

Helium and Type Inference Directives



Bastiaan Heeren (bastiaan@cs.uu.nl)
Utrecht University

Landelijke FP-dag 2004, January 9.

Overview

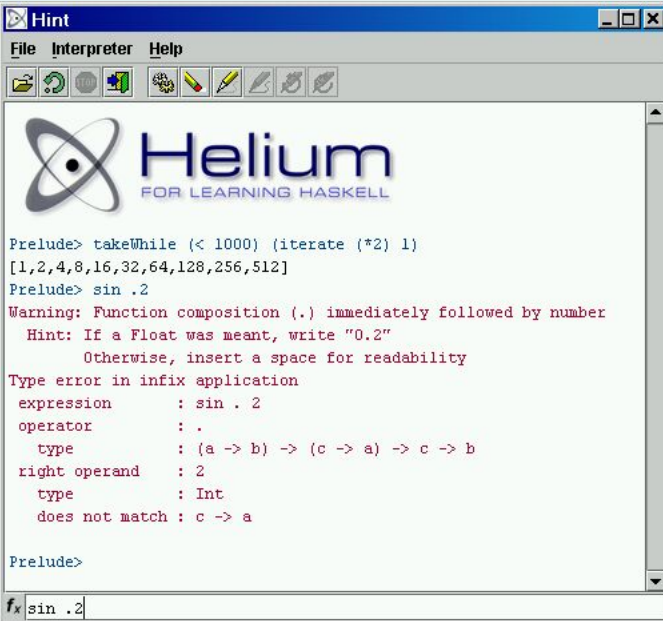
- ▶ Part 1: The Helium compiler
 - What changed since last year?
 - What will we do next?

- ▶ Part 2: Type inference directives
 - Introduction
 - Specialized type rules
 - Sibling functions
 - Conclusion

The Hint interpreter

(by Arie Middelkoop)

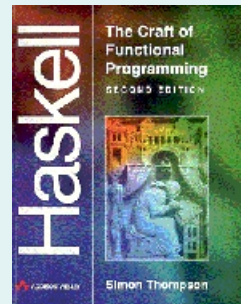
- ▶ A graphical interpreter (written in Java).
- ▶ Jump to the error location in an editor (line and column).
- ▶ Used in FP course (>100 students).



```
Hint
File Interpreter Help
[Icons]
Helium
FOR LEARNING HASKELL
Prelude> takeWhile (< 1000) (iterate (*2) 1)
[1,2,4,8,16,32,64,128,256,512]
Prelude> sin . 2
Warning: Function composition (.) immediately followed by number
Hint: If a Float was meant, write "0.2"
      Otherwise, insert a space for readability
Type error in infix application
expression      : sin . 2
operator        : .
type            : (a -> b) -> (c -> a) -> c -> b
right operand   : 2
type            : Int
does not match : c -> a
Prelude>
fx sin . 2
```

Documentation

- ▶ Tour of the Helium Prelude
- ▶ User Manuals for the compiler and for Hint
- ▶ Windows installer
- ▶ Overview of features in Haskell Workshop 2003 paper
- ▶ Addendum to use *The Craft of Functional Programming* with Helium (by Thompson).



More warnings

- ▶ Based on Thompson's catalogue of error messages (*Some common Hugs error messages and their causes*), we added more warnings to the compiler.

```
Fun x = 17
```

```
(1,1): Undefined constructor "Fun"
```

```
Hint: Use identifiers starting with a lower case letter to define a function  
or a variable
```

```
x = sin .2
```

```
(1,9): Warning: Function composition (.) immediately followed by number
```

```
Hint: If a Float was meant, write "0.2"
```

```
Otherwise, insert a space for readability
```

```
(1,10): Type error in infix application
```

```
expression      : sin . 2
```

```
operator        : .
```

```
type            : (a -> b) -> (c -> a) -> c -> b
```

```
right operand   : 2
```

```
type            : Int
```

```
does not match : c -> a
```

Pattern match warnings

(by Maarten Löffler)

```
main :: Bool -> Bool -> ()
main False _      = ()
main _            True = ()
```

```
(2,1): Warning: Missing pattern in function bindings:
      main True False = ...
```

► Similar warnings for:

- unreachable patterns
- possible fall throughs for guards

Function Recognition

(by Christof Douma)

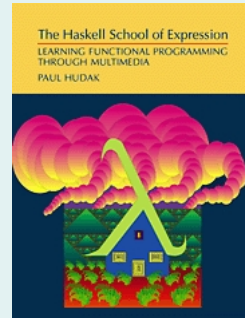
- ▶ Recognition of simple recursive functions in programs such as *map*, *concat* and *filter*. Idea: stimulate students to use these higher-order functions.

```
zonder n | null n      = []  
         | head n == 0 = zonder (tail n)  
         | True       = head n : zonder (tail n)
```

- ▶ We used the data set collected with the logger to test the usefulness of this feature. The number of recognized functions was substantial.
- ▶ Not (yet) part of the distribution.

Future work

- ▶ Test and release Helium with (simple) type classes (version 2.0).
 - The set of type classes is fixed and closed.
 - We keep supporting a Prelude without overloading.
- ▶ Write Hint interpreter with wxHaskell.
- ▶ Enhance Helium's module system.
- ▶ A simple GUI library to support Hudak's *Haskell School of Expression*, and for more appealing demos and laboratory exercises.



