Constraint-based Type Error Diagnosis (Tutorial)

Jurriaan Hage

Department of Information and Computing Sciences, Universiteit Utrecht

J.Hage@uu.nl

February 10, 2017
About me

▶ Assistant professor in Utrecht, Software Technology
▶ Topics of interest:
  ▶ Static analysis of functional languages
    ▶ Non-standard/type and effect systems
  ▶ On and off: program plagiarism detection, object-sensitive analysis, soft typing of dynamic languages, and switching classes
  ▶ PhD students active in legacy system modernization, and testing
  ▶ Type error diagnosis (for functional languages/EDSLs)
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 equipped with an intrinsic type system, preventing some structurally correct programs from being compiled
- "well-typed programs can’t go wrong"
- type incorrect programs $\Rightarrow$ the need for diagnosis
- When type checking we typically assume various simple local properties to have been checked:
  - syntactic correctness
  - well-scopedness
  - definedness of variables
- Which properties it enforces, depends intimately on the language
  - Cf. does every function have the right number of arguments in C vs. Haskell
What is type error diagnosis?

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

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

\[
\begin{align*}
\text{reverse} & = \text{foldr} (\mathit{flip} (:)) [] \\
\text{palindrome } xs & = \text{reverse } xs == xs
\end{align*}
\]

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

Anyone know the likely fix? foldr should be foldl
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 succinct
► 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
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 succinct
- 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

\[ xxxx = xs : [4, 5, 6] \]
\[
\text{where } \text{len} = \text{length } xs \\
xs = [1, 2, 3]
\]
Unresolved top-level overloading

\[
xxxx = xs : [4, 5, 6]
\]
\[
\text{where } \text{len} = \text{length } xs
\]
\[
x = [1, 2, 3]
\]

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

\[ p\text{Expr} = p\text{AndPrioExpr} \]

\[ \langle | \rangle \text{ sem}_\text{Expr}_\text{Lam} \]

\[ \langle $ \rangle \text{ pKey } \"\\" \]

\[ \langle * \rangle \text{ pFoldr1} \ (\text{sem}_\text{LamIds}_\text{Cons}, \text{sem}_\text{LamIds}_\text{Nil}) \text{ pVarid} \]

\[ \langle * \rangle \text{ pKey } \"->" \]

\[ \langle * \rangle \text{ pExpr} \]

gives

ERROR "BigTypeError.hs":1 - Type error in application
*** Expression : \text{sem}_\text{Expr}_\text{Lam} \langle $ \rangle \text{ pKey } \"\\" \langle * \rangle \text{ pFoldr1} \ (\text{sem}_\text{LamIds}_\text{Cons}, \text{sem}_\text{LamIds}_\text{Nil}) \text{ pVarid} \langle * \rangle \text{ pKey } \"->" \]
*** Term : \text{sem}_\text{Expr}_\text{Lam} \langle $ \rangle \text{ pKey } \"\\" \langle * \rangle \text{ pFoldr1} \ (\text{sem}_\text{LamIds}_\text{Cons}, \text{sem}_\text{LamIds}_\text{Nil}) \text{ pVarid} \]
*** Type : [Token] \rightarrow \{([Char] \rightarrow Type \rightarrow Int \rightarrow ([Char],Type,Int,Int)) \rightarrow int \rightarrow Int \rightarrow ([Int,(Bool,Int)]) \rightarrow (PP\_Doc,Type,a,b,[c] \rightarrow [Level],[S] \rightarrow [S])) \rightarrow Type \rightarrow d \rightarrow ([Char],Type,Int,Int)) \rightarrow Int \rightarrow Int \rightarrow e \rightarrow (PP\_Doc,Type,a,b ,f \rightarrow f,[S] \rightarrow [S]),[Token])\}
*** Does not match : [Token] \rightarrow ([Char] \rightarrow Type \rightarrow d \rightarrow ([Char],Type,Int,Int) ) \rightarrow Int \rightarrow Int \rightarrow e \rightarrow (PP\_Doc,Type,a,b,f \rightarrow f,[S] \rightarrow [S]),[Token])\}
Order is arbitrary (in Hugs)

\[
\text{yyyy} :: (\text{Bool} \to a) \to (a, a, a) \\
\text{yyyy} = \lambda f \to (f \text{ True}, f \text{ False}, f \, [])
\]

