Scripting the Type Inference Process

Bastiaan Heeren Jurriaan Hage S. Doaitse Swierstra
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{bastiaan,jur,doaitse}@cs.uu.nl

Abstract

To improve the quality of type error messages in functional programming languages, we propose four techniques which influence the behaviour of constraint-based type inference processes. These techniques take the form of externally supplied type inference directives, precluding the need to make any changes to the compiler. A second advantage is that the directives are automatically checked for soundness with respect to the underlying type system. We show how the techniques can be used to improve the type error messages reported for a combinator library. More specifically, how they can help to generate error messages which are conceptually closer to the domain for which the library was developed. The techniques have all been incorporated in the `Helium` compiler, which implements a large subset of `Haskell`.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Applicative (Functional) Programming; D.3.4 [**Programming Languages**]: Processors—*debuggers*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms

Languages

Keywords

constraints, type inference, type errors, directives, domain-specific programming

1 Introduction

The important role of type systems in modern, higher-order, functional languages such as `Haskell` and `ML` is well-established. Type systems not only guide the novice programmer by pointing out errors at compile-time, but they are equally indispensable to the advanced programmer, when writing complex programs.

Unfortunately, clarity and conciseness are often lacking in the type errors reported by modern compilers. In addition, it is often not apparent what modifications are needed to fix an ill-typed program. For example, when using standard inference algorithms, the reported error site and its actual source can be far apart. As a result, the beginning programmer is likely to be discouraged from programming in a functional language, and may see the rejection of programs as a nuisance instead of a blessing. The experienced user might not look at the messages at all.

The problem is exacerbated in the case of combinator languages. Combinator languages are a means of defining domain specific languages embedded within an existing programming language, using the abstraction facilities present in the latter. However, since the domain specific extensions are mapped to constructs present in the underlying language, all type errors are reported in terms of the host language, and not in terms of concepts from the combinator library. In addition, the developer of a combinator library may be aware of various mistakes which users of the library can make, something which he can explain in the documentation for the library, but which he cannot make part of the library itself.

We have identified the following problems that are inherent to commonly used type inference algorithms.

i. *A fixed order of unification:* Typically, the type inferencer traverses a program in a fixed order, and this order strongly influences the reported error site. Moreover, a type inferencer checks type correctness for a given construct, say function application, in a uniform way. However, for some function applications it might be worthwhile to be able to depart from this fixed order. To overcome this problem, it should be possible to override the order in which types are inferred by delaying certain unifications or changing the order in which subexpressions are visited.

ii. *The size of the mentioned types:* Often, a substantial part of the types shown in a type error message is not relevant to the actual problem. Instead, it only distracts the programmer and makes the message unnecessarily verbose. The preservation of type synonyms where possible reduces the impact of this problem.

iii. *The standard format of type error messages:* Because of the general format of type error messages, the content is often not very poignant. Specialized explanations for type errors arising from specific functions would be an improvement for the

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

following reasons. Firstly, domain specific terms can be used in explanations, increasing the level of abstraction. Secondly, depending on the complexity of the problem and the expected skills of the intended users, one could vary the verbosity of the error message.

iv. *No anticipation for common mistakes:* Often, the designer of a library is aware of the common mistakes and pitfalls that occur when using the library. The inability to anticipate these pitfalls is regretful. This might take the form of providing additional hints, remarks and suggested fixes that come with a type error.

By way of example, consider the set of parser combinators by Swierstra [14], which we believe is representative for most (combinator) libraries. Figure 1 contains a type incorrect program fragment to parse a lambda abstraction using the parser combinators (see Section 2 for a description of the latter). In the example, an expression is either an expression with operators which have the same priority as the boolean `and` operator (`pAndPrioExpr`), or the expression is a lambda abstraction which may abstract a number of variables.[1] The most likely error is indicated in comments in the example itself: the subexpression `<* pExpr` indicates that an expression, the body of the lambda, should be parsed at this point, but that the result (in this case an `Expr`) should immediately be discarded (as is the case with the `"\\"` and the `"->"`). As a result, the constructor `Lambda` is only applied to a list of patterns, so that the second alternative in `pExpr` has result type `Expr -> Expr`. However, the first alternative of `pExpr` yields a parser with result type `Expr` and here the conflict surfaces.

Consider the type errors reported by `Hugs` in Figure 2 , and by `GHC` in Figure 3. Comparing the two messages with the third message which was generated using our techniques, we note the following.

- It refers to parsers and not to more general terms such as functions or applications. It has become domain specific and can solve problem (iii) to a large extent.
- It only refers to the incompatible result types.
- The third message includes precise location information.

In addition, `Hugs` displays the unfolded non-matching type, but it does not unfold the type of `pAndPrioExpr`, which makes the difference between the two seem even larger. This is an instance of problem (ii). Note that if the Parser type had been defined using `newtype` (or `data`), then this problem would not have occurred. However, a consequence of such a definition is that wherever parsers are used, the programmer has to pack and unpack them. This effectively puts the burden on him and not on the compiler.

In the case of `GHC`, the error message explicitly states what the non-matching parts in the types are. On the other hand, it is not evident that the expected type originates from the explicit type signature for `pExpr`. The expressions in the error message include parentheses which are not part of the original source. It is striking that the same long expression is listed twice, which makes the message more verbose without adding any information.

