Improving Type Error Diagnosis

Peter J. Stuckey
Department of Computer Science and Software Engineering, University of Melbourne, 3010, Australia
pjs@cs.mu.oz.au

Martin Sulzmann
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Jeremy Wazny
Department of Computer Science and Software Engineering, University of Melbourne, 3010, Australia
jeremyrw@cs.mu.oz.au

Abstract

We present a number of methods for providing improved type error reports in the Haskell and Chameleon programming languages. We build upon our previous work [19] where we first introduced the idea of discovering type errors by translating the typing problem into a constraint problem and looking for minimal unsatisfiable subsets of constraints. This allowed us to find precise sets of program locations which are in conflict with each other. Here we extend this approach by extracting additional useful information from these minimal unsatisfiable sets. This allows us to report errors as conflicts amongst a number of possible, candidate types. The advantage of our approach is that it offers implementors the flexibility to employ heuristics to select where, amongst all the locations involved, an error should be reported. In addition, we present methods for providing improved subsumption and ambiguity error reporting.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism,Constraints*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms

Languages, Theory

Keywords

type inference, type debugging, Hindley/Milner, overloading, type classes, constraints

1 Introduction

In previous work [19], we introduced a comprehensive type debugging framework which has been implemented as part of the Chameleon system [20]. The basic idea is to map expressions to constraints. Upon encountering a type error, i.e. unsatisfiable constraint, we extract the minimal amount of information necessary for explaining those errors; we look for minimal unsatisfiable constraints. Values attached to individual constraints refer back to program locations that contributed to those constraints. Highlighting those locations allows us to identify expressions which contributed to the type error. Our approach naturally deals with type classes; we are able to highlight type class rules (such as instances and functional dependencies) which are part of the type error.

The difficulty with type error diagnosis, in general, is that there might be several minimal unsatisfiable constraints (which correspond to several type error explanations), yet it may only be computationally feasible to find one minimal unsatisfiable constraint. If there is a common subset of all errors, we can efficiently compute the intersection of all minimal unsatisfiable constraints.

Under our original error reporting scheme, we first highlight all locations corresponding to a minimal unsatisfiable constraint and then, separately also highlight locations which are part of the intersection of all minimal unsatisfiable constraint (if non-empty). Obviously, we would like to provide for a more meaningful full type error explanation e.g. in terms of a textual message.

Throughout the paper, we compare the results of our error reporting approach to the output of GHC [5] version 6. We feel that GHC has the best type error reporting of all the current Haskell implementations. Note that we often reformat error messages, maintaining their basic structure, to fit in a single column.

EXAMPLE 1. Consider the following ill-typed program where the highlighted fragments of program text refer to one possible minimal unsatisfiable subset of constraints.

```
insert x [] = x
insert x (y:ys) | x > y = y : insert x ys
                | otherwise = x : y :  ys
```
□

The mistake here is that in the first clause, `x` is returned (supposedly as the result of inserting `x` into an empty list.) In the second (correct) clause the result is a list whose elements have the same type as `x`. The problem then is that the return type of `insert` cannot both be t_x and $[t_x]$. However, this might not be immediately obvious by looking at the highlighted program text, in particular since no types are actually reported.

Attempting to compile this program using the Haskell compiler GHC, we get the following error message.

```
insert.hs:2:
  Occurs check: cannot construct the infinite type: a = [a]
      Expected type: [a]
```

```
      Inferred type: a
  In the application `insert x ys'
  In the second argument of `(:)', namely `insert x ys'
```

Though slightly shortsighted (it refers only to x while ignoring the y) this message does provide crucial information about the sort of type error that is present. It suggests immediately that the error arises because somehow the type of a list is being equated with the type of its elements. This is valuable since it represents a more specific view of the problem.

Chameleon's error reporting scheme outlined in this paper was used to generate the following error message.

```
insert.hs:1: ERROR: Type error - some locations are also
                              part of other errors
Problem : Pattern variable `x' (bound at line 1, col. 8)
          used with multiple types
Types   : a
          [a]
Conflict: insert x [] = x
          insert x (y:ys) | x > y = y : insert x ys
                          | otherwise = x : y : ys
Common  : insert x [] = x
          insert x (y:ys) | x > y = y : insert x ys
                          | otherwise = x : y : ys
```

The diagnosis of the problem is clearer here – indicating a conflict amongst the multiple possible types of x. The two possible types are shown, and the program locations which contribute to those types are outlined/highlighted. Note that the outline/highlighting style used on the type corresponds to the style of the locations from which it arises. Locations which are common to both types are outlined and highlighted. All of the locations involved in the error are indicated, focusing the programmer's attention on locations which are part of the problem.

Furthermore, this error message also indicates that the locations in the first clause appear in multiple minimal unsatisfiable subsets of type constraints. The 'Common' part of the error message above draws attention to those locations.

In Haskell, programmers enjoy the benefits of type inference. However, it is considered good programming practice to provide explicit type annotations as part of the program text. If the declared type and inferred type do not match, we'd like to identify which locations in the definition contradict the declared type.

EXAMPLE 2. The following program contains a mismatch between the declared and inferred types.

```
prefixLen :: Integral a => a -> [Char] -> [Char]
prefixLen n xs = (\len -> if len < n
                          then show len ++ ":" ++ xs
                          else xs) (length xs)
```
□

The problem here is the use of length, which has type $[a] \rightarrow Int$, which causes len and consequently the function parameter n's type to be Int rather than Integral a => a as declared.

GHC complains:

```
prefixLen.hs:2:
    Cannot unify the type-signature variable `a'
    with the type `Int'
        Expected type: a
        Inferred type: Int
    In the application `length xs'
    In the first argument of
       `(\ len -> if len < n
                  then (show len) ++ (":" ++ ...)
                  else xs)', namely
       `(length xs)'
```

In this message, the connection between the expression length xs and the type-signature variable is not that clear.

The Chameleon system reports the following:

```
prefixLen.hs:2: ERROR: Inferred type does not subsume
                       declared type
Declared: forall a. Integral a => a -> [Char] -> [Char]
Inferred: Int -> [Char] -> [Char]
Problem : The variable `a' makes the declared type
          too polymorphic
          prefixLen :: Integral a => a -> [Char] -> [Char]
          prefixLen n xs = (\len-> if len < n
                                   then show len ++ ":" ++ xs
                                   else xs) (length xs)
```

The error message above clearly states the problem with the declared type, and highlights a set of program locations which lead to the error. The key difference between this messages and GHC's is that the connection between the variable a and the expression responsible for instantiating it (length) is much more obvious.

Functions with an inferred or declared type which is ambiguous represent another form of type error. Given such an error, we wish to explain why it arose and how the type could be fixed.

EXAMPLE 3. In the following program, f's type is ambiguous. We make use of a three-parameter type class Mul where the functional dependency a b -> c states that the first two parameters uniquely determine the third one.

```
class Mul a b c | a b -> c where mul :: a -> b -> c
instance Mul Int Int Int where mul = (*)

f xs = let len = length xs
       in len `mul` 10

zero = f []
```
□

GHC reports:

```
mul.hs:7:
    No instance for (Mul Int b c)
      arising from use of `f' at mul.hs:6
    Possible cause: the monomorphism restriction applied to
                    the following:
      zero :: c (bound at mul.hs:6)
    Probable fix: give these definition(s) an explicit
                  type signature
    In the definition of `zero': zero = f []
```

In this case, GHC's 'probable fix' will not fix the problem at all, since the source of the error is f's body, not zero.

The Chameleon system states the following:

```
mul.hs:4: ERROR: Inferred type scheme is ambiguous:
Type scheme: forall a,b,c. (Num a, Mul Int a b) => [c] -> b
Suggestion : Ambiguity can be resolved at these locations
             a: f xs = let len = length xs
                       in len `mul` 10
```

This message indicates that it is f's inferred type that is ambiguous. The ambiguous variable above is a, which appears within the expressions mul and 10. Adding an annotation to ground a's type at either expression will resolve the ambiguity. The obvious change would be something like: 10::Int.

