Constraint-based Type Inference

Bastiaan Heeren

joint work with Jurriaan Hage and Doaitse Swierstra

June 19, 2003

Part I: Constraint-based type inference

- Introduction
- Bottom-up typing rules
- Equality constraints
- Polymorphism and instance constraints
- Constraint solving
- Summary

Example 1

.hs file

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

Is this program well typed?

Example 1

.hs file

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

Is this program well typed?

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

Student FP: "What did I do wrong?"

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

Example 2

.hs file

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

Is this program well typed?

Example 2

.hs file

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

Is this program well typed?

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

Example 2

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

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

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

Example 3

.hs file

```
main :: (Bool -> a) -> (a, a, a)
main = \f -> (f True, f False, f [])
```

Is this program well typed?

Example 3

.hs file

```
main :: (Bool -> a) -> (a, a, a)
main = \f -> (f True, f False, f [])
```

Is this program well typed?

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

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

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

Hindley/Milner type inference

$$\frac{\tau \prec \Gamma(x)}{\Gamma \vdash_{\mathrm{HM}} x : \tau} \quad [\textsc{Var}]_{\mathrm{HM}}$$

$$\frac{\Gamma \vdash_{\mathrm{HM}} e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash_{\mathrm{HM}} e_2 : \tau_1}{\Gamma \vdash_{\mathrm{HM}} e_1\ e_2 : \tau_2} \quad [\textsc{App}]_{\mathrm{HM}}$$

$$\frac{\Gamma \backslash x \cup \{x :: \tau_1\} \vdash_{\mathrm{HM}} e : \tau_2}{\Gamma \vdash_{\mathrm{HM}} \lambda x \rightarrow e : (\tau_1 \rightarrow \tau_2)} \quad [\textsc{Abs}]_{\mathrm{HM}}$$

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

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

Constraint-based type inference

- A basic operation for type inference is unification.
  Property: let S be *unify*(τ_1, τ_2), then $S\tau_1 = S\tau_2$

We can view unification of two types as a constraint.

Constraint-based type inference

- A basic operation for type inference is unification.
  Property: let S be *unify*(τ_1, τ_2), then $S\tau_1 = S\tau_2$

We can view unification of two types as a constraint.

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

Bottom-up typing rules

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

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

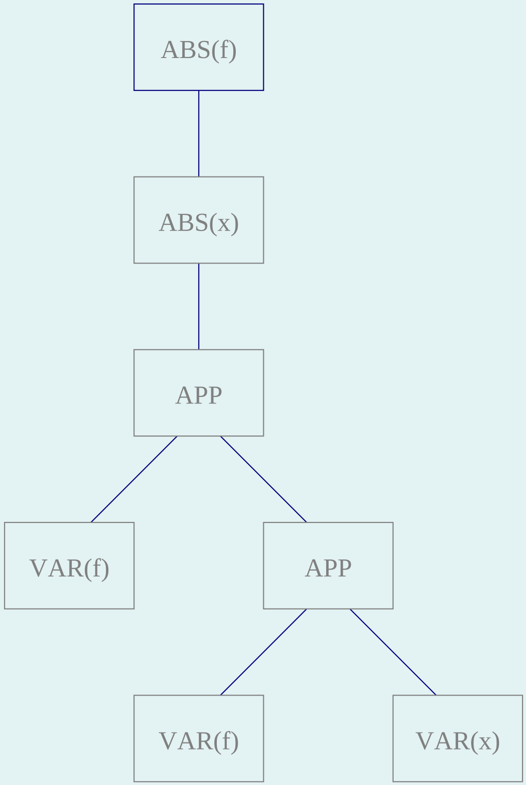
$$\frac{\mathcal{A},\ \mathcal{C}\ \vdash_{\mathrm{BU}}\ e:\tau}{\mathcal{A}\backslash x,\ \mathcal{C}\cup\{\tau'\equiv\beta\mid x:\tau'\in\mathcal{A}\}\ \vdash_{\mathrm{BU}}\ \lambda x\rightarrow e:(\beta\rightarrow\tau)} \qquad [\textsc{Abs}]_{\mathrm{BU}}$$

