that we get the most out of arguments before looking at the application

From *IfNot* to *WhenApart*

- $\mathit{WhenApart}\ a\ b\ f\ o$ represents $\mathit{IfNot}\ (a \sim b)\ f\ o$
- *WhenApart* was introduced along with closed type families: the constraint is true if at this point a and b can never be reconciled.
- We cannot reduce $\mathit{Int} :==: \alpha$ until we know more about α, but if we have $\mathit{Int} :==: [\alpha]$ we can rewrite to *False* for the following type family:

$$\textbf{type}\ \mathit{family}\ a :==: b :: \mathit{Bool}\ \textbf{where}$$
$$a :==: a = \mathit{True}$$
$$a :==: b = \mathit{False}$$

The EDSL-developer facing API (version 1)

Apartness is represented by the operator

infixl 5 $:\not\sim:$

We deal with two kinds of failure:

data *ConstraintFailure* $=$
    $\forall\ t\ .\ t :\not\sim: t$ | *Undischarged Constraint*

A *CustomError* is then a failure and a message

infixl 4 $:\Rrightarrow:$
data *CustomError* $=$
    ConstraintFailure $:\Rrightarrow:$ *ErrorMessage* | *Check Constraint*

The latter if we do not want a message.

Running back to our example

$$
\begin{array}{l}
atop :: CustomErrors\ [\\
\quad d_1 :\not\sim: QDiagram\ b_1\ v_1\ n_1\ m_1\\
\qquad :\Rrightarrow: Text\ \texttt{"Arg. \#1 to 'atop' must be a diagram"},\\
\quad d_2 :\not\sim: QDiagram\ b_2\ v_2\ n_2\ m_2\\
\qquad :\Rrightarrow: Text\ \texttt{"Arg. \#2 to 'atop' must be a diagram"},\\
\quad b_1 :\not\sim: b_2\\
\qquad :\Rrightarrow: Text\ \texttt{"Back-ends do not coincide"},\\
\quad \ldots\\
\quad Check\ (OrderedField\ n_1), Check\ (Metric\ v_1),\\
\quad Check\ (Semigroup\ m_1)\\
\quad] \Rightarrow d_1 \to d_2 \to d_1
\end{array}
$$

The *CustomErrors* type family traverses the list to build the constraint structure.

Abstraction and reuse

For consistency and conciseness we can define a type level implementation for the checks of back-ends, vector spaces, etc.

$$\textbf{type}\ \mathit{DoNotCoincide}\ \mathit{what}\ a\ b =$$
$$a :\not\sim: b :\Rrightarrow: \mathit{Text}\ \mathit{what} :\diamond: \mathit{Text}\ \texttt{" do not coincide: "}$$
$$:\diamond: \mathit{ShowType}\ a :\diamond: \mathit{Text}\ \texttt{" vs. "} :\diamond: \mathit{ShowType}\ b$$

Note that *ShowType* and type level *Text*s are provided by GHC.

The EDSL-developer facing API (version 2) §VI

Some constraints can be checked independently: partition constraints into a list of lists.

```
atop :: CustomErrors [
    [d1 :≁: QDiagram b1 v1 n1 m1
         :⇛: Text "Arg. #1 to 'atop' must be a diagram",
     d2 :≁: QDiagram b2 v2 n2 m2
         :⇛: Text "Arg. #2 to 'atop' must be a diagram"],
    [DoNotCoincide "Back-ends"             b1 b2,
     DoNotCoincide "Vector spaces"         v1 v2,
     DoNotCoincide "Numerical fields"      n1 n2,
     DoNotCoincide "Query annotations"     m1 m2],
    [Check (OrderedField n1), Check (Metric v1),
     Check (Semigroup m1)]
  ] => d1 -> d2 -> d1
```

Return of the giant siblings

$$
\begin{array}{l}
((\langle\$\rangle) :: \mathit{Sibling}\ \texttt{"(<\$>)"}\ (\mathit{Applicative}\ f)\ ((a \to b) \to f\ a \to f\ b) \\
\qquad\qquad\qquad\quad \texttt{"(<\$)"} \qquad\qquad\qquad\quad (a \to f\ b \to f\ a) \\
\qquad\qquad\qquad\quad \mathit{fn} \\
\qquad\quad \Rightarrow \mathit{fn}
\end{array}
$$

Given $f :: \mathit{Char} \to \mathit{Bool} \to \mathit{Int}$,
$f\ \langle\$\rangle\ [1 :: \mathit{Int}]\ \langle*\rangle$ "a" $\langle*\rangle\ [\mathit{True}]$ leads to

```
* Type error in '(<$>)', do you mean '(<$)'
* In the first argument of '(<*>)', namely 'f <$> [1
  ...
```

What are these *fns* doing there?

Implementation of siblings

We can define siblings on top of what we have (almost):

$$
\begin{array}{l}
\textbf{type } \mathit{Sibling\ nameOk\ extra\ tyOk\ nameWrong\ tyWrong\ fn} \\
\quad = \mathit{IfNot}\ (\mathit{fn} \sim \mathit{tyOk}) \\
\qquad\quad (\mathit{ScheduleAtTheEnd}\ (\mathit{IfNot}\ (\mathit{fn} \sim \mathit{tyWrong}) \\
\qquad\qquad (\mathit{fn} \sim \mathit{tyOk}) \\
\qquad\qquad (\mathit{TypeError}\ (\mathit{Text}\ \texttt{"Type error in '"} \ldots \\
\qquad\qquad\qquad\qquad\quad \mathit{Text}\ \texttt{"', do you mean '"} \ldots)))) \\
\qquad\quad \mathit{extra}
\end{array}
$$

One caveat: we need *ScheduleAtTheEnd* to assign the lowest possible priority (otherwise $\mathit{fn} \sim \mathit{tyWrong}$ may succeed while other constraints in the set contradict it).