We present methods for reasoning about sets of type constraints which represent program errors. We build upon our previous work [19] by not only looking at minimal subsets of erroneous constraints, but also considering ways to extract more information which could be useful to the programmer when reasoning about the error. The particular contributions of this paper are:

- By generating all typing constraints in advance we can tailor the error message to address sections of the program which are more likely to be in error. Furthermore, error reports are not restricted to conflicts between only two types, and errors involving type classes and functional dependencies in particular can be carefully handled.
- A method for identifying any program locations which contradict a declared type. The programmer can resolve the problem by fixing any of the nominated locations.
- We improve reporting of missing instances errors by finding the source locations which lead to a demand for an instance that does not exist.
- We present a method for identifying program locations which can be fixed by the programmer to resolve ambiguity errors. Adding an appropriate annotation to any such site resolves the problem.

We continue in Section 2 by introducing types and constraints. Then, in Section 3 we present a method for finding types to report to the user, given a minimal unsatisfiable set of constraints. In Section 4, we look at ways to improve reporting of subsumption, missing-instance and ambiguity errors. We discuss related work in Section 5, and conclude in Section 6.

2 Types and Constraints

We first review our type debugging framework introduced in [19]. For simplicity of presentation we limit ourselves to a subset of Haskell. E.g. we consider only function, class and instance declarations, disallow nested definitions, and ignore the monomorphism restriction. We include multi-parameter type classes with functional dependencies [11], since our support for these most distinguishes our approach from others. Our implementation handles all of what we omit here, apart from the monomorphism restriction.

For brevity, we leave out the function bodies associated with instance declarations. We note that programs are fully *labelled*, i.e. we label program locations with unique numbers. We follow the convention that programs are labelled by a pre-order of the abstract syntax tree (AST). That is *lower* location numbers represents *higher* program locations (higher in the AST.) Note that if-then-else expressions are given two locations: one for the conditional expression, and one for the two branches.

$$
\begin{array}{lll}
\text{Declarations} & d & ::= \; \texttt{class}\ (Ctx \Rightarrow TC)_l \mid fd,...,fd \\
 & & \quad\;\; \texttt{where}\ f :: (Ctx \Rightarrow t)_l \\
 & & \quad \mid \; \texttt{instance}\ (Ctx \Rightarrow TC)_l \\
 & & \quad \mid \; \{df;\ldots;df\}_l \\
 & & \quad \mid \; an \\
\text{Expressions} & e & ::= \; f_l \mid x_l \mid c_l \mid (\lambda x_l.e)_l \\
 & & \quad \mid \; (e\ e)_l \mid (\texttt{if}_l\ e\ \texttt{then}\ e\ \texttt{else}\ e)_l \\
\text{Patterns} & p & ::= \; x_l \mid c_l \mid (c\ p \ldots\ p)_l \\
\text{Definitions} & df & ::= \; f_l\ p\ \ldots\ p = e \\
\text{Annotations} & an & ::= \; f_l :: \ Ctx \Rightarrow t \\
\text{Types} & t & ::= \; a \mid t\ t \mid T \\
\text{Type Schemes} & \sigma & ::= \; t \mid \forall \bar{a}.Ctx \Rightarrow t \\
\text{Type Class} & TC & ::= \; U\ t \ldots t \\
\text{Context} & Ctx & ::= \; TC,\ldots,TC \\
\text{FD} & fd & ::= \; a \cdots a \;\texttt{->}_l\; a
\end{array}
$$

In the above, f represents a function name, x represents a lambda bound variable, c represents a data constructor, T represents a type constructor, and a represents a type variable. A function declaration d is a sequence of definitions for the same function f, with an optional type annotation for f. Note that we assume annotations to be closed. That is, $f_l :: \ Ctx \Rightarrow t$ is interpreted as $f_l :: \ \forall \bar{a}.Ctx \Rightarrow t$ where $\bar{a} = fv(Ctx,t)$. For convenience, we often leave the forall quantifier implicit. We write $\bar{o}$ to denote a sequence of objects $o_1,...,o_n$.

The subscripts l indicate labelled program locations. As stated, we make use of natural numbers as labels; each program site is labelled with a distinct number. We write $source(l)$ to indicate the textual location of the fragment labelled l in the source program. We often write a type variable referring to location l as t_l.

In our type system framework, we make use of two kinds of constraints—equations and user-defined constraints. An equation is of the form $t_1 = t_2$, where t_1 and t_2 are types that share the same structure as types in the language. User-defined constraints are written $U\ \bar{t}$ or $f(t)$. We use these two forms to distinguish between constraints representing type class overloading and those arising from function definitions. The meaning of user-defined constraints is described by a set of Constraint Handling Rules (CHRs), as we will see below.

Conjunctions of constraints are written using a comma separator. We often treat conjunctions as sets of constraints. We assume a special (always satisfiable) constraint *True* representing the empty conjunction of constraints, and a special never-satisfiable constraint *False*. If C is a conjunction we let C_e be the equations in C and C_u be the user-defined constraints in C. We assume the usual definitions of substitution, most general unifier (mgu), etc. [14]. The meaning of type class constraints, as specified by class and instance declarations, and the relations among program text and types is described by Constraint Handling Rules (CHRs) [4]. More details follow below.

We will make use of *justified* constraints which have a list of labels representing program locations attached. The *justification* of a constraint refers to the program locations from which the constraint arose. We shall denote justified constraints using a subscript list of locations, and typically write singleton justified constraints $C_{[i]}$ as simply C_i. We write $J_1 +\!\!+ J_2$ to represent the result of appending justification J_2 to the end of J_1. For our purposes, we can safely remove any repeated location which appears to the right of another occurrence of that location. e.g. $[1,2,1,3,2]$ becomes $[1,2,3]$. We will often do this automatically to keep justifications a manageable length.

2.1 CHRs with Justifications

CHRs [4] manipulate a global set of primitive constraints, using rewrite rules of two forms

$$
\begin{array}{lrcl}
\textit{simplification} & c_1,\ldots,c_n & \Longleftrightarrow & d_1,\ldots,d_m \\
\textit{propagation} & c_1,\ldots,c_n & \Longrightarrow & d_1,\ldots,d_m
\end{array}
$$

where $c_1,\ldots,c_n$ are user-defined constraints, and $d_1,\ldots,d_m$ are constraints.

The logical interpretation of the rules is as follows. Let $\bar{x}$ be the variables occurring in $\{c_1,\ldots,c_n\}$, and $\bar{y}$ be the other variables occurring in the rule. The logical reading is

$$
\begin{array}{ll}
\textit{simplification} & \forall \bar{x}((c_1 \wedge \cdots \wedge c_n) \leftrightarrow \exists \bar{y}\ (d_1 \wedge \cdots \wedge d_m)) \\
\textit{propagation} & \forall \bar{x}((c_1 \wedge \cdots \wedge c_n) \supset \exists \bar{y}\ (d_1 \wedge \cdots \wedge d_m))
\end{array}
$$

We write $\supset$ for logical implication, to avoid any confusion with the function type constructor. In our use, constraints occurring on the right hand side of rules have attached justifications. We extend the usual derivation steps of CHRs to maintain justifications.

A *simplification derivation step* applying a renamed rule instance $r \equiv c_1,\ldots,c_n \Longleftrightarrow d_1,\ldots,d_m$ to a set of constraints C is defined as follows. Let $E \subseteq C_e$ be such that the most general unifier of E is θ. Let $D = \{c'_1,\ldots,c'_n\} \subseteq C_u$, and suppose there exists substitution σ on variables in r such that $\{\theta(c'_1),\ldots,\theta(c'_n)\} = \{\sigma(c_1),\ldots,\sigma(c_n)\}$, that is a subset of C_u *matches* the left hand side of r under the substitution given by E. The *justification* J of the matching is the concatenation of the justifications of $D \cup E$ (in the order D, in order of matching, followed by E in an arbitrary order). For our purposes we require E to be *minimal* in the sense that no subset of E can allow the rule match to occur. See [19] for an explanation of the algorithm we use.

