Scripting the Type Inference Process

Bastiaan Heeren Jurriaan Hage S. Doaitse Swierstra

Institute of Information and Computing Sciences
Utrecht University

{bastiaan,jur,doaitse}@cs.uu.nl

August 25, 2003

Overview

- Introduction
- Type inference directives
  - Specialized type rules
  - Phasing of type constraints
  - Sibling functions
  - Permuting function arguments
- Conclusion

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

```
<$> :: (a -> b) -> Parser s a -> Parser s b
```

- A specialized type rule

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

Specialized type rules

```
<$> :: (a -> b) -> Parser s a -> Parser s b
```

- A specialized type rule

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

- ...with type constraints

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

Specialized type rules

```
<$> :: (a -> b) -> Parser s a -> Parser s b
```

- A specialized type rule

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

- ...with type constraints

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

- ...and "small" unification steps

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

Syntax for a specialized type rule

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

.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

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

.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
```

Soundness

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

- A mistake is easily made
- Type safety can still be guaranteed at run-time

.type file

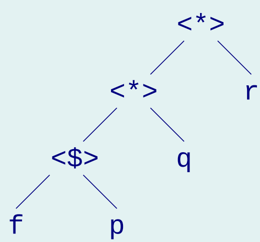
```
  x :: t1;     y :: t2;
-----------------------------
   x <$> y :: Parser s b;

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

- This type rule is not restrictive enough and thus rejected

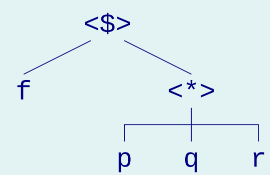
AST versus conceptual structure

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

- By design, associativities and priorities of the parser combinators minimize the number of parentheses in a practical situation.
- The inferencing process closely follows the shape of the abstract syntax tree, but the shape may differ from the way a programmer reads the expression.



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.

Assigning phase numbers

.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 result of parser
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

Phasing by example

.hs file

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

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])]
```

Phasing by example

.hs file

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

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])]
```

A phased approach might result in:

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

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
```

Permuting function arguments

.hs file

```
-- option :: Parser s a -> a -> Parser s a

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

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

```
ERROR "Permuted.hs":4 - 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

Example

.hs file

```
-- option :: Parser s a -> a -> Parser s a

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

- Improved error message:

```
(4,8): 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
```

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.

Summary and future work

	fixed order	size of types	standard format	no anticipation
specialized type rules	√	√	√	√
phasing	√	×	×	×
siblings	×	×	√	√
permuting	×	×	√	√

Work in progress:

- Designing type inference directives for the Helium Prelude
- Employment of directives in education
- Extend framework to work for type classes
- More support to design specialized type rules
- Extending the facilities for phasing