Alternatives and conversions

- *diagrams* distinguishes vectors from points
- You can compute the perpendicular of a vector (but not a point (pair)) with *perp*
- Can we provide a hint on how to convert a pair to a vector if the argument happens to be a pair?

```
* Expecting a 2D vector but got a tuple.
  Use 'r2' to turn the tuple into a vector.
```

As with siblings this may not be what the programmer intends, but the change will resolve the type error.

The new definition of *perp*

```
perp :: CustomErrors [
  [v :≁: V2 a :⇛?:
    ([v ~ (a, a) :⇛!:
      Text "Expecting a 2D vector but got a tuple."
      :$$: Text "Use r2 to turn a tuple into a vector."
      ],
      Text "Expected a 2D vector, but got "
      :◇: ShowType v)],
  [Check (Num a)]] => v -> v
```

With every apartness check we can associate a list of further checks on what in this case *v* might actually be.

A word of warning

- Why is the unification $v \sim (a, a)$ not so dangerous now?
- If we arrive there at all we know:
  - compilation will fail
  - we know the top level type constructor of v
- However: writing (a, a) does imply that a unification may take place.
- To be safe: only compare against $T\ a1 \mathinner{..} an$ with T a fixed type constructor, and all ai fresh.

What we did on our holidays

- We have worked out some rules for
  - *path* (Chris Done, 2015), appendix to the paper
  - *diagrams*
  - *persistent*
  - *map*, *Eq*, and making *foldr* and *foldl* siblings
  - *formatting* (Chris Done)
- They can be added to members of type classes too!

Wrapping up

Let's visit the Terminal/jEdit and take a look at *now* and (%).

Back to our slogan

Expression level type error messages
by
type level programming

- In retrospect, this makes a lot of sense
- Kind level programming for diagnosing type level programming?
- Possible relationships with dependently typed programming, staged programming, and higher-ranked analyses with effect operators
  - All provide a way to perform computations at the type level/compile time, with different restrictions.

Beyond the scope of this tutorial

- Type error diagnosis in Elm (with Falco Peijenburg and Alejandro Serrano)
- Type error diagnosis in LambdaPi (with Joey Eremondi and Wouter Swierstra)
- Refining type guards in Typescript (with Ivo Gabe de Wolff)
- Unification modulo type isomorphism (with Arjen Langebaerd and Bastiaan Heeren)
- Questions can be asked off-line

Ongoing and future work

- More domain-specific type error diagnosis in GHC

Ongoing and future work

- More domain-specific type error diagnosis in GHC
- Impredicativity in OutsideIn(X) (with Simon PJ and Dimitrios Vytiniotis)

Ongoing and future work

- More domain-specific type error diagnosis in GHC
- Impredicativity in OutsideIn(X) (with Simon PJ and Dimitrios Vytiniotis)
- Type classes in Helium

Ongoing and future work

- More domain-specific type error diagnosis in GHC
- Impredicativity in OutsideIn(X) (with Simon PJ and Dimitrios Vytiniotis)
- Type classes in Helium
- Analysis-specific error diagnosis: what if we introduce strictness or uniqueness types?

Ongoing and future work

- More domain-specific type error diagnosis in GHC
- Impredicativity in OutsideIn(X) (with Simon PJ and Dimitrios Vytiniotis)
- Type classes in Helium
- Analysis-specific error diagnosis: what if we introduce strictness or uniqueness types?
- Helping Alejandro (Russo) with this MAC problems

Thank you for your attention

VII. Other approaches

Type error slicing

- Sulzmann, Wazny, Stuckey: Chameleon system
  - Could deal with some language extensions
- Haack and Wells, and later also Rahli and a few others: type error slicing for (full) ML
- Thomas Schilling did something like this for Haskell

Heuristics

- IFL 2006, Helium's heuristics
- Nabil el Boustani translated it to Generic Java
- Danfeng Zhang and Andrew Myers (Bayesian predictor)
- Pavlinovic et al. (uses SMT solver to find optimal solutions)

Repair systems

- Suggesting fixes
- Helium's siblings
- McAdam's unification modulo type isomorphism
- Arjen Langebaerd MSc thesis (on Helium)

Black box systems

- Seminal (Lerner et al.): use type system on variations of the type incorrect program to determine how to diagnose the error
- Advantage: non-invasive, low effort and low risk

Counterfactual typing

- Erwig and Chen: counterfactual typing
- Allows the use of a type incorrect identifier to decide the correct type for the identifier
- The basis of the technology comes from feature selection

Beyond the intrinsic type system

- Weijers, Hage and Holdermans, Security type error diagnosis, SCP 2014
- Combines slicing (to get an over approximation of the locations), uses heuristics to further narrow down

Domain specific type error diagnosis

- Scripting The Type Inferencer by Heeren, Hage, Swierstra, 2003
- For Scala Hubert Plociniczak, Odersky and others did something similar
- Current work on Domsted

Post-processing

- David Raymond Christensen: better error diagnosis through post-processing in the dependently type Idris language