Then we create a set of constraints $C' = C - \{c'_1,\ldots,c'_n\} \cup \{\theta(c'_1) = c_1,\ldots,\theta(c'_n) = c_n, (d_1)_{+J},\ldots,(d_n)_{+J}\}$. Note that the equation $\theta(c'_i) = c_i$ is short for $\theta(s_1) = t_1,\ldots,\theta(s_m) = t_m$ where $c'_i \equiv p(s_1,\ldots,s_m)_{J'}$ and $c_i \equiv p(t_1,\ldots,t_m)$. The annotation $+J$ indicates that we add the justification set J to the *beginning* of the original justifications of each d_i. The other constraints (the equality constraints arising from the match) are given empty justifications.

A *propagation derivation step* applying a renamed rule instance $r \equiv c_1,\ldots,c_n \Longrightarrow d_1,\ldots,d_m$ is defined similarly except the resulting set of constraints is $C' = C \cup \{\theta(c'_1) = c_1,\ldots,\theta(c'_n) = c_n, (d_1)_{+J},\ldots,(d_n)_{+J}\}$.

A derivation step from global set of constraints C to C' using an instance of rule r is denoted $C \longrightarrow_r C'$. A *derivation*, denoted $C \longrightarrow^*_P C'$ is a sequence of derivation steps using rules in P where no derivation step is applicable to C'. The operational semantics of CHRs exhaustively apply rules to the global set of constraints, being careful not to apply propagation rules twice on the same constraints (to avoid infinite propagation). For more details on avoiding re-propagation see e.g [1]. Note that we will only ever generate and use confluent sets of rules – making the specific order of rule application irrelevant.

EXAMPLE 4. Consider the following justified rules.

$$
\begin{array}{lrcl}
(r1) & G\ a\ b & \Longrightarrow & True_1 \\
(r2) & G\ a\ b,\ G\ a\ c & \Longrightarrow & (b = c)_2 \\
(r3) & g(t) & \Longleftrightarrow & (G\ a\ b)_3,\ (t = a \rightarrow b)_3 \\
(r4) & G\ Bool\ Bool & \Longleftrightarrow & True_4 \\
(r5) & G\ a\ Bool & \Longrightarrow & (a = Bool)_{[4,2]} \\
(r6) & f(t_5) & \Longleftrightarrow & (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t_6)_6, \\
 & & & not_a(t_7)_7, \\
 & & & (t_9 = t_{10} \rightarrow t_8)_8, g(t_9)_9, \\
 & & & (t_{11} = t_{12} \rightarrow t_{10})_{10}, g(t_{11})_{11}, \\
 & & & (t_{12} = Char)_{12} \\
(r7) & not_a(t) & \Longleftrightarrow & t = Bool \rightarrow Bool
\end{array}
$$

□

We begin a CHR derivation with the goal $f(t)$ and proceed as follows. We underline the constraints $D \cup E$ causing each rule firing.

$$
\begin{array}{ll}
 & \underline{f(t)} \\
\rightarrow_{(r6)} & t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t_6)_6, \\
 & \underline{not_a(t_7)_7}, (t_9 = t_{10} \rightarrow t_8)_8, \\
 & \underline{g(t_9)_9}, (t_{11} = t_{12} \rightarrow t_{10})_{10}, g(t_{11})_{11}, (t_{12} = Char)_{12} \\
\rightarrow_{(r7)} & t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t_6)_6, \\
 & t_7 = t''', (t''' = Bool \rightarrow Bool)_7, (t_9 = t_{10} \rightarrow t_8)_8, \\
 & \underline{g(t_9)_9}, (t_{11} = t_{12} \rightarrow t_{10})_{10}, g(t_{11})_{11}, (t_{12} = Char)_{12} \\
\rightarrow_{(r3)} & G\ a'\ b'_{[9,3]}, (t' = a' \rightarrow b')_{[9,3]}, t_9 = t', \\
 & t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t_6)_6, \\
 & t_7 = t''', (t''' = Bool \rightarrow Bool)_7, (t_9 = t_{10} \rightarrow t_8)_8, \\
 & (t_{11} = t_{12} \rightarrow t_{10})_{10}, \underline{g(t_{11})_{11}}, (t_{12} = Char)_{12} \\
\rightarrow_{(r3)} & G\ a''\ b''_{[11,3]}, (t'' = a'' \rightarrow b'')_{[11,3]}, t_{11} = t'', \\
 & \underline{G\ a'\ b'_{[9,3]}}, \underline{(t' = a' \rightarrow b')_{[9,3]}}, \underline{t_9 = t'}, \\
 & t = t_5, (t_5 = t_6)_5, \underline{(t_7 = t_8 \rightarrow t_6)_6}, \\
 & \underline{t_7 = t'''}, \underline{(t''' = Bool \rightarrow Bool)_7}, \underline{(t_9 = t_{10} \rightarrow t_8)_8}, \\
 & (t_{11} = t_{12} \rightarrow t_{10})_{10}, (t_{12} = Char)_{12} \\
\rightarrow_{(r5)} & (a''' = Bool)_{[9,3,6,7,8,2,4]}, \underline{a' = a'''}, \\
 & \underline{G\ a''\ b''_{[11,3]}}, \underline{(t'' = a'' \rightarrow b'')_{[11,3]}}, \underline{t_{11} = t''}, \\
 & \underline{G\ a'\ b'_{[9,3]}}, \underline{(t' = a' \rightarrow b')_{[9,3]}}, \underline{t_9 = t'}, \\
 & t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t_6)_6, \\
 & t_7 = t''', (t''' = Bool \rightarrow Bool)_7, \underline{(t_9 = t_{10} \rightarrow t_8)_8}, \\
 & \underline{(t_{11} = t_{12} \rightarrow t_{10})_{10}}, (t_{12} = Char)_{12} \\
\rightarrow_{(r5)} & (a'''' = Bool)_{[11,3,9,6,7,8,2,4,10]}, a'' = a'''', \\
 & (a' = Bool)_{[9,3,6,7,8,2,4]}, a' = a''' \\
 & G\ a''\ b''_{[11,3]}, (t'' = a'' \rightarrow b'')_{[11,3]}, t_{11} = t'', \\
 & G\ a'\ b'_{9,3]}, (t' = a' \rightarrow b')_{[9,3]}, t_9 = t', \\
 & t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t_6)_6, \\
 & t_7 = t''', (t''' = Bool \rightarrow Bool)_7, (t_9 = t_{10} \rightarrow t_8)_8, \\
 & (t_{11} = t_{12} \rightarrow t_{10})_{10}, (t_{12} = Char)_{12}
\end{array}
$$

2.2 Translation to Constraints

Each type annotation f_l :: $(U_1\ \overline{t_1},...,U_n\ \overline{t_n}) \Rightarrow t$ creates an *annotation* CHR $f_a(\tau) \Longleftrightarrow (\tau = t)_l, (U_1\ \overline{t_1})_l, ..., (U_n\ \overline{t_n})_l$. We use τ to denote a fresh type variable. We convert a function declaration for f into a set of justified constraints C as shown below. Given this set of constraints we create a *definition* CHR for f of the form $f(t_{def(f)}) \Longleftrightarrow C$. Note that in case of a type annotated function we add a call to the annotation CHR to the rhs of the definition CHR.