What's wrong with this program?
Order is arbitrary (in Hugs)

```haskell
yyyy :: (Bool -> a) -> (a, a, a)
yyyy = \f -> (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
zzzz = \ f \rightarrow (f \ [], f \ True, f \ False)

Ov.hs:8:23:
Could’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. Succinct: 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
Consider the expression \( x \rightarrow x + 2 \).

Hindley-Milner will

- introduce a fresh \( \alpha \) for \( x \)
- look at the body \( x + 2 \): unify the arguments of \( + \) with their formal types (here all \( Int \))
- \( \alpha \) becomes \( Int \), and the whole expression has type \( Int \rightarrow Int \)
Adding let-polymorphism to the mix

Consider

```latex
let \ y = \ \lambda \ z \ . \ z
\in \ \lambda \ x \ . \ y \ x + 2
```

For \(\ z\), \(\alpha_1\) is introduced, so that the body of \(\ y\) has type \(\alpha_1\).

Since \(\alpha_1\) does not show up in any other type (it is free) we may generalize over \(\alpha_1\) so that \(\ y :: \forall \beta . \beta \rightarrow \beta \)

Visit the body, introducing \(\alpha\) for \(\ x\), and instantiating \(\beta\) in \(\ y\) to, say, \(\alpha_2\) to give \(\alpha_2 \rightarrow \alpha_2\)

Unifying \(\alpha\) with \(\alpha_2\) 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 lambda-calculus

\[ \tau \prec \Gamma(x) \]
\[ \Gamma \vdash_{\text{HM}} x : \tau \]

\[ \Gamma \vdash_{\text{HM}} e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_{\text{HM}} e_2 : \tau_1 \]
\[ \Gamma \vdash_{\text{HM}} e_1 \ e_2 : \tau_2 \]

\[ \Gamma \vdash_{\text{HM}} \lambda x \rightarrow e : (\tau_1 \rightarrow \tau_2) \]

\[ \Gamma \vdash_{\text{HM}} e_1 : \tau_1 \quad \Gamma \setminus x \cup \{x: \text{generalize}(\Gamma, \tau_1)\} \vdash_{\text{HM}} e_2 : \tau_2 \]
\[ \Gamma \vdash_{\text{HM}} \text{let } x = e_1 \text{ in } e_2 : \tau_2 \]

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

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

- Unifications are performed in a fixed order
- Order may be changed: many alternative implementations of HM exist
- Order of unification is unimportant for the resulting types,
- but it is important if you blame the first unification that is inconsistent with the foregoing.
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., $\alpha == \text{Int}$
   ▶ 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 $\text{unify}(\tau_1, \tau_2)$, 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 $\text{unify}(\tau_1, \tau_2)$, 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. $S$ satisfies $(\tau_1 \equiv \tau_2) = \text{def} \quad S\tau_1 = S\tau_2$
- Example:
  - $[\tau_1 := \text{Int, } \tau_2 := \text{Int}]$ satisfies $\tau_1 \rightarrow \tau_1 \equiv \tau_2 \rightarrow \text{Int}$
Bottom-up typing rules

\[
\begin{align*}
\{x:\beta\}, \emptyset & \vdash_{\text{BU}} x: \beta \\