Note that if, instead of applying the constructor `Lambda` to the result of the parser, we immediately apply an arbitrary semantic function, then the messages generated by `Hugs` and `GHC` become more complex. Again, we see an instance of problem (ii).

[1] We assume here that we are dealing with a list of tokens, and not characters, but this is no essential difference.

In Figure 5 we have shown the absolute extreme of concision: when our facility for specifying so-called sibling functions is used, the inferencer discovers that replacing $<*$ by the similar combinator $<*>$ yields a type correct program. The fact that $<*>$ and $<*$ are siblings is specified by the programmer of the library, usually because his experiences are that these two are often mixed up. This kind of facility helps to alleviate problem (iv). Please note that it is better to generate a standard type error here, and to give the "probable fix" as a hint. There is always the possibility that the probable fix is not the correct one, and then the user needs more information.

In the light of the problems just described, we present an approach that can be used to overcome the problems for a given library. An important feature of our approach is that the programmer of a such a library need not be familiar with the type inference process as it is implemented in the compiler: everything can be specified by means of a collection of type inference directives, which can then be distributed as part of the combinator library. If necessary, a user of such a library may adapt the directives to his own liking. An additional benefit is that the type inference directives can be automatically checked for soundness with respect to the type inference algorithm present in the compiler.

We have implemented our techniques in the Helium compiler [8], which implements a large subset of `Haskell`; the most notable omission is that of type classes. This compiler was constructed at Utrecht University with a focus on generating high quality error messages, and is used in an introductory course on functional programming. We expect that our techniques can be quite easily incorporated into compilers which have a constraint-based type inference process with a clear separation between generating and solving constraints.

The paper is structured as follows. After a minimal introduction to the parser combinator library in Section 2, we propose solutions for the four problems just identified (Section 3), and describe how to specify the necessary type inference directives. In Section 4, we explain some technical details that are essential in making this work. Finally, Section 5 discusses related work, and Section 6 concludes this paper.

2 Preliminaries

In this section we briefly describe the parser combinators which we use in our examples. Whenever necessary we explain why they are defined as they are, especially where this departs from what might be the more intuitive way of defining them. The parser combinators [14] were defined to correspond as closely as possible to (E)BNF grammars, although the complete library provides combinators for many other often occurring patterns.

Consider for the remainder of this section the `Haskell` declarations in Figure 6. The type `Parser s a` describes a parser which takes a list of symbols of type `s` and delivers a list of possible results (to cope with failing and ambiguous parsings). A result consists of (the semantics of) whatever was parsed at this point, which is of type `a`, and the remainder of the input.

The main combinators in our language are the operators $<*>$, $<|>$ and $<\$>$, and the parser `sym`. The first of these is the sequential composition of two parsers, the second denotes the choice between two parsers. We may recognize terminal symbols by means of the `sym` parser, which takes the symbol to recognize as its parameter. To be able to parse a symbol, we have to be able to test for equal-

```
data Expr     = Lambda Patterns Expr    -- can contain more alternatives
type Patterns = [Pattern]
type Pattern  = String

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

Figure 1. Type incorrect program

```
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])]
```

Figure 2. Hugs, version November 2002

```
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'
```

Figure 3. The Glasgow Haskell Compiler, version 5.04.3

```
Compiling Example.hs
(7,6): The result types of the parsers in the operands of <|> don't match
  left parser   : pAndPrioExpr
    result type : Expr
  right parser  : Lambda <$ pKey "\\" <*> many pVarid <* pKey "->" <* pExpr
    result type : Expr -> Expr
```

Figure 4. Helium, version 1.1 (type rules extension)

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

Figure 5. Helium, version 1.1 (type rules extension and sibling functions)

```
infixl 7 <$>, <$
infixl 6 <*>, <*
infixr 4 <|>

type Parser s a = [s] -> [(a,[s])]

<$> :: (a -> b) -> Parser s a -> Parser s b
<*> :: Parser s (a -> b) -> Parser s a -> Parser s b
<|> :: Parser s a        -> Parser s a -> Parser s a
<$  :: a                 -> Parser s b -> Parser s a
<*  :: Parser s a        -> Parser s b -> Parser s a

sym    :: (s -> s -> Bool) -> s   -> Parser s s
tok    :: (s -> s -> Bool) -> [s] -> Parser s [s]
option :: Parser s a -> a -> Parser s a
many   :: Parser s a -> Parser s [a]
symbol :: Char  -> Parser Char Char
token  :: String -> Parser Char String
```

Figure 6. The parser combinators

ity. In Haskell this is usually done by way of type classes, but we have chosen to include the predicate explicitly. For the techniques described in this paper, this makes no essential difference. For notational convencience we introduce symbol which works on characters.

We now have all we need to implement BNF grammars. For instance the production $P \rightarrow QR \mid a$ might be transformed to the Haskell expression

```
pP = pQ <*> pR <|> symbol 'a'.
```

The type of each of the parsers pQ <*> pR and symbol 'a' must of course be the same, as evidenced by the type of <|>. The type of the combinator <*> specifies that the first of the two parsers in the composition delivers a function which can be applied to the result of the second parser. An alternative would be to return the tupled results, but this has the drawback that, with a longer sequence of parsers, we obtain deeply nested pairs which the semantic functions have to unpack.

Usually, one uses the <$> combinator to deal with the results that come from a sequence of parsers:

```
pP = f <$> pQ <*> pR <|> symbol 'a',
```

where f is a function which takes a value of the result type of pQ and a second parameter which has the result type of pR, delivering a value which should have type Char (because of the symbol 'a' parser). The operator <$> has a higher priority than <*>, which means that the first alternative for <|> should be read as ((f <$> pQ) <*> pR). A basic property of <$> is that it behaves like function application, consuming its arguments one by one. However, this is generally not the way parser builders read their parsers. Usually, the parser is constructed first, and the semantic functions are added afterwards. This difference in perception is one of the sources of confusion when people use the parser combinators. In the following section we describe how our techniques can help to alleviate this problem.

For <$> and <*>, we have variants <$ and <*, which discard the result of their right operand. This is useful when what is parsed needs only to be recognized, but is not used later on. An example of this can be found in the code of Figure 1, which throws away whatever comes out of either pKey application. This means that we can simply apply the constructor Lambda, instead of a function which takes four parameters and throws two of them away.

In the same example, we see an application of the many combinator which recognizes a list of things, in this case pVarIds. Here, pVarId is a predefined parser which recognizes an identifier. The pKey combinator is used to recognize keywords and reserved operators (which have been tokenized already so we do not have to deal with whitespace). Finally, the token combinator is a combination of many and symbol, and the option combinator is used when a parser can recognize the empty word. For example,

```
option (token "hello!") ""
```

either recognizes the string "hello!" and returns it, or it succeeds without consuming any input. In the latter case, the option parser returns the empty string instead. Note that in Haskell strings are defined to be equivalent to lists of characters.

3 Type inference directives

This section describes four techniques that help to improve the quality of type error messages. To start with, we present a notation to define your own type rules with their specialized type error messages. Next we explain why flexibility in the order of unification is absolutely necessary in order to get appropriate messages. The final two examples deal with common mistakes.

We have implemented all four techniques in the Helium compiler [8]. In Helium, the directives for a module Name.hs are collected in the file Name.type, which is automatically loaded when Name.hs is imported.

3.1 Specialized type rules

This section describes how to write specialized type rules and explains how this influences the error reporting in case a type rule fails. There are serious disadvantages to incorporating these rules directly in a type inferencer. It requires training and experience to extend an existing type inferencer, and it implies a loss of compositionality and maintainability of the implemented type rule. Since the correctness of a type inferencer is quite a subtle issue, it is no longer possible to guarantee that the underlying type system remains unchanged. Instead, we follow a different approach in which the type rules can be specified externally. This makes it relatively easy to specify a type rule and to experiment with it without having to change any code of the type inference engine.

Specializing a type rule

Let us take a closer look at a traditional type inference rule for infix application. An infix application is type correct if the types of the two operands fit the type of the operator. Clearly, infix application is nothing but syntactic sugar for normal prefix function application. Applying the type rule for function application twice in succession results in the following.

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

Here, $\Gamma \vdash_{\text{HM}} e : \tau$ means that under the type environment Γ we can assign type τ to expression e [4]. Instead of using this general type rule to deal with infix applications, one could come up with a more specific type rule for a particular operator, for instance <$>. Let <$> be part of the type environment Γ, and have type signature

$(a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b$. Then we can write down the following specialized type rule.

$$\frac{\Gamma \vdash_{\text{HM}} x : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\text{HM}} y : \mathit{Parser}\ \tau_3\ \tau_1}{\Gamma \vdash_{\text{HM}} x <\$> y : \mathit{Parser}\ \tau_3\ \tau_2}$$

If we encounter a local definition of <$>, then the above type rule will not be used within the scope of that local definition, even if the new definition of <$> has the same type as the old one. To avoid confusion, we only want to use the type rule if the very same operator, here <$>, is used.

The type rule does not adjust the scope, as can be concluded from the fact that the same type environment Γ is used above and below the line. In the rest of this paper we will only consider specialized type rules with this property. This limitation is necessary to avoid complications with monomorphic and polymorphic types. Since the type environment remains unchanged, we will omit it from now on.

In general, a type rule contains a number of constraints, on each of which a type inferencer may fail. For instance, the inferred types for the two operands of <$> are restricted to have a specific shape (a function type and a *Parser* type), the relations between the three type variables constrain the inferred types even further and, lastly, the type in the conclusion must fit into its context. To obtain a better understanding why some inferred types may be inconsistent with this type rule, let us reformulate the type rule to make the type constraints more explicit.

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

An equality constraint, written $\tau \equiv \tau'$, can be thought of as the unification of two types. Algorithms that determine the most general unifier of two types are well understood.

In addition to the type variables introduced in the type rule, three more type variables are introduced in the constraint set, namely a, b, and s. The order in which the constraints are solved is irrelevant for the success or failure of the type inference process. However, the order chosen does determine where the type inferencer first notices an inconsistency. Typically, the order is determined by the type inference algorithm that one prefers, e.g., the standard bottom-up algorithm $\mathcal{W}$ [4] or the folklore top-down algorithm $\mathcal{M}$ [10]. To acquire additional information, we split up each constraint in a number of more basic type constraints. The idea of these small unification-steps is the following: for a type constraint that cannot be satisfied, the compiler can produce a more specific and detailed error message. The example now becomes

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

The definition of a type rule, as included in a `.type` file, consists of two parts, namely a deduction rule and a list of constraints. The deduction rule consists of premises, which occur above the line, and a conclusion below the line. A premise consists of a single meta-variable, which matches with every possible expression, and a type. On the other hand, the conclusion may contain an arbitrary expression, except that lambda abstractions and let expressions are not allowed, because they modify the type environment. There is no restriction on the types in the premises and the conclusion. Below the deduction rule, the programmer can list a number of equality constraints between types. Each of these is followed by a corresponding error message.

Example 1. We present a special type rule for the <$> combinator. Each of the constraints is specified with an error message that is reported if the constraint cannot be satisfied. The order in which the constraints are listed determines the order in which they shall be considered during the type inference process.