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

Example

.hs file

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

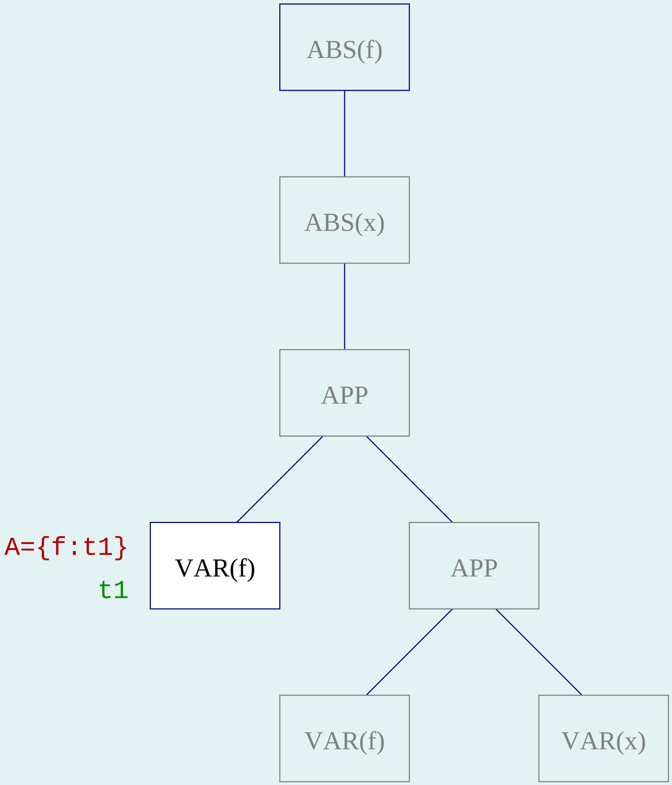


Constraints

Example

.hs file

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

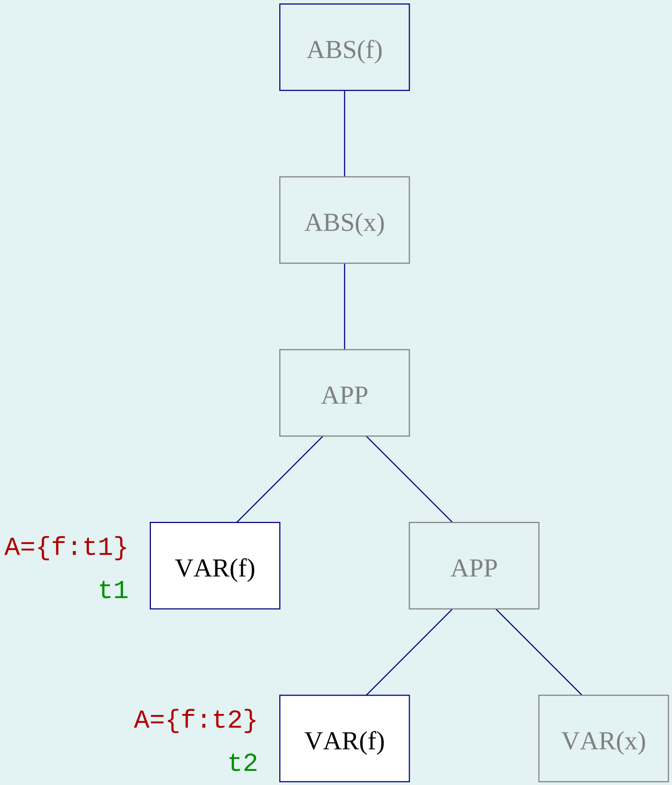


Constraints

Example

.hs file

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

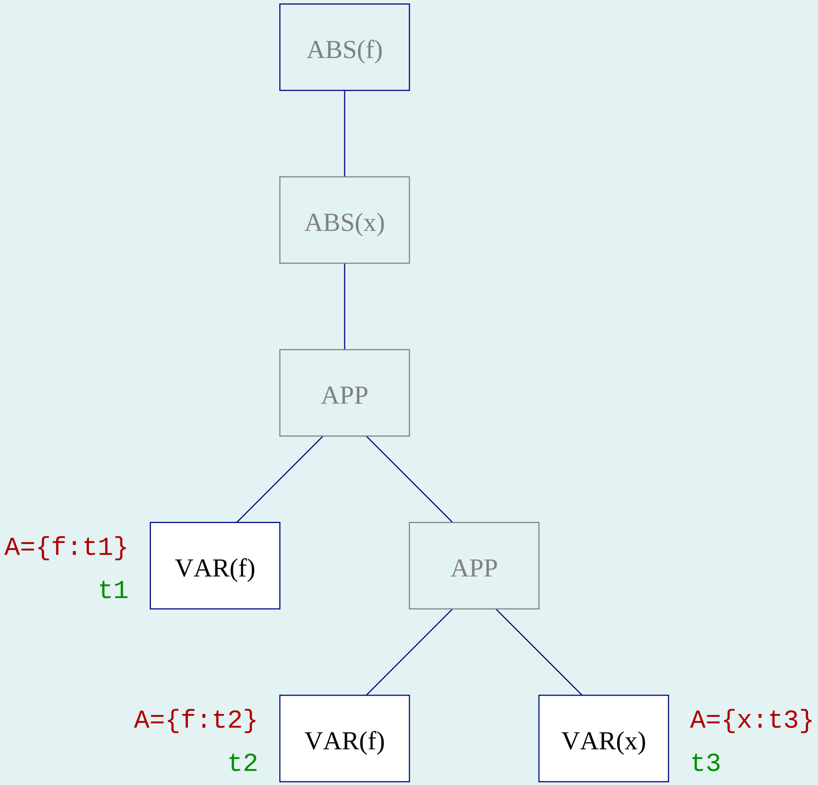


Constraints

Example

.hs file

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

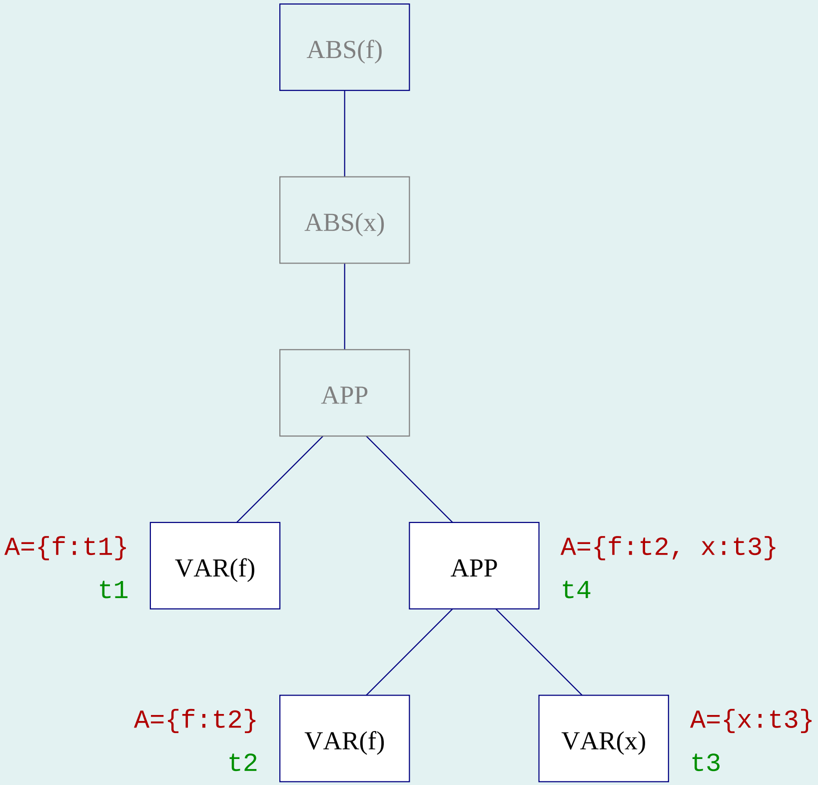


Constraints

Example

.hs file

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

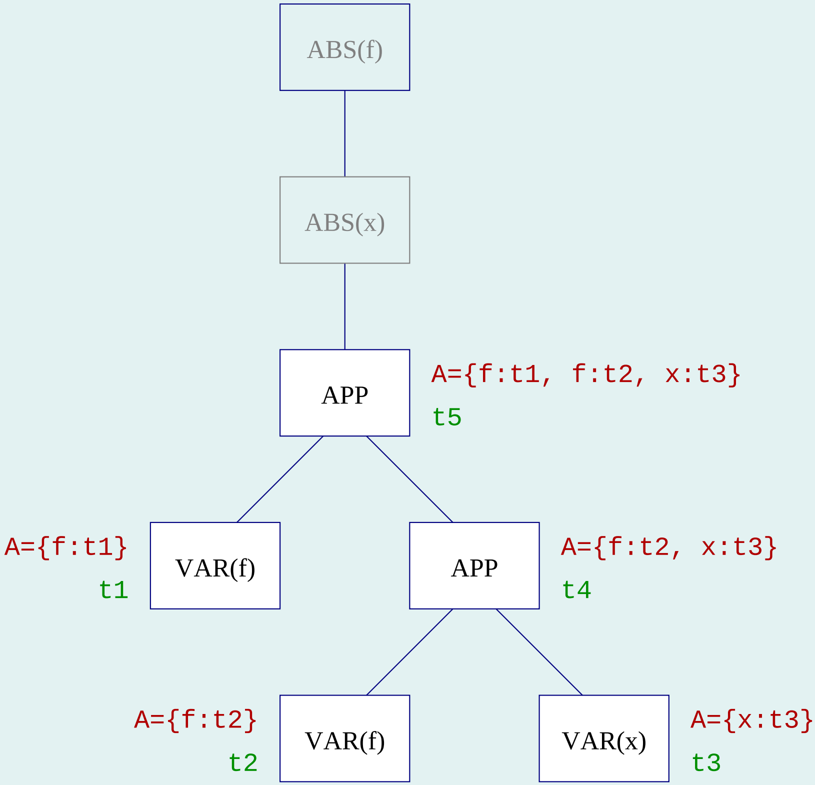


Constraints
t2 ≡ t3 -> t4

Example

.hs file

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

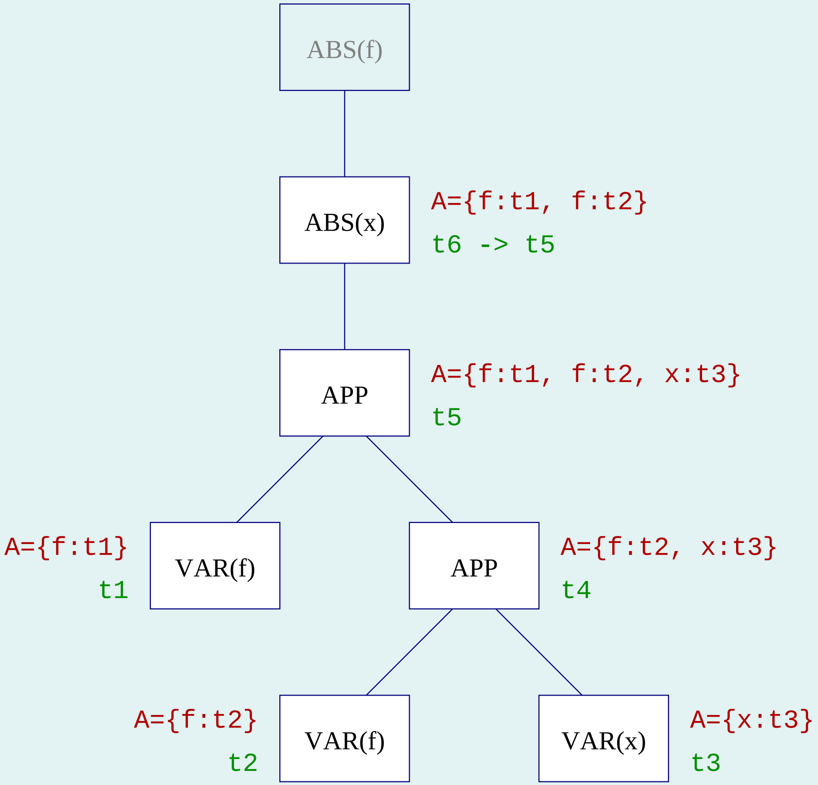


Constraints		
t2	≡	t3 -> t4
t1	≡	t4 -> t5

Example

.hs file

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

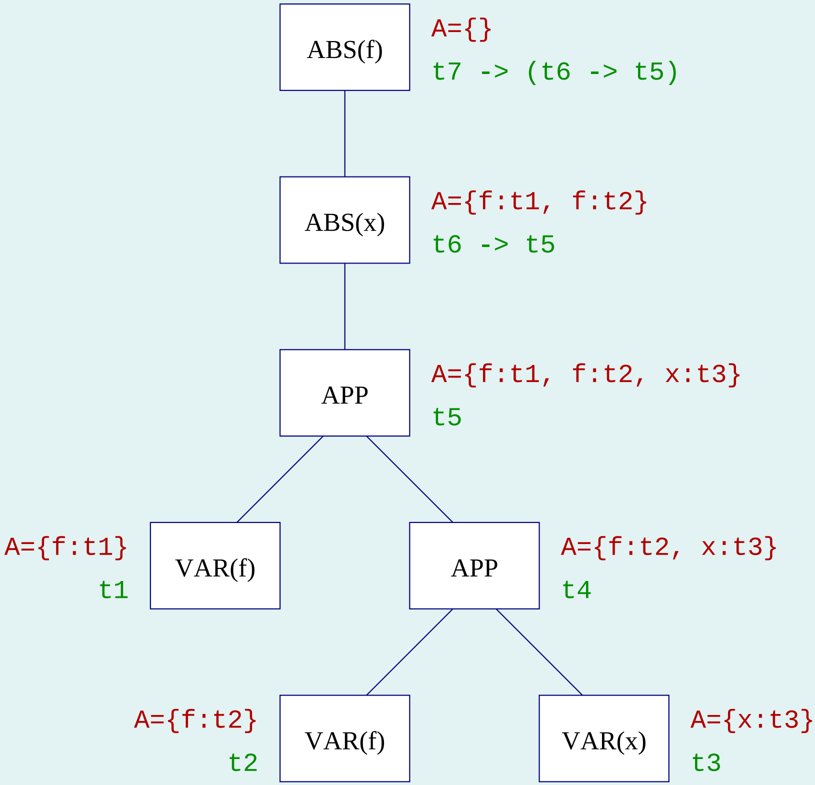


Constraints		
t2	≡	t3 -> t4
t1	≡	t4 -> t5
t3	≡	t6

Example

.hs file

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



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

Example

.hs file

```
twice = \f -> \x -> 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:

$$\tau_1 \leq_M \tau$$

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

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

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