Part 2:

Type Inference Directives

Introduction

.hs file

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

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

Error message by Hugs:

```
ERROR "Example.hs":7 - Type error in application
*** Expression      : pAndPrioExpr <|> Lambda <$ pKey "\\\" <*>
                    many pVarid <*> pKey "->" <*> pExpr
*** Term           : pAndPrioExpr
*** Type          : Parser Token Expr
*** Does not match : [Token] -> [(Expr -> Expr), [Token]]
```

Introduction

.hs file

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

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

Error message by GHC:

Example.hs:7:

```
Couldn't match 'Expr' against 'Expr -> Expr'
  Expected type: [Token] -> [(Expr, [Token])]
  Inferred type: [Token] -> [(Expr -> Expr, [Token])]
In the expression:
  (((Lambda <$ (pKey "\\\")) <*> (many pVarid)) <*> (pKey "->"))
  <*> pExpr
In the second argument of '(<|>)', namely
  '(((Lambda <$ (pKey "\\\")) <*> (many pVarid)) <*> (pKey "->"))
  <*> pExpr'
```

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.

Type inference directives

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
- ▶ Implemented for the Helium compiler

(<http://www.cs.uu.nl/helium/>)

Type inference directives

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
- ▶ Implemented for the Helium compiler

(<http://www.cs.uu.nl/helium/>)

- ▶ Examples:
 - Type inference directives in Prelude.type can help the students of an introductory course on functional programming
 - The designer of a (combinator) library can supply directives so that type error messages become domain-specific
- ▶ We use directives for a set of parser combinators as a running example

Specialized type rules

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } s \ a \rightarrow \text{Parser } s \ b$

► A specialized type rule

$$\frac{\Gamma \vdash x : a \rightarrow b \quad \Gamma \vdash y : \text{Parser } s \ a}{\Gamma \vdash x \langle \$ \rangle y : \text{Parser } s \ b}$$

Specialized type rules

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } s \ a \rightarrow \text{Parser } s \ b$

► A specialized type rule

$$\frac{\Gamma \vdash x : a \rightarrow b \quad \Gamma \vdash y : \text{Parser } s \ a}{\Gamma \vdash x \langle \$ \rangle y : \text{Parser } s \ b}$$

► ...with type constraints

$$\frac{x : \tau_1 \quad y : \tau_2}{x \langle \$ \rangle y : \tau_3} \quad \left\{ \begin{array}{l} \tau_1 \equiv a \rightarrow b \\ \tau_2 \equiv \text{Parser } s \ a \\ \tau_3 \equiv \text{Parser } s \ b \end{array} \right.$$

Specialized type rules

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } s \ a \rightarrow \text{Parser } s \ b$

- ▶ A specialized type rule

$$\frac{\Gamma \vdash x : a \rightarrow b \quad \Gamma \vdash y : \text{Parser } s \ a}{\Gamma \vdash x \langle \$ \rangle y : \text{Parser } s \ b}$$

- ▶ ...with type constraints

$$\frac{x : \tau_1 \quad y : \tau_2}{x \langle \$ \rangle y : \tau_3} \quad \left\{ \begin{array}{l} \tau_1 \equiv a \rightarrow b \\ \tau_2 \equiv \text{Parser } s \ a \\ \tau_3 \equiv \text{Parser } s \ b \end{array} \right.$$

- ▶ ...and “small” unification steps

$$\frac{x : \tau_1 \quad y : \tau_2}{x \langle \$ \rangle y : \tau_3} \quad \left\{ \begin{array}{ll} \tau_1 \equiv a_1 \rightarrow b_1 & s_1 \equiv s_2 \\ \tau_2 \equiv \text{Parser } s_1 \ a_2 & a_1 \equiv a_2 \\ \tau_3 \equiv \text{Parser } s_2 \ b_2 & b_1 \equiv b_2 \end{array} \right.$$

Syntax for a specialized type rule

$$\frac{x : \tau_1 \quad y : \tau_2}{x \langle \$ \rangle y : \tau_3} \quad \left\{ \begin{array}{ll} \tau_1 \equiv a_1 \rightarrow b_1 & s_1 \equiv s_2 \\ \tau_2 \equiv \text{Parser } s_1 a_2 & a_1 \equiv a_2 \\ \tau_3 \equiv \text{Parser } s_2 b_2 & b_1 \equiv b_2 \end{array} \right.$$

.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
```

Syntax for a specialized type rule

$$\frac{x : \tau_1 \quad y : \tau_2}{x \langle \$ \rangle y : \tau_3} \quad \left\{ \begin{array}{ll} \tau_1 \equiv a_1 \rightarrow b_1 & s_1 \equiv s_2 \\ \tau_2 \equiv \text{Parser } s_1 a_2 & a_1 \equiv a_2 \\ \tau_3 \equiv \text{Parser } s_2 b_2 & b_1 \equiv b_2 \end{array} \right.$$

.type file

```
x :: t1;   y :: t2;
```

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

t1 == a1 -> b1 : 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 result of parser

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.

Error message attributes

Type error messages can contain context specific information, such as:

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

Error message attributes

Type error messages can contain context specific information, such as:

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

.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@
...
```

Example

.hs file

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

Compiling this program results in the following type error message:

```
(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
```

Example

.hs file

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

Compiling this program results in the following type error message:

```
(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
```

- ▶ By analyzing the specialized type rules we can reject unsound rules.
- ▶ Type safety can still be guaranteed at run-time

Anticipating 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.

Anticipating 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 (from introduction)

.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:

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

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 inferencing. 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.
- ▶ It becomes possible to report error messages which correspond more closely to the conceptual domain of a combinator library.