```
  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
```

Now take a look at the following function definition, which is clearly ill-typed.

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

Because it is pretty obvious which of the six constraints is violated here (the right operand of <$> is not a parser, hence, `t2` ≡ `Parser s1 a2` cannot be satisfied), the following error is reported.

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

Note that this type error message is still not too helpful since important context specific information is missing, such as the location of the error, pretty-printed parts of the program, and conflicting types. To overcome this problem, we use attributes in the specification of error messages.

Error message attributes

A fixed error message for each constraint is too simplistic. The main focus of a message should be the contradicting types that caused the unification algorithm to fail. To construct a clear and concise message, one typically needs the following information.

- *The inferred types of the subexpressions:* One should be able to refer to the actual type of a type variable that is mentioned in either the type rule or the constraint set. In the special case that a subexpression is a single identifier which is assigned a polymorphic type, then we prefer to display this generalized type instead of simply using the instantiated type.
- *A pretty-printed version of the expression and its subexpressions:* This should resemble the actual code as closely as possible, and should (preferably) fit on a single line.
- *Position and range information:* This also includes the name and location of the source file at hand.

Example 2. To improve the error message of Example 1, we replace the annotation of the type constraint

```
t2 == Parser s1 a2 :  right operand is not a parser
```

by the following error message, which contains attributes.

```
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@
```

In the error message, the expression in the conclusion is called `expr`. We can access its attributes by using the familiar dot notation, and surrounding it by `@` signs. For example, `@expr.pos@` refers to the position of `expr` in the program source. Similarly, `pp` gives a pretty printed version of the code.

The specification of a type constraint and its type error message is layout-sensitive: the first character of the error report (which is a '@' in the example above) determines the level of indentation. The definition of the error report stops at the first line which is indented less. As a result, the error report for the definition of `test` in Example 1 now becomes:

```
(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
```

For a given expression (occurring in the conclusion of a type rule), the number of type constraints can be quite large. We do not want to force the user to write out all these constraints and give corresponding type error messages. For this reason, the user is allowed to move some constraints from the list below the type rule to the type rule itself, as we illustrate in the next example.

Example 3. We continue with Example 1. Because we prefer not to give special error messages for the case that the result type is not a parser, we may as well give the following type rule.

```
   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
```

At this point, only three of the original type constraints remain. If any of the "removed" constraints contributes to an inconsistency, then a standard error message will be generated. These constraints will be considered before the explicitly listed constraints.

Order of the type constraints

In the type rule specifications we have so far only listed the constraints for that rule and the order in which they should be considered. In principle, we do not assume that we know anything about how the type inferencer solves the constraints. The only thing a type rule specifies is that if two of the constraints contribute to an inconsistency, then the first of these will be considered to be the source of the error. An error report will be generated for this constraint, after which the type inference process continues.

The situation is not as simple as this. Each of the meta-variables in the rule corresponds to a subtree of the abstract syntax tree for which sets of constraints are generated. How should the constraints of the current type rule be ordered with respect to these constraints?

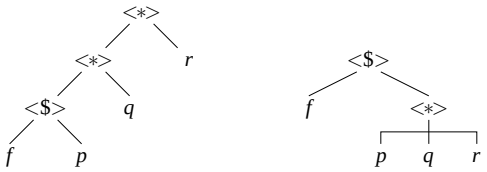

Figure 7. Abstract syntax tree (left) compared with the conceptual structure (right)

Example 4. If we want the constraints generated by the subexpression `y` to be considered after the constraint `t1 == a1 -> b`, then we should change the type rule in Example 3 to the following.

```
   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
```

Note that in this rule we have now explicitly stated at which point the constraints of `x` and `y` should be considered. By default, the sets of constraints are considered in the order of the corresponding meta-variables in the type rule, to be followed afterwards by the constraints listed below the type rule. Hence, we could have omitted `constraints x`.

By supplying type rules to the type inferencer we can adapt the behaviour of the type inference process. It is fair to assume that the extra type rules should not have an effect on the underlying type system, especially since an error in the specification of a type rule is easily made. We have made sure that user defined type rules that conflict with the default type rules are automatically rejected at compile time. A more elaborate discussion of this subject can be found in Section 4.2.

3.2 Phasing

Recall the motivation for the chosen priority and associativity of the <$> and <*> combinators: it allows us to combine the results of arbitrary many parsers with a single function in a way that feels natural for functional programmers, and such that the number of parentheses is minimized in a practical situation. However, the abstract syntax tree that is a consequence of this design principle differs considerably from the view that we suspect many users have of such an expression. Unfortunately, the shape of the abstract syntax tree strongly influences the type inference process. As a consequence, the reported site of error for an ill-typed expression involving these combinators can be counter-intuitive and misleading. Ideally, the type inferencer should follow the conceptual perception rather than the view according to the abstract syntax tree.

Phasing by example

Let `f` be a function, and let `p`, `q`, and `r` be parsers. Consider the following expression.

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

Figure 7 illustrates the abstract syntax tree of this expression and its conceptual view. How can we let the type inferencer behave

according to the conceptual structure? A reasonable choice would be to treat it in a similar way as a non-curried function application, that is, first infer a type for the function and all its arguments, and then unify the function and argument types. We can identify four steps if we apply the same idea to the parser combinators. Note that the four step process applies to the program as a whole.

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.

One way to view the four step approach is that all parser related unifications are delayed. Consequently, if a parser related constraint is inconsistent with another constraint, then the former will be blamed.

Example 5. The following example presents a type incorrect attempt to parse a string followed by an exclamation mark.

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

The type error message of Hugs is not too helpful here.