Constraints and polymorphism

- Syntax of an instance constraint:

$$\tau_1 \leq_M \tau$$

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

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

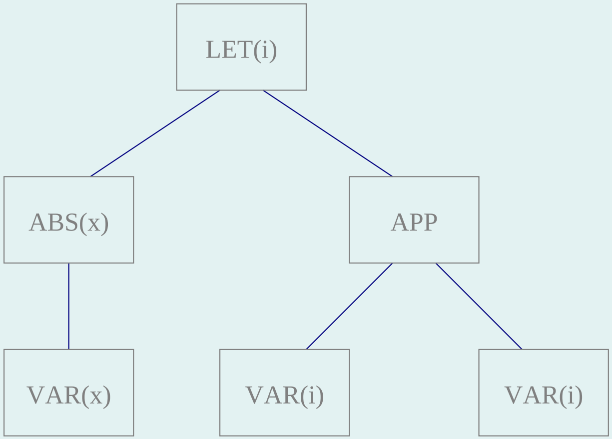
- Example:
  - $[$`t1 := t2, t4 := t5 -> t5`$]$ satisfies `t4` $\leq_\emptyset$ `t1 -> t2`

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

Example

.hs file

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

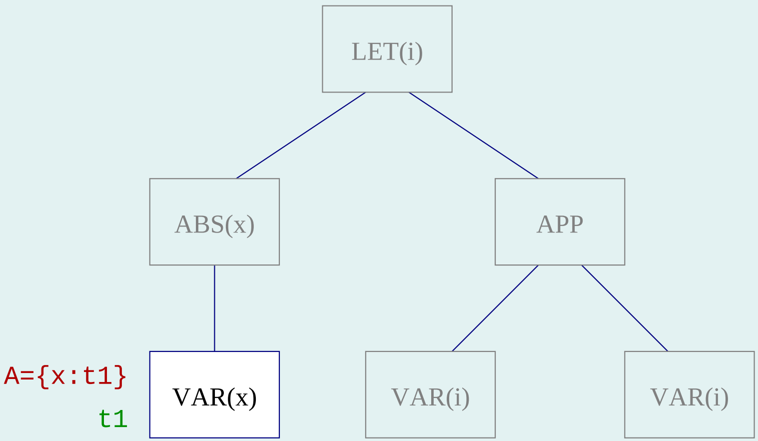


Constraints

Example

.hs file

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

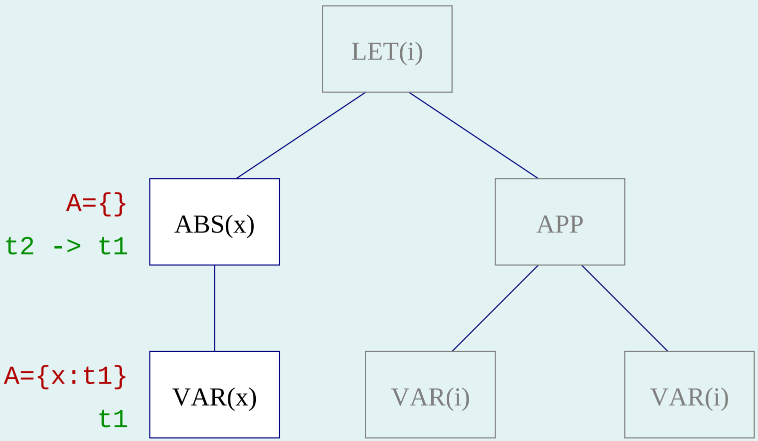


Constraints

Example

.hs file

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



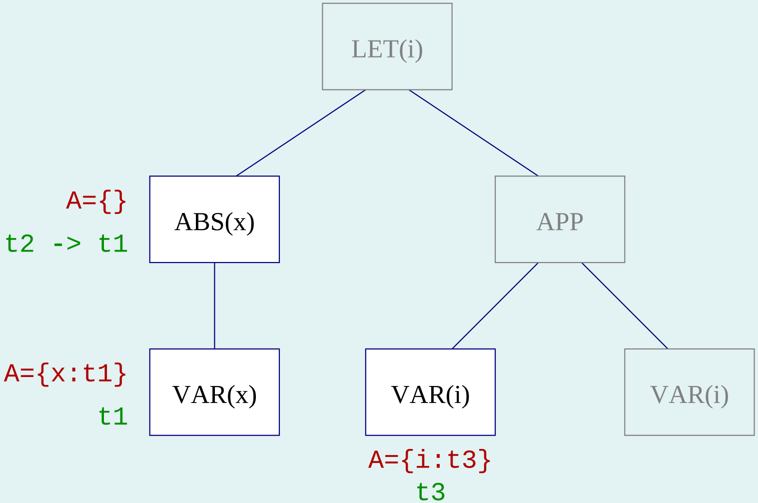
Constraints

t1 $\equiv$ t2

Example

.hs file

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

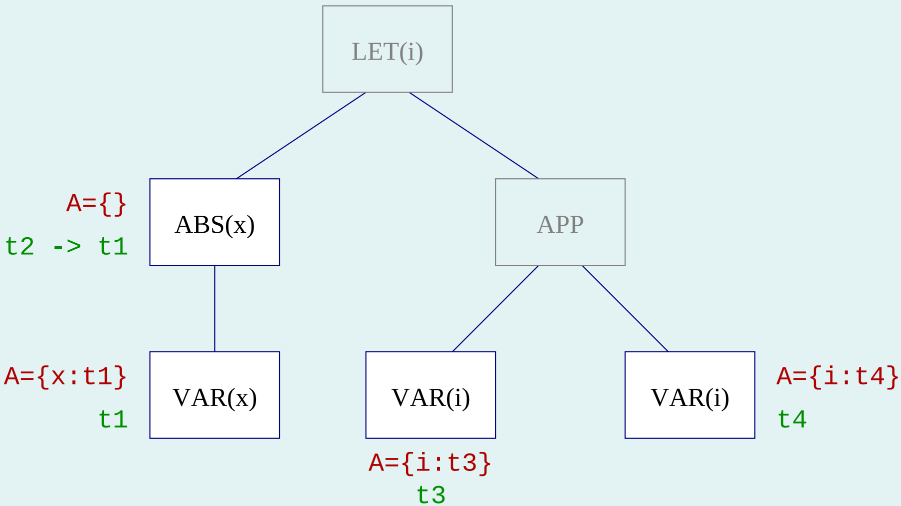


Constraints		
t1	$\equiv$	t2

Example

.hs file

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

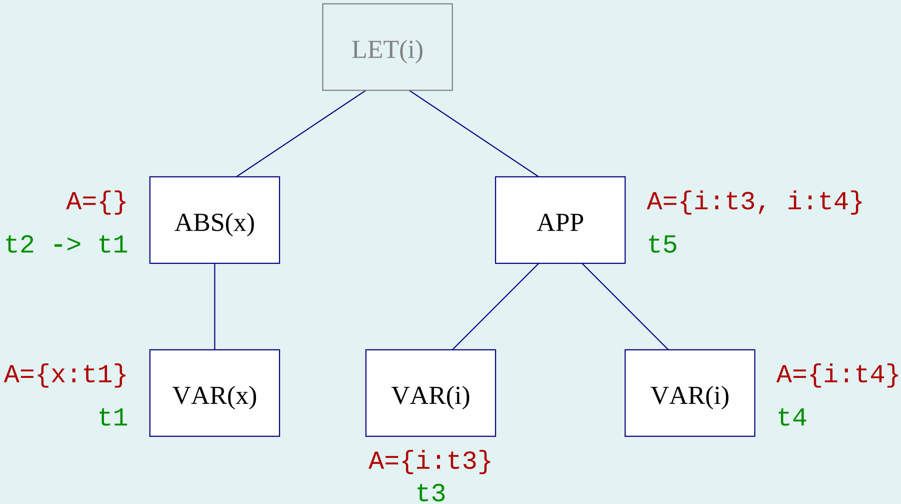


Constraints		
t1	$\equiv$	t2

Example

.hs file

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

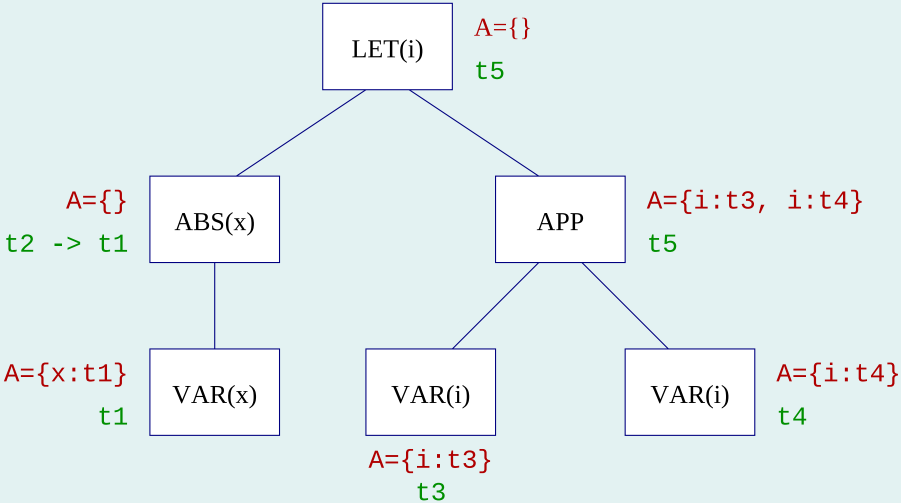


Constraints		
t1	$\equiv$	t2
t3	$\equiv$	t4 -> t5

Example

.hs file

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



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

Example

.hs file