\mathcal{A}_1, \mathcal{C}_1 & \vdash_{\text{BU}} e_1 : \tau_1 & \mathcal{A}_2, \mathcal{C}_2 & \vdash_{\text{BU}} e_2 : \tau_2 \\
\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \rightarrow \beta\} & \vdash_{\text{BU}} e_1 \ e_2 : \beta
\end{align*}
\]

\[\mathcal{A}, \mathcal{C} \vdash_{\text{BU}} e : \tau\]

\[\mathcal{A}\setminus x, \mathcal{C} \cup \{\tau' \equiv \beta \mid x:\tau' \in \mathcal{A}\} \vdash_{\text{BU}} \lambda x \rightarrow e : (\beta \rightarrow \tau)\]

- 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
  - \(\tau\): assigned type (variable)
Example

\[ \text{twice} = f \mapsto x \mapsto f(f(x)) \]

Constraints
Example

\[
twice = \lambda f \rightarrow \lambda x \rightarrow f(f(x))
\]
Example

\[ twice = \lambda f \rightarrow \lambda x \rightarrow f(f(x)) \]

Constraints
Example

\[ \text{twice} = \lambda f \rightarrow \lambda x \rightarrow f(f\,x) \]

Constraints
Example

\[ twice = \lambda f \rightarrow \lambda x \rightarrow f(f(x)) \]

Constraints

\[ t_2 \equiv t_3 \rightarrow t_4 \]
Example

\[ \textit{twice} = \lambda f \rightarrow \lambda x \rightarrow f(f x) \]

Constraints

\[
\begin{align*}
t_2 & \equiv t_3 \rightarrow t_4 \\
t_1 & \equiv t_4 \rightarrow t_5
\end{align*}
\]
Example

\[ \text{twice} = \lambda f \rightarrow \lambda x \rightarrow f(f(x)) \]

Constraints

\[
\begin{align*}
t2 & \equiv t3 \rightarrow t4 \\
t1 & \equiv t4 \rightarrow t5 \\
t3 & \equiv t6
\end{align*}
\]
Example

\[ \text{twice} = \forall f \rightarrow \forall x \rightarrow f(f(x)) \]

Constraints:

\[
\begin{align*}
t2 & \equiv t3 \rightarrow t4 \\
t1 & \equiv t4 \rightarrow t5 \\
t3 & \equiv t6 \\
t1 & \equiv t7 \\
t2 & \equiv t7
\end{align*}
\]
Example

\[ twice \triangleq \lambda f \rightarrow \lambda x \rightarrow f (f \, x) \]

\[ C = \begin{cases} 
  t_2 & \equiv t_3 \rightarrow t_4 \\
  t_1 & \equiv t_4 \rightarrow t_5 \\
  t_3 & \equiv t_6 \\
  t_1 & \equiv t_7 \\
  t_2 & \equiv t_7 
\end{cases} \]

\[ S = \begin{cases} 
  t_1, t_2, t_7 & := t_6 \rightarrow t_6 \\
  t_3, t_4, t_5 & := t_6 
\end{cases} \]

\[ S \text{ satisfies } C \ (\text{moreover, } S \text{ is a minimal substitution that satisfies } C). \] As a result, we have inferred the type

\[ S(t_7 \rightarrow t_6 \rightarrow t_5) = (t_6 \rightarrow t_6) \rightarrow t_6 \rightarrow t_6 \]
Constraints and polymorphism

- Syntax of an instance constraint:
  \[ \tau_1 \leq_M \tau \]

- Semantics with respect to a substitution \( S \):
  \[ S \text{ satisfies } (\tau_1 \leq_M \tau_2) \overset{\text{def}}{=} S\tau_1 \prec \text{generalize}(SM, S\tau_2) \]

- Example:
  \[ [t_1 := t_2, t_4 := t_5 \to t_5] \text{ satisfies } t_4 \leq_\emptyset t_1 \to t_2 \]
Constraints and polymorphism

- Syntax of an instance constraint:

\[ \tau_1 \leq_M \tau \]

- Semantics with respect to a substitution $S$:

$S$ satisfies $(\tau_1 \leq_M \tau_2)$ = def $S\tau_1 < \text{generalize}(SM, S\tau_2)$

- Example:

- $[t1 := t2, t4 := t5 \rightarrow t5]$ satisfies $t4 \leq_{\emptyset} t1 \rightarrow t2$