```
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 yield:

```
(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
```

Observe the two major improvements. First of all, it focuses on the problematic function, instead of mentioning an application. Secondly, the types do not involve the complex expanded *Parser* type synonym, nor do they contain the symbol type of the parsers, which in this example is irrelevant information.

Assigning phase numbers

Delaying the satisfaction of constraints can be achieved by annotations with a phase number. This phase number influences the order in which the constraints are solved. The constraints in phase number i are solved before the constraint solver continues with the constraints of phase $i+1$. Consequently, phasing has a global effect on the type inference process.

Adding the keyword `phase`, followed by a phase number, will assign the constraints after this directive to this phase. By default, constraints are assigned to phase 5, leaving space to introduce new phases. Of course, the constraints of a type rule can be assigned to different phases.

Example 6. We introduce phases numbered from 6 to 8 for the steps *2*, *3*, and *4* respectively. We assign those phase numbers to the constraints in the specialized type rule for <$>. Note that step *1* takes place in phase 5, which is the default. No constraint generated by the following type rule will be solved in phase 5.

```
  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
```

One may wonder what happens when the sets `constraints x` and `constraints y` are included among the listed constraints. Because phasing is a global operation, the constraints in these sets continue to keep their own assigned phase number.

Sometimes the opposite approach is desired: to verify the correctness of the parser related unifications before continuing with the rest of the program. This technique is similar to pushing down the type of a type declaration as an expected type, a useful technique applied by, for instance, the GHC compiler.

Example 7. Let us take another look at the ill-typed function definition `test` in Example 1.

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

If the constraints introduced by the type rule for <$> are assigned to an early phase, e.g. 3, then, effectively, the right operand is imposed to have a *Parser* type. Since `"hello, world!"` is of type *String*, it is at the location of this literal that we report that a different type was expected by the enclosing context. By modifying the `.type` file along these lines, we may obtain the following error message.

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

3.3 Sibling functions

This section and the next one deal with anticipating common mistakes. Although some mistakes are made over and over again, the quality of the produced error reports can be unsatisfactory. In some cases it is possible to detect that a known pitfall resulted in a type error message. If so, a more specific message than the standard one should be presented, preferably with hints to solve the problem.

One typical mistake that leads to a type error is confusing two functions that are somehow related. For example, novice functional programmers have a hard time remembering the difference between inserting an element in front of a list (`:`), and concatenating two lists (`++`). Even experienced programmers may mix up the types of `curry` and `uncurry` now and then. Similar mistakes are likely to occur in the context of a combinator language. We will refer to such a pair of related functions as *siblings*. The idea is to suggest

replacing a function with a sibling function if this resolves the type error. The types of two siblings should be distinct, since we cannot distinguish the differences based on their semantics.

Example 8. Consider the parser combinators from Section 2, and, in particular, the special variants that ignore the result of the right operand parser. These combinators are clearly siblings of their basic combinator. A closer look to the program in Figure 1 tells us that the most likely source of error is the confusion over the combinators $<\!\!*\!\!>$ and $<\!\!*$. The observation that replacing one $<\!\!*$ combinator by $<\!\!*\!\!>$ results in a type correct program paves the way for a more appropriate and considerably simpler error message.

A function can have multiple sibling functions, but the sibling relation is not necessarily transitive. Furthermore, a sibling function should only be suggested if replacement completely resolves a type error. Moreover, the suggested function should not only match with its arguments, but it should also fit the context to prevent misleading hints. Implementing this in a traditional type inference algorithm can be quite a challenge. A practical concern is the runtime behaviour of the type inferencer in the presence of sibling functions. Ideally, the presence of sibling functions should not affect the type inference process for type correct programs; only for type incorrect programs is some extra computation performed, and only for operators that contribute directly to the type error. In Section 4.3 we discuss type graphs, a flexible data structure that is powerful enough to handle sibling functions.

A set of sibling functions can be declared in the file containing the type inference directives by giving a comma separated list of functions.

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

The type error that is constructed for the program in Figure 1 can be found in Figure 5. A more conservative approach would be to show a standard type error message, and add the *probable fix* as a hint.

3.4 Permuted arguments

Another class of problems is the improper use of a function, such as supplying the arguments in a wrong order, or mistakenly pairing arguments. McAdam discusses the implementation of a system that tackles these problems by unifying types modulo linear isomorphism [12]. Although we are confident that these techniques can be incorporated into our own system, we limit ourselves to a small subset, that is, permuting the arguments of a function.