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

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

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

for `identity`.

Greedy constraint solver

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

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

Ordering type constraints

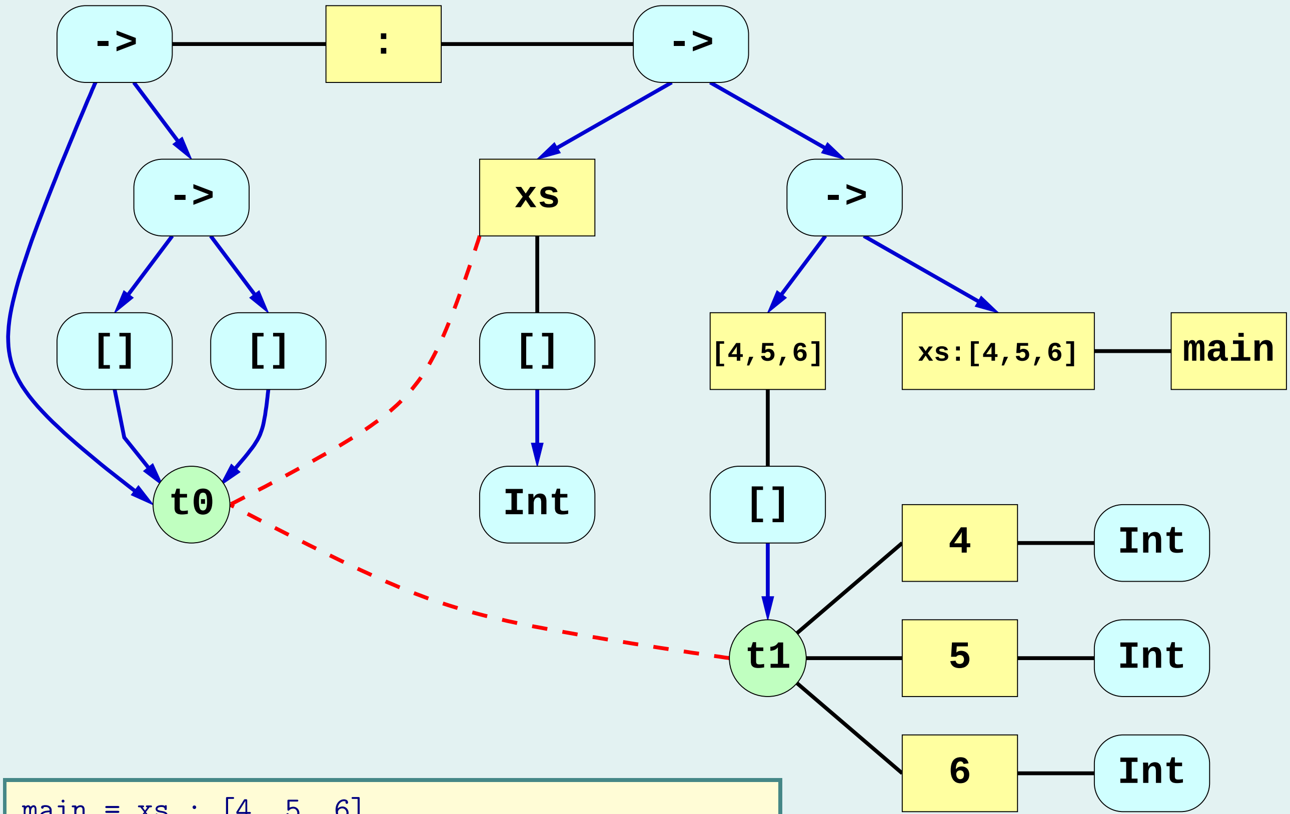
- One is free to choose the order in which the constraints should be considered by the greedy constraint solver. (Although there is a restriction for an implicit instance constraint)
- Instead of returning a list of constraints, return a constraint tree that follows the shape of the AST.
- A tree-walk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

Global constraint solver

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

- Advantages:
  - Global properties can be detected
  - A lot of information is available
  - The type inference process can be unbiased
  - It is easy to include new heuristics to spot common mistakes.
- Disadvantage:
  - Extra overhead makes this solver slower

Type graphs




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

Type graph heuristics

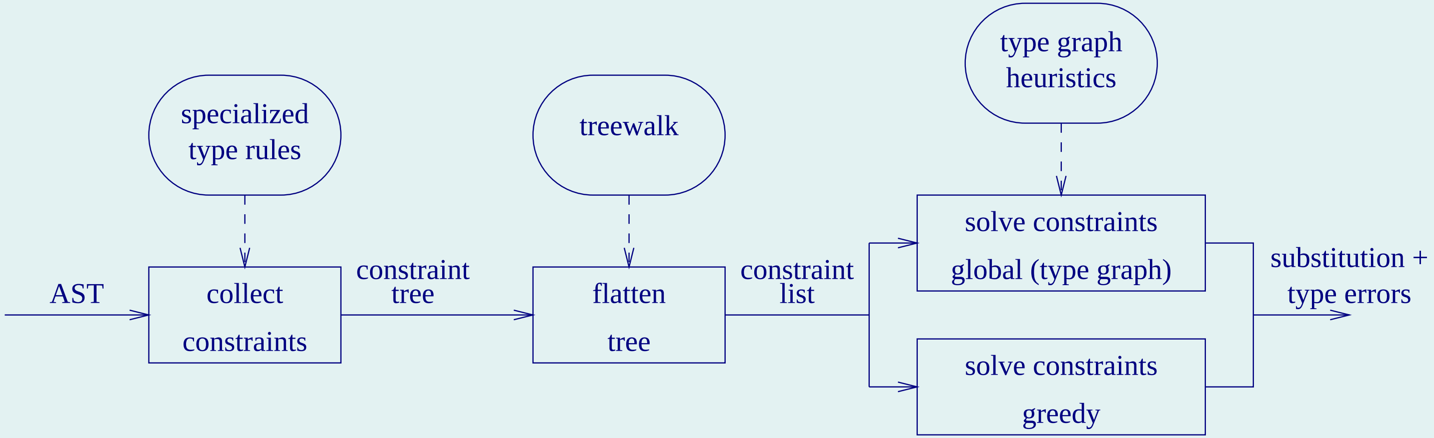
If a type graph contains an inconsistency, then heuristics help to choose which location is reported as type incorrect.

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

Summary

We have described a *parametric* type inferencer