Program Text	Constraints
f_l :: $Ctx \Rightarrow t$	$f_a(t_l)_l$
$\{df_{i_1};\ldots;df_{i_n}\}_j$	$(t_j = t_{i_1})_j,\ldots,(t_j = t_{i_n})_j$
$f_l\ p_{i_1}\ldots p_{i_n} = e_j$	$(t_l = t_{i_1} \rightarrow \cdots \rightarrow t_{i_n} \rightarrow t_j)_l$
x_i	$(t_i = t_x)_{[bnd(x),i]}$
f_i	$f_a(t_i)_i$ $\quad f$ (annotated)
f_i	$(t_i = t_{def(f)})_i$ $\quad f$ (recursive)
f_i	$f(t_i)_i$ $\quad f$ (non-recursive)
$(e_i\ e_j)_k$	$(t_i = t_j \rightarrow \tau)_k, (t_k = \tau)_k$
$(\backslash x_i \rightarrow e_j)_k$	$(t_k = t_i \rightarrow t_j)_k$
c_i	$(t_i = type(c))_i$
(if$_{l_1}$ e_i then e_j else $e_k)_{l_2}$	$(t_i = Bool)_{l_1}, (t_{l_2} = t_j)_{l_2}, (t_{l_2} = t_k)_{l_2}$

Here $bnd(x)$ is the location l where x is bound (either a lambda expression or pattern location), $def(f)$ is the location l of the declaration of function f, and $type(c)$ is (a new copy of) the type of the constructor c. We treat patterns just like ordinary expressions in terms of constraint generation.

EXAMPLE 5. We give an example of constraint generation for the following definition.

```
f0 x1 = (\y3 -> (if5 y6 then [x9]7 else x8)4)2
f10 :: Eq a => a -> Bool -> a
```

We generate the annotation CHR

$$f_a(t) \Longleftrightarrow (t = a \rightarrow Bool \rightarrow a)_{10}, (Eq\ a)_{10}$$

We then proceed in a top-down fashion, and generate C: $(t_0 = t_1 \rightarrow t_2)_0$, $(t_1 = t_x)_1$, $(t_2 = t_3 \rightarrow t_4)_2$, $(t_3 = t_y)_3$, $(t_4 = t_7)_4$, $(t_4 = t_8)_4$, $(t_6 = Bool)_5$, $(t_6 = t_y)_{[3,6]}$, $(t_7 = [t_9])_7$, $(t_8 = t_x)_{[1,8]}$, $(t_9 = t_x)_{[1,9]}$. We then create the definition CHR for f as $f(t_0) \Longleftrightarrow C, f_a(t_0)$. □

Class and instance declarations are also converted into CHRs. The class declaration

$$\texttt{class}\ (Ctx \Rightarrow (U\ t_1\ \ldots\ t_n)_{l_1} \mid fd_1, \ldots, fd_n \texttt{where} f :: (C \Rightarrow t)_{l_2}$$

creates the *super-class* and *method* CHRs.

$$U\ t_1\ \ldots\ t_n \Longrightarrow Ctx_{l_1}$$
$$f(t_{l_2}) \Longleftrightarrow t = t_{l_2}, C_{l_2}, (U\ t_1\ \ldots\ t_n)_{l_2}$$

And for each $fd_i \equiv a_{i_1} \ldots a_{i_k} \text{->}_l\ a_{i_0}$ the *FD* CHR

$$U\ t_1\ \ldots\ t_n, U\ s_1\ \ldots\ s_n \Longrightarrow (t_{i_0} = s_{i_0})_l$$

where $s_j = t_j$ if $t_j = a_{i_m}$ for some $1 \leq m \leq k$ and otherwise s_j is a new variable.

Each instance declaration $\texttt{instance}\ Ctx \Rightarrow (U\ t_1\ \ldots\ t_n)_{l_0}$ creates the *instance* CHR $U\ t_1\ \ldots\ t_n \Longleftrightarrow Ctx_{l_0}$ and for each functional dependency for the class definition $fd_i \equiv a_{i_1} \ldots a_{i_k} \text{->}_l\ a_{i_0}$ an *instance improvement* CHR

$$U\ s_1\ \ldots\ s_n \Longrightarrow (t_{i_0} = s_{i_0})_{[l_0,l]}$$

where (again) $s_j = t_j$ if $t_j = a_{i_m}$ for some $1 \leq m \leq k$ and otherwise s_j is new variable. We often refer to the FD and instance improvement CHRs as *improvement* rules.

We refer the interested reader to [18, 3] for a detailed discussion on the connection between CHRs and type classes.

EXAMPLE 6. As an example of the translation of instance declarations consider the following program, annotated with location numbers.

```
class (G a b)1 | b ->2 a where (g :: a -> b)3
instance4 G Bool Bool
f5 = (not7 (g9 (g11 'a'12)10)8)6
```
□

The CHRs generated from this program are those already given in Example 4. For pre-defined functions such as `not` we assume that the location attached to the annotation constraints is left empty.

2.3 Type Inference

Once CHRs have been generated from a source program, inference can be performed. To infer the type of function f, given CHRs P, we perform $f(t) \rightarrow_P^* C$. If C_e is satisfiable, then f has type $\forall fv(C).\phi C_u \Rightarrow \phi t$ where $\phi = mgu(C_e)$. If however C_e is unsatisfiable, then f contains a type error, and no definite type can be inferred.

Our goal is to report type inference errors only in terms of the program locations which contributed to them. In the event of a type error we look for a minimal subset of C which is still unsatisfiable [19]. The set is minimal in the sense that removal of any constraint would render it satisfiable. Such a set represents the smallest set of program locations which caused that type error. By examining the justifications of the constraints in this minimal unsatisfiable subset, we can identify the program locations which led to the type error. In the previous iteration of Chameleon, our response to a type error was simply to highlight those locations.

Note that there may be more than one minimal unsatisfiable subset of a constraint. Here we focus on finding and using only one, since calculating all of them is generally computationally infeasible. Finding constraints which are common to all minimal unsatisfiable subsets present is fairly inexpensive, however, and forms an important part of the error diagnosis (if there is a common subset present.) See [19] for a description of the algorithm for finding a single minimal unsatisfiable subset as well as finding constraints common to all such subsets.

In Haskell, there are still further sources for type errors. Types must be "unambiguous" and in the presence of type annotations we need to check that the inferred type "subsumes" the annotated type. Also, any type class instances required by the program must be present. In the following section, we devote our attention to type inference errors. Error reporting in case of subsumption and ambiguity errors is discussed in Section 4.

3 Generating error messages

As mentioned earlier, the Chameleon system is geared towards finding sources of type conflict. Previous versions simply highlighted all of the conflicting locations, and did not report any of the conflicting types within a program, since there is no definite way to: a) decide which location is the actual source of the error ; or b) determine which are the correct, intended types. Unfortunately, this often made it difficult for the user to determine what the problem actually was, since error messages would only report the source of the conflict, without providing any reason for it.

Our incentive for using text-based error messages is to reveal some of the problematic types in the program. Type conflicts arise because according to the rules of our type system, some location(s) within the program would have to have two (or more) distinct, ununifiable types. We can report such an error by displaying those types, explaining why they cannot coexist and by showing how they individually came to be.

One problem with traditional $\mathcal{W}$-style type inference algorithms is that they process programs in a fixed order. They proceed through the abstract syntax tree, unifying types until something fails. At that point, they simply stop and report that the last location they visited was problematic. Furthermore, the error reported will always consist of exactly two incompatible types, an 'expected' type and an 'inferred' type, which could not be unified.

In our constraint-based formulation of inference we are able to take a broader view, and consider all locations (not just one) involved in a type error, when deciding which conflicting types to report. In addition, we are not restricted to reporting only two problematic types. Indeed, we will see cases where it may be advantageous to report three or more types.

Briefly, our algorithm for generating text error messages with type information, from a minimal unsatisfiable set of justified constraints is as follows:

1. Select a location to report the type conflict about
2. Find the types that conflict at that location
    - Assign each a colour (in the paper we use a combination of highlighting and outline styles) and determine which locations contribute to it
3. Report an error diagnosis in terms of the conflicting types at

the chosen location.
Highlight in colour conflicting locations based on which of the conflicting types they contribute to.