Example 9. The function `option` expects a parser and a result that should be returned if no input is consumed. But in which order should the arguments be given? Consider the following program and its type error message.

```
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] -> [(String,[Char])
],[a])]
```

The error message does not guide the programmer in fixing his program. In fact, it assumes the user knows that the non-matching type is equal to `Parser a (Parser Char String)`.

Instead of having to specify for each function, whether you want the type inferencer to attempt to resolve an inconsistency by permuting the arguments to the function, our type inferencer does this by default. A conservative type error message for the program of Example 9 would now be:

```
(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
```

If, for a given type error, both the method of sibling functions and permuted arguments can be applied, then preference is given to the former.

In a class room setting, we have seen that the permuted arguments facility gives useful hints in many cases. However, we are aware that sometimes it may result in misleading information from the compiler. During a functional programming course we have collected enough information to determine how often this occurs in a practical setting. The data remains to be analyzed.

4 Technical details

In this section we briefly discuss the way in which our type rules are applied, and the machinery that we use to test for sibling functions and permuted function arguments.

4.1 Applying specialized type rules

In Section 3.1 we introduced notation to define specialized type rules for combinator libraries. Typically, a set of type rules is given to cover the existing combinators and possibly some more complex combinations of these. Since we do not want to forbid overlapping patterns in the conclusions of the type rules, we have to be more specific about the way we apply type rules to a program.

The abstract syntax tree of the program is used to find matching patterns. We look for fitting patterns starting at the root of this tree, and continue in a top-down fashion. In case more than one pattern can be applied at a given node, we select the type rule which occurs first in the `.type` file. Consequently, nested patterns should be given before patterns that are more general. This first-come first-served way of dealing with type rules also takes place when two combinator libraries are imported: the type rules of the combinator library which is imported first have precedence.

Example 10. Matching the patterns on the abstract syntax tree of the program involves one subtle issue. Consider the following patterns, in the given order.

name	*pattern*	*meta-variables*
R1	$f <\!\!\$\!\!> p <\!\!*\!\!> q$	f, p, and q
R2	$p <\!\!*\!\!> q <\!\!*\!\!> r$	p, q, and r
R3	$f <\!\!\$\!\!> p$	f and p

What happens if we apply these rules to the code fragment

```
test = fun <$> a <*> b <*> c.
```

At first sight, rule *R1* seems to be a possible candidate to match the right-hand side of `test`. However, following the chosen priority and associativity of the operators, the second rule matches the top node of the abstract syntax tree, which is the rightmost <∗>. Meta-variable *p* in *R2* matches with the expression `fun <$> a`. Since this subexpression matches rule *R3*, we apply another type rule.

4.2 Correctness of specialized type rules

In Heeren, Hage and Swierstra [7] we proved the correctness of the underlying type system. Before using a specialized type rule, we verify that it does not change the type system. We think that such a feature is essential, because a mistake is easily made. We do this by ensuring that for a given expression in the deduction rule, the constraints generated by the type system and the constraints generated from the specialized type rule are equivalent.

A specialized type rule allows us to influence the order in which constraints are solved. The order of solving constraints is irrelevant except that the constraints for let definitions should be solved before continuing with the body [7]. If we make sure that all our specialized type rules respect this fact, then the correctness of the new type rules together with the underlying type system is guaranteed.

Validation of a specialized type rule

A type rule is validated in two steps. In the first step, a type rule is checked for various restrictions. The (expression) variables that occur in the conclusion can be divided into two classes: the variables that are present in a premise, which are the meta-variables, and the variables that solely occur in the conclusion. Each meta-variable should occur exactly once in the conclusion, and it should not be part of more than one premise. Every non-meta-variable in the conclusion should correspond to a top-level function inside the scope of the type rule's module. The type signature of such a function should be known a priori.

If none of the restrictions above is violated, then we continue with the second step. Here, we test if the type rule is a specialized version of the standard type rules present in the type system. Two types are computed: one for which the type rule is completely ignored, and one type according to the type rule. The type rule will be added to the type system if and only if the types are equivalent (up to the renaming of type variables). Before we discuss how to check the soundness of a type rule, we present an example of an invalid type rule.

Example 11. Take a look at the following type rule.

```
  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
```

Because the programmer forgot to specify that `x` should work on the result type of `y`, the type rule above is not restrictive enough. Thus, it is rejected by the type system with the following error message.

```
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)
```

Note that the 3-tuple in this error message lists the type of `x`, `y` and `x <$> y`, which reflects the order in which they occur in the type rule.

To determine this, we first ignore the type rule, and use the default type inference algorithm. Let Γ be the current type environment to which we add all meta-variables from the type rule, each paired with a fresh type variable. For the expression e in the conclusion, the type inference algorithm will return a type τ and a substitution $\mathcal{S}$ such that $\mathcal{S}\Gamma \vdash_{\text{HM}} e : \mathcal{S}\tau$. Let ϕ be $(\mathcal{S}\beta_1, \ldots, \mathcal{S}\beta_n, \mathcal{S}\tau)$, where β_i is the fresh type variable of the ith meta-variable. The order of the meta-variables is irrelevant, but it should be consistent.