\[
\begin{align*}
A_1, C_1 \vdash_{\text{BU}} e_1 : \tau_1 & \quad A_2, C_2 \vdash_{\text{BU}} e_2 : \tau_2 \\
A_1 \cup A_2 \setminus x, C_1 \cup C_2 \cup \{\tau' \leq_M \tau_1 \mid x: \tau' \in A_2\} \vdash_{\text{BU}} \text{let } x = e_1 \text{ in } e_2 : \tau_2
\end{align*}
\]

[LET]$_{\text{BU}}$
Example

\[ \text{identity} = \text{let } i = \lambda x \rightarrow x \text{ in } i \ i \]

Constraints
Example

\[ \text{identity} = \text{let } i = \lambda x \rightarrow x \text{ in } i \, i \]

Constraints

\[ A = \{x : t_1\} \]

\[ t_1 \]

\[ \text{VAR}(x) \]

\[ \text{ABS}(x) \]

\[ \text{LET}(i) \]

\[ \text{APP} \]

\[ \text{VAR}(i) \]

\[ \text{VAR}(i) \]

\[ \text{VAR}(i) \]
Example

\[ \text{identity} = \text{let } i = \backslash x \rightarrow x \text{ in } i \; i \]

Constraints

\[ t_1 \equiv t_2 \]
Example

\[ \text{identity} = \text{let } i = \lambda x \rightarrow x \text{ in } i \ i \]

\[ \text{Constraints}\]
\[ t_1 \equiv t_2 \]
Example

\[
\text{identity} = \text{let } i = \lambda x \to x \text{ in } i \ i
\]

Constraints

\[
t_1 \equiv t_2
\]
Example

\[ \text{identity} = \textbf{let } i = \lambda x \rightarrow x \textbf{ in } i \ i \]
Example

\[
\text{identity} = \text{let } i = \lambda x \rightarrow x \text{ in } i 
\]

Constraints

\[
\begin{align*}
t_1 & \equiv t_2 \\
t_3 & \equiv t_4 \rightarrow t_5 \\
t_3 & \leq \emptyset t_2 \rightarrow t_1 \\
t_4 & \leq \emptyset t_2 \rightarrow t_1
\end{align*}
\]
Example

\[ \text{identity} = \textbf{let}\ i = \backslash x \to x\ \textbf{in}\ i\ i \]

\[ \text{let } i = x \to x \text{ in } i \]

\[ \begin{align*}
    \textbf{C} = & \begin{cases}
    t_1 & \equiv t_2 \\
    t_3 & \equiv t_4 \to t_5 \\
    t_3 & \leq_{\emptyset} t_2 \to t_1 \\
    t_4 & \leq_{\emptyset} t_2 \to t_1
    \end{cases} \\
    \textbf{S} = & \begin{cases}
    t_1 & := t_2 \\
    t_3 & := (t_6 \to t_6) \to t_6 \to t_6 \\
    t_4, t_5 & := t_6 \to t_6
    \end{cases}
\]

\[ \textbf{S} \text{ satisfies } \textbf{C} \ (\text{moreover, } \textbf{S} \text{ is a minimal substitution that satisfies } \textbf{C}) \]. As a result, we have inferred the type

\[ S(t_5) = t_6 \to t_6 \]

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

- Supports domain specific type error diagnosis
- Details of the type rules: see Bastiaan Heeren's PhD
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
For the program,

\( \text{allinc} = \lambda \text{xs} \rightarrow \text{map} (+1) \text{xs} \)

Helium generates \((-d\ \text{option})\)

\begin{align*}
  v5 & := \text{Inst}((\forall a\ b.\ (a \rightarrow b) \rightarrow \text{[a]} \rightarrow \text{[b]})) \\
  v9 & := \text{Inst}((\forall a.\ \text{Num} a \Rightarrow a \rightarrow a \rightarrow a)) \\
  \text{Int} & \equiv v10 : \{\text{literal}\} \\
  v9 & \equiv v8 \rightarrow v10 \rightarrow v7 : \{\text{infix application}\} \\
  v8 \rightarrow v7 & \equiv v6 : \{\text{left section}\} \\
  v3 & \equiv v11 : \{\text{variable}\} \\