3.1 Selecting an erroneous location

As you can see by the multiple highlighted locations in Example 1, there can be a large number of sites within the program which are part of a type error. The first problem is to determine which of these locations to select for more detailed reporting.

Any location that the debugger highlights is a potential source of error. Lacking further guidance, the debugger cannot determine whether one location is more likely to have a mistake than another. Therefore, any scheme for selecting a location is at best a heuristic.

When choosing a location to report, one strategy that we have found to be reasonably successful is to select the location from amongst those in the minimal unsatisfiable subset which appears 'highest' in the program's abstract syntax tree. The most compelling reason for this selection is that fresh variables and ground types remain in their respective scopes/sub-trees.

EXAMPLE 7. The following program is ill-typed. The two branches of the if-then-else have disjoint types.

```
f c = if c then \ g x -> g x
          else \ e y -> y e
```
□

GHC prints the following error message:

```
top.hs:2:
    Occurs check: cannot construct the infinite type:
        t = (t -> t1) -> t1
        Expected type: t
        Inferred type: (t -> t1) -> t1
    In the first argument of 'y', namely 'e'
    In a lambda abstraction:  e y -> y e
```

It is our conjecture that most programmers would prefer to consider the types of the two branches in isolation, and that combining them leads to confusion – particularly when the order of processing is arbitrary.

By selecting the highest program location involved in the error, we treat the types of the two branches separately, and report the following:

```
top.hs:1: ERROR: Type error - one error found
Problem : Branches of if-then-else have incompatible types
Types   : (a -> b) -> a -> c (then)
          d -> (d -> e) -> f (else)
Conflict: f x = if x then \ g x -> g x
                     else \ e y -> y e
```

3.2 Finding the types to report

In this section we present an algorithm which, given a minimal unsatisfiable set of constraints and a program location to report, will generate an appropriate text message to present to the programmer.

This algorithm is designed so that it can adopt a different error reporting strategy for each type of location that could lead to an error. So, for example, if the location we pick is an application node in the AST, we can complain specifically about the conflict between the function being applied and its argument. Or if the nominated location is an if-then-else we could report the conflict between the 'then' and 'else' branches; and so on.

Furthermore, instead of just reporting a set of conflicting types, the algorithm can relate each of those types to location(s) in the program. As such, the algorithm is necessarily closely tied to both the language grammar, as well as constraint generation. It can obviously be adapted for other languages though.

The algorithm, which finds the conflicting types to report, for a location l is presented in Figure 1. We assume that the minimal unsatisfiable set of constraints which constitutes the type error is C. Variables τ represent fresh types. Again, we treat patterns as expressions.

The procedure for finding the reportable types depends on the sort of program location l is. As such, each type of location at which an error could arise must be accounted for. Note that in the case of if-then-else expressions, there are two possible causes of error: either the conditional is not *Bool* or the two branches disagree which correspond to different locations.

As mentioned earlier, this algorithm is intimately tied to the constraint generation scheme. The intuition behind it is as follows. Given a minimal unsatisfiable set of constraints C, we cannot extract any concrete type information to report. Since C is minimal, however, we need only remove a single constraint to make it satisfiable. For each kind of location in the program, the algorithm finds any constraints which can tell us something useful about the type error from the point of view of that location. (This information is shown in parentheses following each reported type.) At least one constraint is then removed from C, the substitution ϕ (of the remaining constraints) is built, and the interesting types can be extracted as necessary. So in essence, when we remove a constraint and then build the unifier, we are doing inference for all locations involved except one, and then report the reason why that last step cannot proceed.

Note that all constraint manipulation performed reflects the generation scheme of Section 2.2, and is guaranteed to succeed by construction.

3.3 Finding the source of each type

Although we can now present a great deal of information about a type error, the connection between the types reported and the locations highlighted may not be obvious. As shown earlier, we will further refine our approach to highlighting, by tying it to the conflicting types that were found.

For each type that is reported, we will find all those constraints amongst the minimal unsatisfiable subset which collectively cause it to occur. We will then assign each reported type its own colour, and highlight the locations of the constraints which contributed to it in the same colour.

Assume that we find type t for the expression at location l, whose type is represented by variable t_l, in the minimal unsatisfiable constraint C. We then find a minimal $D \subseteq C$, such that $\models D \supset t_l = t$, where $\models$ is just the semantic entailment relation in first-order logic. The algorithm for finding such a D is related to that for finding minimal unsatisfiable subsets [19]. Having picked a distinct colour for location l, we can then highlight all locations which justify the constraints in D by that colour.

Locations contributing to multiple conflicting types should be assigned a different, distinct colour. We have chosen to highlight them in a mix of the colours whose types arose from them.

EXAMPLE 8. In the following program, there is a conflict amongst the branches of the if-then-else expression.

$$\begin{array}{rl}
location & \{df_{i_1};\ldots;df_{i_n}\}_l \\
for & T_l = \{t_l = \tau \mid t_l = \tau \in C\} \\
let & R_l = \{\tau \mid t_l = \tau \in T_l\} \\
 & \phi = mgu(C - T_l) \\
report & \phi R_l \ (individual\ clauses)
\end{array}$$

$$\begin{array}{rl}
location & f_l\ p_{i_1} \ldots\ p_{i_n} = e_j \\
for & (t_l = t_{i_1} \rightarrow \ldots \rightarrow t_{i_n} \rightarrow t_j) \in C \\
let & \phi = mgu(C - \{(t_l = t_{i_1} \rightarrow \ldots \rightarrow t_{i_n} \rightarrow t_j)\}) \\
report & \phi(t_{i_1} \rightarrow \ldots \rightarrow t_{i_n} \rightarrow t_j)\ (def.skeleton), \\
 & \phi t_l\ (use)
\end{array}$$

$$\begin{array}{rl}
location & x_l \\
for & T_l = \{t_{bnd(x)} = \tau \mid t_{bnd(x)} = \tau \in C\} \\
let & R_l = \{\tau \mid t_{bnd(x)} = \tau \in T_l\} \\
 & \phi = mgu(C - T_l) \\
report & \phi R_l\ (uses\ of\ x)
\end{array}$$

$$\begin{array}{rl}
location & c_l \\
for & (t_l = \tau) \in C \\
let & \phi = mgu(C - \{(t_l = \tau)\}) \\
report & \phi\tau\ (constructor\ type), \phi t_l\ (used\ at\ l)
\end{array}$$

$$\begin{array}{rl}
location & f_l\ (recursive) \\
for & (t_l = \tau) \in C \\
let & \phi = mgu(C - \{(t_l = \tau)\}) \\
report & \phi\tau\ (def.skeleton),\ \phi t_i\ (recursive\ use)
\end{array}$$

$$\begin{array}{rl}
location & f_l\ (non\text{-}recursive\ or\ annotated) \\
for & (t_l = \tau) \in C \\
let & \phi = mgu(C - \{(t_l = \tau)\}) \\
report & \phi\tau\ (definition\ of\ f),\ \phi t_l\ (use\ at\ l)
\end{array}$$

$$\begin{array}{rl}
location & (\backslash x_i \rightarrow e_j)_l \\
for & (t_l = t_i \rightarrow t_j) \in C \\
 & (\tau = t_l) \in C \\
let & \phi = mgu(C - (\tau = t_l)) \\
report & \phi t_l\ (abstraction), \phi\tau\ (elsewhere)
\end{array}$$

$$\begin{array}{rl}
location & (e_i\ e_j)_l \\
for & (t_i = t_j \rightarrow \tau) \in C \\
let & \phi = mgu(C - \{(t_i = t_j \rightarrow \tau)\}) \\
report & \phi t_i\ (function), \phi t_j\ (argument)
\end{array}$$

$$\begin{array}{rl}
location & (\texttt{if}\ e_i\ \texttt{then}\ e_j\ \texttt{else}\ e_k)_l \\
for & (t_l = t_j) \in C \\
 & (t_l = t_k) \in C,\ t_j \neq t_k \\
let & \phi = mgu(C - \{(t_l = t_j), (t_l = t_k)\}) \\
report & \phi t_j\ (then), \phi t_k\ (else)
\end{array}$$

$$\begin{array}{rl}
location & \texttt{if}_l\ e_i\ \texttt{then}\ e_j\ \texttt{else}\ e_k \\
for & (t_i = Bool) \in C \\
let & \phi = mgu(C - \{(t_i = Bool)\}) \\
report & \phi t_i\ (test), Bool\ (if)
\end{array}$$