Example 11 (continued). Let $e = x$ <$> y, and construct a type environment

$$\Gamma = \begin{cases} \text{<\$>} : \forall s, a, b \;.\; (a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \\ x : \beta_1 \\ y : \beta_2 \end{cases}$$

Given Γ and e, the default type inference algorithm returns a most general unifier $\mathcal{S}$ and a type τ.

$$\mathcal{S} = [\beta_1 := \beta_3 \to \beta_4, \beta_2 := \mathit{Parser}\ \beta_5\ \beta_3]$$
$$\tau = \mathit{Parser}\ \beta_5\ \beta_4$$

As a result we find that

$$\phi = (\beta_3 \to \beta_4, \mathit{Parser}\ \beta_5\ \beta_3, \mathit{Parser}\ \beta_5\ \beta_4)$$

Next, we use the type rule and ignore the standard type system. Let Γ' be the type environment containing all top-level definitions of which the type is known at this point, and let $\mathcal{C}$ be the set of type constraints given in the type rule. Compute a most general substitution $\mathcal{S}$ that satisfies $\mathcal{C}$, and let ψ be $(\mathcal{S}\tau_1, \ldots, \mathcal{S}\tau_n, \mathcal{S}\tau)$, where τ_i is the type of the i^{th} meta-variable, and τ is the type of the conclusion. The type rule is consistent with the default type system if and only if ϕ and ψ are equivalent up to variable renaming.

Example 11 (continued). Let Γ' be a type environment that contains <$> with its type scheme, and let $\mathcal{C}$ be the two constraints specified in the type rule. Next, we compute

$$\mathcal{S} = \quad [t1 := a1 \to b, t2 := \mathit{Parser}\ s\ a2]$$

and, consequently,

$$\psi = (a1 \to b, \mathit{Parser}\ s\ a2, \mathit{Parser}\ s\ b).$$

Because $\phi \neq_\alpha \psi$ we reject the examined type rule.

In theory, there is no need to reject a type rule that is more specific than the default type rules, since it will not make the type system unsound. Our implementation of specialized type rules, however, cannot cope with more specific type rules for the reason that the constraints implied by the type rule replace the constraint collection from the standard type system. Permitting a type rule that is too restrictive can result in the rejection of type correct programs.

Example 12. Although the following specialized type rule is sound, it is rejected because it is too restrictive in the symbol type. It is important to realize that applying the type rule to expressions of the form `x <$> y` is not influenced by the type mentioned in the type rule, here `Parser Char b`. Note that this specialized type rule has an empty set of type constraints.

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

Phasing and let expressions

Phasing the type inference process gives a great degree of freedom to order the constraints, because it is a global operation. However, this freedom is restricted by the correct treatment of let-polymorphism. The type scheme of an identifier defined in a let should be computed before it is instantiated for occurrences of that identifier in the body. This imposes a restriction on the order since the constraints corresponding to the let-definition should be solved earlier than the constraints that originate from the body.

Example 13. Consider the following function.

```
maybeTwice =
   let p = map toUpper <$> token "hello"
   in option ((++) <$> p <*> p) []
```

A type scheme should be inferred for `p`, before we start inferring a type for `maybeTwice`. Therefore, all type constraints that are collected for the right-hand side of `p` are considered before the constraints of the body of the let, thereby ignoring assigned phase numbers. Of course, phasing still has an effect inside the local definition as well as inside the body. Note that if we provide an explicit type declaration for `p`, then there is no reason to separate the constraint sets of `p` and the body of the let.

4.3 Type graphs

To conclude our technical discussion, we want to say something about the use of type graphs which allowed us to easily implement our scriptable type inferencer. The implementation is based on type graphs which we describe in a technical report [7]. Essentially, a type graph is an advanced data structure to represent substitutions, which also keeps track of the reasons for a unification. Type graphs allow us to solve the collected type constraints in a more global way, so that we suffer less from the notorious left-to-right bias present in most type inference algorithms. Type graphs have the following advantages.

- All the justifications for a particular shape of an inferred type remain available. Similarly, all the unification steps that contribute to a type error are still accessible in a type graph. For example, it is easy to compute a minimal set of program points contributing to a given type error, or to trace the origin of a type constructor.
- Since we solve a set of constraints at once, we can discover global properties.
- Because a type graph can be in an inconsistent state, the construction of a type graph is separated from the removal of inconsistencies. This makes it easier to plug in heuristics, such as the approach of Walz [15].
- Type graphs allow us to locally add and remove sets of constraints. If the use of `(++)` in an expression contributes to an error, then we can remove the corresponding constraints and add those of one of its siblings, e.g. `(:)`, instead to see whether this solves the problem. Permuted arguments can also be implemented by local modifications to the type graph.

However, there is a tradeoff between quality and performance. Obviously, the extra overhead caused by the type graph increases the compile time of the system. For normal sized programs, the price to be paid remains within reasonable proportions. For instance, type graphs have been implemented successfully in a concrete educational setting, and are part of the Helium Compiler [8]. Although benchmarks with up to 1400 lines of code have been typed within reasonable time, it is unknown to us to what extent the overhead becomes problematic for larger modules.

5 Related work

The poor type error messages produced by most modern compilers are still a serious obstacle to appreciate higher-order, functional programming languages. Several approaches have been proposed to improve the quality of type error messages. Because recent extensions to the type system appear to have a negative effect on the clarity of the error reports, it is becoming more important to understand the difficulties of type inference in full detail.