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



Exercise 1: Constraint-based type inference

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

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

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

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

Part II: Type inference directives

- Introduction
- Directives
  - Specialized type rules
  - Phasing of type constraints
  - Sibling functions
  - Permuted function arguments
- Summary
- Conclusion

The problems

Type error messages suffer from the following problems.

1. **A fixed order of unification.** The order of traversal strongly influences the reported error site, and there is no way to depart from it.
2. **The size of the mentioned types.** Irrelevant parts are shown, and type synonyms are not always preserved.
3. **The standard format of type error messages.** Because of the general format of type error messages, the content is often not very poignant. Domain specific terms are not used.
4. **No anticipation for common mistakes.** Error messages focus on the problem, and not on how to fix the program. It is impossible to anticipate common pitfalls that exist.

The solution

Idea: supply type inference directives to the compiler to improve error reporting.

- For a given .hs file, a programmer may supply a .type file containing the directives
- The directives are automatically included when the module is imported

The solution

Idea: supply type inference directives to the compiler to improve error reporting.

- For a given .hs file, a programmer may supply a .type file containing the directives
- The directives are automatically included when the module is imported
- Examples:
  - Type directives in Prelude.type can help the students of an introductory course on functional programming

The solution

Idea: supply type inference directives to the compiler to improve error reporting.

- For a given .hs file, a programmer may supply a .type file containing the directives
- The directives are automatically included when the module is imported
- Examples:
  - Type directives in Prelude.type can help the students of an introductory course on functional programming
  - The designer of a (combinator) library can supply directives that are domain-specific

We use directives for a set of parser combinators as a running example.

Specializing a type rule (1/3)

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

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

Specializing a type rule (1/3)

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

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

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

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

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

Specializing a type rule (1/3)

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

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

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

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

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

$$\frac{\Gamma \ \vdash_{\mathrm{HM}} \ x : \tau_1 \rightarrow \tau_2 \quad \Gamma \ \vdash_{\mathrm{HM}} \ y : \mathit{Parser}\ \tau_3\ \tau_1}{\Gamma \ \vdash_{\mathrm{HM}} \ x <\$> y : \mathit{Parser}\ \tau_3\ \tau_2}$$

Specializing a type rule (2/3)

- Use equality constraints to make the restrictions that are imposed by the type rule explicit.
- We only consider type rules that have the same type environment Γ above and below the line.
- The type rule can only be used if the operator is unchanged. Type rules are invalidated by shadowing.

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

Specializing a type rule (2/3)

- Use equality constraints to make the restrictions that are imposed by the type rule explicit.
- We only consider type rules that have the same type environment Γ above and below the line.
- The type rule can only be used if the operator is unchanged. Type rules are invalidated by shadowing.

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

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

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

Specializing a type rule (3/3)

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

.type file