Figure 1. Finding the conflicting types at a location l, given a minimal unsatisfiable set of constraints C

```
not x = if x then string2bool "False" else fst ("True", 1)
```

We report:

```
not.hs:1: ERROR: Type error - one error found
Problem : Branches of if-then-else have incompatible types
Types   : Bool    (then)
          [Char]  (else)
Conflict: not x = if x then string2bool "False"
                       else fst ("True", 1)
```

□

All of the highlighted locations are part of the error. Those shaded cause the `Bool` type in the `then` branch, whereas those outlined cause the `[Char]` in the `else` branch. The if-then-else itself is both shaded and outlined, as it is a common part of the error.

EXAMPLE 9. Consider the following ill-typed program:

```
f 'a' b    z = error "'a'"
f c   True z = error "True"
f x   y    z = if z then x else y
f x   y    z = error "last"
```

□

GHC reports:

```
mdef.hs:3:
    Couldn't match 'Char' against 'Bool'
        Expected type: Char
        Inferred type: Bool
    In the definition of 'f': f x y z = if z then x else y
```

What's confusing here is that GHC combines type information from a number of clauses in a non-obvious way. In particular, in a more complex program, it may not be clear at all where the *Char* and *Bool* types it complains about come from. Indeed, it isn't even obvious where the conflict in the above program is. Is it complaining about the two branches of the if-then-else (if so, which is *Char* and which *Bool*?), or about `z` which might be a *Char*, but as the conditional must be a *Bool*?

The Chameleon system reports:

```
multi.hs:1: ERROR: Type error - one error found
Problem : Definition clauses not unifiable
Types   : Char -> a -> b -> c
          d -> Bool -> e -> f
          g -> g -> h -> i
Conflict: f 'a' b z = error "'A'"
          f c True z = error "True"
          f x y z = if z then x else y
```

Note we do not mention the last definition equation which is irrelevant to the error.

In general, we might need to report any number of types. Consider the following (where we write p_1@p_2 to give pattern p_2 the alias p_1):

$$\texttt{f}\ \texttt{p}_1\texttt{@('a')}\ \texttt{p}_2\ \ldots\ \texttt{p}_{n-1}\ \texttt{p}_n\texttt{@(True)} =_{i=1}^{n-1} \begin{array}{l}\texttt{if undefined}\\ \texttt{then p}_i\\ \texttt{else p}_{i+1}\end{array}$$

Here we'd have $n-1$ clauses and would need to display a type for each. In practice, however, we'd expect that cases where a large number of types need to be printed are uncommon. Regardless, we feel that displaying all individual, clause-specific types is always preferable to reporting some arbitrary combination of those types.

Since the types that are reported are built entirely from the minimal unsatisfiable set of constraints, they may not be as ground as may be expected. In Example 9, it is clear that the third argument to `f` must be *Bool*, but this is not reported in any of the types. Our system, however, only reports the minimal amount of type information to indicate the error. In general, of course, it may not be possible to deduce such concrete type information, since there may be multiple minimal unsatisfiable subsets.

3.4 Alternative error location selection

In the previous sections we have only considered reporting errors for locations from the minimal unsatisfiable subset which are highest in the abstract syntax tree. Although in general this seems to be a reasonable choice, occasionally it may be beneficial to try sites which are deeper.

The problem with picking any location is that it is only a guess. Without evidence supporting that guess it is not worthwhile making, since it's more likely to be a bad guess than a good one – a good guess being one which pinpoints the place where the programmer really made a mistake.

Our best heuristic for narrowing down the set of potentially good guesses is to find the set of constraints/locations common to all minimal unsatisfiable subsets. When such a set exists (is non-empty) we will choose the highest location within it, rather than within the entire minimal unsatisfiable subset. (Note that if there's only one type conflict these two sets will be the same.)

We re-examine Example 1. The error message given there was in terms of the highest location common to all minimal unsatisfiable subsets, which is the pattern variable `x` in the first definition of `insert`. Contrast that with the following message which is generated if we select the highest location in the entire minimal unsatisfiable subset.

```
insert.hs:3: ERROR: Type error - some locations are also
                                 part of other errors
Problem : Definition clauses incompatible
Types   : a -> b -> a
          c -> d -> [c]
Conflict: insert x [] = x
          insert x (y:ys) | x > y = y :
                                    insert x ys
                          | otherwise = x : y : ys
Common  : insert x [] = x
          insert x (y:ys) | x > y = y : insert x ys
                          | otherwise = x : y : ys
```

Though also helpful, the error that is reported is slightly removed from the actual source of the problem.

Obviously, error reports which are closest to the actual source of the mistake tend to be most helpful. In general, however, we usually cannot find these optimal locations.

3.5 Reporting errors involving functional dependencies

So far we have only considered reporting errors which involve the standard Hindley/Milner type system extended with Haskell-style type classes. Some extensions to the type system, such as functional dependencies can introduce type errors and their involvement must be reported in a useful way.

The problem is that our existing error reporting scheme is insufficient in the presence of functional dependencies. The scheme outlined earlier (in the beginning of Section 3) depends on constraints justified by some locations to be of a certain form. For example, if location l is an application site that is part of the set of erroneous locations, then we expect a constraint $(t = t_l \to r)_l$ to be part of the minimal unsatisfiable subset. Unfortunately, this may not be the case once propagation rules are involved, and justifications can be freely copied between Herbrand constraints.

Let us consider type inference of the function f defined in Example 6. The constraints determined in the derivation shown in Section 2.1 are unsatisfiable. The only minimal unsatisfiable subset C is: $(a'''' = Bool)_{[11,3,9,6,7,8,2,4,10]}$, $a'' = a''''$, $(t'' = a'' \to b'')_{[11,3]}$, $t'' = t_{11}$, $(t_{11} = t_{12} \to t_{10})_{10}$, $(t_{12} = Char)_{12}$.

If we were to apply the approach outlined in Section 3.1 we would attempt to explain the error in terms of location 6. (We can ignore locations 2,3 and 4 which arise from class and instance declarations.)

Since 6 is an application site, we would require a constraint of form $(t_7 = t_8 \to t_6)_6$ in order to report the error, according to Figure 1. Unfortunately, no such constraint is present. The only reason 6 is part of the justification set is because it was copied when an earlier rule fired. Indeed, location 6 is a critical part of the error, but all that we know from the minimal unsatisfiable subset is that 6, amongst others, caused some rule to fire, which led to the error.

Clearly we need a different approach to handle errors arising from functional dependencies in a useful way. Our new method, specialised for errors involving improvement rules, is conceptually similar to the previous method, in that we extract types from a minimal unsatisfiable result, by first removing a constraint, and thus making the result satisfiable. In this case, however, the constraint we will remove is always one propagated by an improvement rule.

We report the error in terms of the types affected by the improvement rule. We first show the two types (and their justifications) that caused the error when they were unified by the improvement rule. We then show why the improvement rule fired (the locations responsible) and the effect of the rule, in causing the error.