  v5 & \equiv v6 \rightarrow v11 \rightarrow v4 : \{\text{application}\} \\
  v3 \rightarrow v4 & \equiv v2 : \{\text{lambda abstraction}\} \\
  v2 & \equiv v0 : \{\text{right-hand side}\} \\
  v0 & \equiv v1 : \{\text{right hand side}\} \\
  s22 & := \text{Gen}([], v1) : \{\text{Generalize allinc}\}
\end{align*}
Greedy constraint solver

Given a set of type constraints, the greedy constraint solver returns a substitution that satisfies these constraints, and a list of constraints 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.
  - \( W \): almost a post-order tree walk
  - \( M \): almost a pre-order tree walk
  - Bottom-up: ...
  - Pushing down type signatures: ...
Some constraints ‘belong’ to certain subexpressions:

\[
\begin{align*}
\mathcal{T}_c &= [c_2, c_3] \triangleright \{ c_1 ? \mathcal{T}_{c_1}, \mathcal{T}_{c_2}, \mathcal{T}_{c_3} \} \\
c_1 &= (\tau_1 \equiv \text{Bool}) \quad c_2 = (\tau_2 \equiv \beta) \quad c_3 = (\tau_3 \equiv \beta) \\
\mathcal{A}_1, \mathcal{T}_{c_1} &\vdash e_1 : \tau_1 \\
\mathcal{A}_2, \mathcal{T}_{c_2} &\vdash e_2 : \tau_2 \\
\mathcal{A}_3, \mathcal{T}_{c_3} &\vdash e_3 : \tau_3 \\
\mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \mathcal{A}_3, \mathcal{T}_c &\vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \beta
\end{align*}
\]

- $c_1$ 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 $\text{Bool}$ (do $c_1$ before $\mathcal{T}_{c_1}$).
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 \(xs : [4, 5, 6]\))

\[
\text{main} = xs : [4, 5, 6] \\
\text{where } \text{len} = \text{length } xs \\
xs = [1, 2, 3]
\]
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 ($3 \times Int$ versus $1 \times Bool$)
  - 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
, highParticipation 0.95 path
, phaseFilter
] ++

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

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

(2,9): Warning: Definition "len" is not used
(1,11): Type error in constructor
expression : :
  type : \text{a} \rightarrow [\text{a}] \rightarrow [\text{a}]
  expected type : [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{b}
probable fix : use ++ instead
Example: permute function arguments

\[\text{test :: Parser Char String}\]
\[\text{test} = \text{option } "" \text{ (token } "\text{hello!}"\text{)}\]

In Helium:

(2,8): Type error in application
expression : option "" (token "hello!")
term : option
type : Parser a b \rightarrow b \rightarrow Parser a b
does not match : String \rightarrow Parser Char String \rightarrow 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.

\[
\text{myfold } f \ z \ [ ] = [ z ] \\
\text{myfold } f \ z \ (x : xs) = \text{myfold } f \ (f \ z \ x) \ xs \\
\text{rev} = \text{myfold } (\text{flip } (:)) \ [ ] \\
\text{palin} :: \text{Eq } a \Rightarrow [ a ] \rightarrow \text{Bool} \\
\text{palin } xs = \text{rev } xs =\equiv xs
\]

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