```
  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

.type file

```
  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

.hs file

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

This results in the following type error message:

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

Example

.hs file

```
test :: Parser Char String
test = map toUpper <$> "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

Solution: use error message attributes

Error message attributes

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

.type file

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


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

Example

.hs file

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

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

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

Implicit constraints

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

.type file

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

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

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

Order of the type constraints

- No knowledge about how the constraints are solved
- The earliest inconsistency is reported
- Each meta-variable represents a subtree for which also type constraints are collected. This constraint set can be explicitly mentioned in the type rule.

Order of the type constraints

- No knowledge about how the constraints are solved
- The earliest inconsistency is reported
- Each meta-variable represents a subtree for which also type constraints are collected. This constraint set can be explicitly mentioned in the type rule.

.type file

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

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

Soundness

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

- Because a mistake is easily made
- Invalid type rules are rejected when a Haskell file is compiled
- Type safety can still be guaranteed at run-time

Example

.type file

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

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

This specialized type rule is not restrictive enough:

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

Example

.type file

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

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

This specialized type rule is not restrictive enough:

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

Missing constraint:

```
a1 == a2 : function cannot be applied to parser's result
```

Another example

.type file

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

This specialized type rule is too restrictive:

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

Another example

.type file

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

This specialized type rule is too restrictive:

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

A correct specialized type rule:

.type file

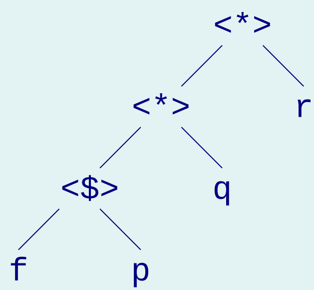
```
   x :: a -> b;    y :: Parser s a;
-------------------------------------
      x <$> y :: Parser s b;
```

AST versus conceptual structure

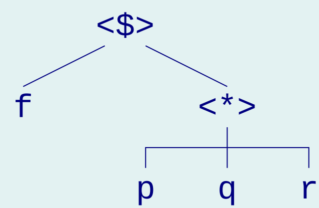
.hs file

```
f <$> p <*> q <*> r
```

- The associativity and priority of the parser operators are chosen to minimize the number of parentheses in a practical situation
- The inferencing process follows the shape of the abstract syntax tree closely
- The actual shape of an AST differs from the way a programmer interprets it



abstract syntax tree



conceptual structure

As a consequence, the reported error for an ill-typed expression involving these combinators can be counter-intuitive and misleading.

Phasing by example (1/2)

.hs file

```
test :: Parser Char String
test = (++) <$> token  "hello world"
            <*> symbol '!'
```

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.

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

Phasing by example (2/2)

- Hugs reports the following:

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

- The four step approach might result in:

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

Assigning phase numbers (1/2)

.type file

```
  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 cannot be applied to parser's result
b1 == b2 : parser has an incorrect result type
```

- The constraints in phase number i are solved before the constraint solver continues with the constraints of phase $i+1$
- The default phase number is 5

Assigning phase numbers (2/2)

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

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

.hs file

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

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

Anticipate common mistakes

One typical mistake is confusing two functions that are somehow related.

Examples:

- `curry` and `uncurry`
- `(:)` and `(++)`
- $(<\!*\!>)$ and $(<\!*)$

We will refer to such a pair of related functions as *siblings*.

Anticipate common mistakes

One typical mistake is confusing two functions that are somehow related.

Examples:

- `curry` and `uncurry`
- `(:)` and `(++)`
- $(<\!*\!>)$ and $(<\!*)$

We will refer to such a pair of related functions as *siblings*.

By declaring siblings in a .type file, the type inferencer will consider suggesting a *probable fix*.

.type file

```
siblings   <$> , <$
siblings   <*> , <*
```

Example

.hs file

```
data Expr     = Lambda Patterns Expr
type Patterns = [Pattern]
type Pattern  = String

pExpr :: Parser Token Expr
pExpr
  =  pAndPrioExpr
 <|> Lambda <$  pKey "\\"
            <*> many  pVarid
            <*  pKey "->"
            <*  pExpr         -- <* should be <*>
```

An extreme of concision is:

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

Permuting function arguments (1/2)

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

.hs file

```
test :: Parser Char String
test = option "" (token "hello!")
```

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

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

Permuting function arguments (2/2)

.hs file

```
test :: Parser Char String
test = option "" (token "hello!")
```

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

Summary

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

- Specialized type rules
- Phasing of type constraints
- Identification of sibling functions
- Testing for permuted function arguments

Summary

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

- Specialized type rules
- Phasing of type constraints
- Identification of sibling functions
- Testing for permuted function arguments

Results:

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

Conclusion

The major advantages of our approach can be summarized as follows.

- Type directives are supplied externally. As a result, no detailed knowledge of how the type inference process is implemented is necessary.
- Type directives can be concisely and easily specified by anyone familiar with type inference. Consequently, experimenting effectively with the type inference process becomes possible.
- The directives are automatically checked for soundness. The major advantage here is that the underlying type system remains unchanged, thus providing a firm basis for the extensions.
- For combinator libraries in particular, it becomes possible to report error messages which correspond more closely to the conceptual domain for which the library was developed.

Exercise 2: Specialized type rules

Example of a specialized type rule.

.type file

```
  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
```

The error messages can be refined with error message attributes.

.type file

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