We start from the minimal unsatisfiable subset C that arises when inferring the type for `f`. From this we need to determine the last constraint $s = t$ added by an improvement rule. Let $f(t) \longrightarrow^* C' \longrightarrow_R C''$ be the derivation for `f` where C'' is unsatisfiable, and the last derivation step involved an improvement rule R. We can always assume that the last step in a derivation leading to a functional-dependency error is a propagation rule step. [1] Now $s = t$ is the justified constraint occurring in $C'' - C'$. Let $\phi = mgu(C - \{s = t\})$. We report the types ϕs and ϕt (highlighting their minimal justifications) and then report the rule R and the justification for it firing, including the original location of the user constraints involved. The source position of the user constraints which caused the last rule to fire can be easily recovered from their justifications.

Note that our method for assigning justifications, outlined in Section 2.1, ensures that any user constraint derived from $(U t_1 \ldots t_n)_J$ will have a justification of form $J +\!\!+ K$, and so the leading location of J will still head all of the new justifications. If the user constraint $(U t_1 \ldots t_n)_{[l,\ldots]}$ is involved in the rule application, we can conclude that it originally arose from location $source(l)$ in the program text.

When reporting such an error we also explain how the improvement rule was derived, i.e. the combination of class and instance declarations it arose from.

We now revisit the program and constraints of Example 6.

GHC reports:

```
gs.hs:3:
  Couldn't match `Bool' against `Char'
```

[1] We already assumed in Section 2.1 that all rules are confluent. Consider that none of the simplification rules a propagation rule could cause to fire (only those arising from an instance declaration, see Section 2.2) can result in an error.

```
      Expected type: Bool
      Inferred type: Char
  When using functional dependencies to combine
    G Bool Bool, arising from use of `g' at gs.hs:3
    G Char Bool, arising from use of `g' at gs.hs:3
  When generalising the type(s) for `f'
```

We determine the last constraint added to the minimal unsatisfiable subset by an improvement rule (r5) is $s = t \equiv (a'''' = Bool)_{[11,3,9,6,7,8,2,4,10]}$. We calculate $\phi = mgu(C - \{s = t\})$ and then report $\phi a'''' = Char$ justified by $[3, 10, 11, 12]$, $\phi Bool = Bool$ justified by nothing, and finally the rule firing on constraint $G\ a''\ b''_{[11,3]}$ justified altogether by $[11, 3, 9, 6, 7, 8, 2, 4, 10]$

The Chameleon system reports the following:

```
gs.hs:3: ERROR: Functional dependency causes type error
Types   : Char
          Bool
Problem : class G a b | b -> a ...
          instance G Bool Bool ...
          Enforce: G a Bool ==> a = Bool
          On constraint:
             G Char Bool (from line 3, col. 14)
Conflict: f = not (g (g 'a'))
```

EXAMPLE 10. Consider the following program.

```
class Collects ce e | ce -> e where
    empty :: ce
    insert :: e -> ce -> ce

f c = insert 'a' (insert True c)
```
□

GHC declares:

```
collects.hs:5:
    Couldn't match `Bool' against `Char'
        Expected type: Bool
        Inferred type: Char
    When using functional dependencies to combine
      Collects ce Bool, arising from use of `insert'
                        at collects.hs:7
      Collects ce Char, arising from use of `insert'
                        at collects.hs:7
    When generalising the type(s) for `f'
```

We report:

```
collects.hs:5: ERROR: Functional dependency causes
                      type error
Types   : Char
          Bool
Problem : class Collects ce e | ce -> e ...
          Enforces: Collects ce e1,
                    Collects ce e2 ==> e1 = e2
          On constraints:
             Collects ce Char (from line 5, col. 7)
             Collects ce Bool (from line 5, col. 19)
Conflict: f c = insert 'a' (insert True c)
```

The advantage of our error report is that we are not limited to identifying just the locations of the `Collects` constraints above, we can straightforwardly point out all of the other complicit locations, and identify which of the conflicting types they contribute to.

It is clear that once functional dependencies are involved, we really need to be able to explain individual steps in a derivation to fully explain complex type errors. The problem becomes more complicated when other type extensions (in terms of CHRs) are allowed as in Chameleon. This is ongoing research in the Chameleon project.

4 Reporting subsumption, missing instance and ambiguity errors

In the previous section we looked at reporting type errors, which in our system, appear as unsatisfiable sets of constraints. Here we look at reporting errors related to types which can be inferred, but fail to meet some other condition; specifically, types which are too general, or which are ambiguous.

4.1 Subsumption errors

All user-declared types must be checked for correctness against an inferred, most-general type. For a declared type to be valid, it must be subsumed by the inferred type.

EXAMPLE 11. In the following program, the declared type is not subsumed by the inferred type:

```
f :: a -> (a, b)
f x = (x, x)
```
□

The (most general) inferred type of `f` is $a \to (a, a)$. In the declared type, the `b` in the tuple is too general, since it would allow `x`, which is monomorphic, to have two distinct types.

EXAMPLE 12. In this next program, the subsumption error involves a type class constraint:

```
notNull :: Eq a => [a] -> Bool
notNull xs = xs > []
```
□

Given the usual Haskell definition of `(>)` with type $Ord\ a \Rightarrow a \to a \to a$, the inferred type of `notNull` is $Ord\ a \Rightarrow [a] \to Bool$. The inferred *Ord* however is not subsumed by the *Eq* of the declared type. Indeed, `notNull` can only work for types $[a]$ where a is a member of *Ord*, a subclass of *Eq*.

In Chameleon, subsumption is defined as follows. Begin with inferred type $\forall \bar{a}.C \Rightarrow t$, and declared type $\forall \bar{a}'.C' \Rightarrow t'$. We assume that there are no name clashes. Let a be a fresh variable. Let $C'' \equiv C' \wedge C \wedge t = a \wedge t' = a$ and $C'' \rightarrow_P^* D$. Let $V = fv(C', t')$. Then $\forall \bar{a}.C \Rightarrow t$ subsumes $\forall \bar{a}'.C' \Rightarrow t'$ iff $\models (\bar{\exists}_V C') \leftrightarrow (\bar{\exists}_V D)$ where $\bar{\exists}_V F$ is the formula $\exists a_1 \cdots \exists a_n F$ and $\{a_1, \ldots a_n\} = fv(F) - V$. Correctness of this definition follows from [18].

Intuitively, this works by checking that the annotated type is equivalent to a combination of the annotated and inferred types. That is, the inferred type must be more general, since we check that adding it does not introduce anything that wasn't already present in the annotation.

We note that there is quite a bit of design space when it comes to defining a subsumption procedure. E.g. the subsumption procedure described in [18] is more "liberal" and works well for a runtime evidence translation scheme. Regardless, the subsumption error reporting ideas presented here can be straightforwardly adapted to work with any subsumption procedure expressible in terms of CHRs.

As can be seen from Examples 11 and 12, a subsumption error may arise in one of two ways: either the declared type component is more general than the inferred type, or the inferred user constraints cannot be matched against declared user constraints. First we consider the case where an inferred constraint is not represented by a corresponding declared constraint.

Given inferred and declared type schemes $\forall \bar{a}.C \Rightarrow t$, $\forall \bar{a}'.C' \Rightarrow t'$ and $C'' \longrightarrow^* D$, the subsumption test can fail if there is any $u \notin$

C'_u for some $u \in D_u$. Let U be the set of all such us. We restrict ourselves to reporting one unmatched constraint in U. We select the $u \in U$ with the fewest number of locations in its justification. If there are multiple candidates, we pick one arbitrarily. For constraint $u_{[i,\ldots]}$ we can then report that constraint u arising from location i in the program is unaccounted for by the annotation.

Returning to the `notNull` program of Example 12, we report the following.

```
notNull.hs:2: ERROR: Inferred type does not subsume
                     declared type
Declared: forall a. Eq a => [a] -> Bool
Inferred: forall a. Ord a => [a] -> Bool
Problem : Constraint Ord a, from following location,
          is unmatched.
          notNull :: Eq a => [a] -> Bool
          notNull xs = xs > []
```

It should be noted that GHC also seems to do well at reporting this sort of error; it appears to record the source location of each user constraint, so it can then report where any unaccounted for constraints come from.