It is well understood that the reported site of error is greatly influenced by the order in which types are unified. Several modifications of the unification order in the standard algorithm $\mathcal{W}$ [4] have been presented, among which the top-down algorithm $\mathcal{M}$ [10]. Generalizations of these algorithms exist [7, 11]. To remove the left-to-right bias, Yang [9] presents algorithm $\mathcal{U}_{\mathrm{AE}}$, which unifies types in the assumption environment. Nevertheless, an algorithm with a fixed order of unification can never be satisfactory for all inputs.

A number of papers attempt to explain the type inference process by tracing type unifications [2, 5]. Unfortunately, lots of type variables are involved in the explanations, which tend to be lengthy and verbose. Chitil [3] discusses a tool to navigate through an explanation graph to inspect types.

Although it is generally impossible to blame a single location for a type inconsistency, we believe that the most appropriate site should be selected and reported, rather than enumerating all contributing sites. For instance, this decision could be based on the number of justifications for a particular type constant, as was suggested by Walz [15]. Separating the collection of constraints and solving a set of constraints allows us to perform a global analysis of a program. Formulating the type inference process as a constraint problem is not new [1, 13]. Recently, Haack and Wells [6] have shown how to compute a minimal set of program locations that contributed to a type inconsistency from a set of constraints. The type error slices in this paper are a nice and compact notation to present the error paths in our type graph.

The focus of most type error messages is to explain the conflicting types in a program, but little attention has been devoted to include suggestions on how to repair a type incorrect program. As a result, programmers have to extract the corrections that are required to resolve a type error without any help. The only paper we are aware of that addresses repairing type incorrect programs considers unification of types modulo linear isomorphism [12].

6 Conclusion

We have shown how the four techniques for externally modifying the behaviour of a constraint-based type inferencer can improve the quality of type error messages. 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 systems. 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.

We have shown how our techniques can be applied to a parser combinator library. We think it is clear that our techniques can be applied equally well to other libraries, including the standard `Haskell` Prelude. In fact, we are currently in the process of constructing type inference directives for the `Helium` Prelude.

Our techniques have all been implemented in the `Helium` compiler, but they can be applied in other compilers as well. We want to point out that implementing our ideas was relatively easy, mainly as a consequence of using a constraint-based approach to type inference.

It might seem that the directives we have discussed work only for type graphs. This is not the case. Specialized type rules and phasing both work equally well when using a greedy constraint solver (comparable in behaviour to, e.g., $\mathcal{W}$ and $\mathcal{M}$). (This is in fact possible in `Helium`.) The sibling functions and permuted function arguments were straightforwardly implemented using type graphs. We think that implementing them for greedy solvers is a bigger challenge, which probably involves some mechanism for backtracking.

Finally, our specialized type rules do not allow a change of scope. We believe that an extension to allow this raises more complications than it is worth.

There are still a number of possibilities to be examined in future work. The first of these is to build backtracking into our type inference mechanism so that greedy constraint solvers can also benefit from sibling functions and the permutation of function arguments. As a second possibility, we expect to extend the facility for permuted function arguments to handle isomorphic types in general. In the near future we shall extend `Helium` with type classes. As a result, we expect to be able to extend our framework along similar lines. Two other extensions which we are taking into consideration are adding flexibility in specifying the priority of specialized type rules, and extending the facilities for phasing; at this point, phasing is a purely global operation, which might be too coarse for some applications.

By their nature, the specialized type rules follow the structure of the abstract syntax tree. A different approach, is to include a facility to discover whether changing the structure of an ill-typed expression, e.g., by moving some of the parentheses of the expression, results in a well-typed expression. This information can then be included as a hint.

Acknowledgements

We thank Dave Clarke and Arjan van IJzendoorn for their suggestions and comments.

7 References

[1] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[2] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. In *ACM Letters on Programming Languages*, volume 2, pages 17–30, December 1993.

[3] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 193–204, September 2001.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages*, pages 207–212, 1982.

[5] D. Duggan and F. Bent. Explaining type inference. In *Science of Computer Programming 27*, pages 37–83, 1996.

[6] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the 12th European Symposium on Programming*, pages 284–301, April 2003.

[7] B. Heeren, J. Hage, and S. D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands, July 2002.

[8] A. van IJzendoorn, D. Leijen, and B. Heeren. The Helium compiler. `http://www.cs.uu.nl/helium`.

[9] J. Yang. Explaining type errors by finding the sources of type conflicts. In G. Michaelson, P. Trindler, and H.-W. Loidl, editors, *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.

[10] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[11] O. Lee and K. Yi. A generalized let-polymorphic type inference algorithm. Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, March 2000.

[12] B. J. McAdam. How to repair type errors automatically. In *3rd Scottish Workshop on Functional Programming*, pages 121–135. Stirling, U.K., August 2001.

[13] M. Sulzmann, M. Odersky, and M. Wehr. Type inference with constrained types. Research Report YALEU/DCS/RR-1129, Yale University, Department of Computer Science, April 1997.

[14] S. D. Swierstra. Combinator parsers: From toys to tools. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.

[15] J. A. Walz and G. F. Johnson. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 44–57, St. Petersburg, FL, January 1986.