GHC raises the following error:

```
notNull.hs:2:
    Could not deduce (Ord a) from the context (Eq a)
    arising from use of '>' at notNull.hs:2
    Probable fix:
        Add (Ord a) to the type signature(s) for 'notNull'
    In the definition of 'notNull':  notNull xs = xs > []
```

Other Haskell systems such as Hugs [10] and nhc98 [17], however, report the error without identifying the program locations responsible.

We now consider the case where the subsumption test fails because the declared type is too polymorphic. The aim is to find any type variables in the declaration, which correspond to more-instantiated types in the inferred type.

Given inferred and declared type schemes $\forall\bar{a}.\, C \Rightarrow t$ and $\forall\bar{a}'.\, C' \Rightarrow t'$, let $C'' \equiv C' \wedge C \wedge t = a \wedge t' = a$ so that $C'' \longrightarrow^* D$ and $\phi_1 = mgu(C'_e)$ and $\phi_2 = mgu(D_e)$. The subsumption test can also fail if $\phi_1(t') \neq \phi_2(t')$ which can only occur when $\phi_2(t')$ is more specialised than $\phi_1(t')$ (since D includes the constraints C'_e). We select an $a \in fv(t')$ such that $\phi_1(a) \neq \phi_2(a)$. Such an a is an annotation variable which is instantiated by the type of the program body. This must be explained, so we then find a minimal subset D_2 of C'' such that $\models D_2 \supset a = \phi_2(a)$. See [19] for a description of the algorithm for finding a minimal subset of implicants. It is essentially the same process as finding a minimal unsatisfiable subset. The justifications of constraints in D_2 identify the program locations which instantiate annotated type variable a, and therefore invalidate the declared type.

EXAMPLE 13. The following program contains a subsumption error.

```
notEq :: Eq a => a -> b -> Bool
notEq x y = if x == y then False else True
```
□

GHC reports:

```
notEq.hs:1:
  Inferred type is less polymorphic than expected
    Quantified type variable 'b' is unified with
    another quantified type variable 'a'
  When trying to generalise the type inferred for 'notEq'
    Signature type: forall a b. (Eq a) => a -> b -> Bool
    Type to generalise: a -> a -> Bool
  When checking the type signature for 'notEq'
  When generalising the type(s) for 'notEq'
```

The Chameleon system reports:

```
notEq.hs:2: ERROR: Inferred type does not subsume
                   declared type
Declared: forall a,b. Eq a => a -> b -> Bool
Inferred: forall b. Eq b => b -> b -> Bool
Problem : The variable 'a' makes the declared type
          too polymorphic
          notEq :: Eq a => a -> b -> Bool
          notEq x y = if x == y then False else True
```

The advantage of this error message is that it clearly indicates the program locations which violate the declared type; it is `(==)` applied to `x` and `y` which force them to have the same type.

4.2 Missing instance errors

In Haskell, the arguments of type classes appearing within a type declaration or inferred function type, must be of one of two forms, either a single variables (a) or a variable applied to a number of types ($a\ \tau_1 \ldots \tau_n$).

A non-conforming constraint is one whose arguments have not been reduced to one of those forms, indicating that some instance of that class is missing. We aim to report the error so that the programmer can see and address the source of the problem directly, by either changing the program so that it no longer requires such an instance, or by adding that instance if possible.

EXAMPLE 14. The following program violates the condition on type classes mentioned earlier. It is typical of the sort of mistake that beginners make. The complexity of the reported error is compounded by Haskell's overloading of numbers.

```
sumLists = sum .  map sum

sum [] = []
sum (x:xs) = x + sum xs
```
□

GHC does not report the error in `sum` until a monomorphic instance is required, at which point it discovers that no instance of $Num\ [a]$ exists. This means that unfortunately such errors may not be found through type checking alone – it may remain undiscovered until someone attempts to run the program. The function `sumLists` forces that here, and GHC reports:

```
sum.hs:1:
No instance for (Num [a])
  arising from use of 'sum' at sum.hs:1
Possible cause: the monomorphism restriction applied to
  the following:
  sumLists :: [[[a]]] -> [a] (bound at sum.hs:1)
Probable fix: give these definition(s) an explicit
  type signature
In the first argument of '(.)', namely 'sum'
In the definition of 'sumLists':
  sumLists = sum .  (map sum)
```

It would be useful to report not only the missing instance, but also the program locations which lead to the type for which we lack an instance. In other words, we need to find a minimal subset of constraints which imply the erroneous class constraint. Note that this is similar to the 'explain' command of the Chameleon debugger [19].

We restrict ourselves to single parameter type classes for this description, although the idea extends to any number of parameters. Assume we are interested in the type of function f, and $f(t) \longrightarrow^* C$, so that $f :: \forall fv(C).\phi\, C_u \Rightarrow \phi\, t$, where $\phi = mgu(C_e)$. Note that since C is unsimplified, all user constraints in C_u are of form $U\ a$, where a

is a type variable. We find a $U\ \tau \in C_u$ such that $\phi\tau$ is not a variable or a variable applied to any number of types. For some type constructor T, if $\phi\tau = T$, then let $\tau' = T$, otherwise if $\phi\tau = T\ \bar{\tau}$, let $\tau' = T\ \bar{a}$, where as are fresh variables; τ' is the type of the missing instance – we need to find where it comes from in the program text. Using the algorithm for finding minimal sets of implicants, mentioned earlier, we find a minimal subset $D_e \subseteq C_e$, such that $\models D_e \supset \tau = \tau'$. The justifications of constraints in D_e identify the problematic locations.

For the program in Example 14, we report the following:

```
sum.hs:3: ERROR: Missing instance
Instance:Num [a]: sum [] = []
                  sum (x:xs) = x + sum xs
```

This indicates that the demand for this instances arises from the interaction between `[]` on the first line and `(+)` on the second.

4.3 Ambiguity errors

In addition to checking subsumption, we require that all type schemes, whether inferred or declared, are unambiguous. Generally speaking, a type scheme is ambiguous if grounding all variables in the type component does not ground all variables in the constraint component.

More formally, we consider a function f such that $f \longrightarrow_P^* C$, and therefore $f :: \forall\bar{a}.C \Rightarrow t$, where $\bar{a} = fv(C)$, $\phi = mgu(C_e)$ and P is the set of CHRs from the program containing f. Let ρ be a renaming on $\bar{a}$, and $C \wedge \rho C \wedge t = \rho t \longrightarrow_P^* D$. If $\models D \supset a = \rho a$ for all $a \in \bar{a}$ then f is unambiguous. We say that f is ambiguous in any a which fails this test.

When reporting an ambiguity error, we assume that the inferred class constraints are correct, and that the error should be fixed by simply eliminating the ambiguous variables. This can be done through the introduction of additional type information to the program, which could be in the form of a type annotation. Therefore, the aim of our ambiguity reporting scheme is to uncover those locations in the program where additional type information can be inserted by the programmer to ground the problematic variables and resolve the ambiguity. Note that we ignore the possibility of user-declared ambiguous types, since they are trivial to report.

Returning to Example 3 in the introduction. GHC does not check for ambiguity until the use of `f` and hence really reports a different error. Hugs, on the same program, simply reports

```
ERROR "mul.hs":6 - Ambiguous type signature in inferred type
*** ambiguous type :  (Mul Int a b, Num a) => [c] -> b
*** assigned to :  f
```

The Chameleon system highlights where we could improve the type by grounding the ambiguous variable `a`.

In order to find the locations where ambiguous type variables appear, we proceed as follows. We start with a user constraint $f(t)$ representing the ambiguous type scheme
$\forall\bar{a}.C \Rightarrow t$, and using the method outlined above, find one ambiguous variable $a \in \bar{a}$. We then build ρ, the m.g.u. of C_e, without substituting for a. Given variables $t_l, ..., t_{l+n} \in fv(C)$, representing the types of locations $l, ..., l+n$, if $a \in fv(\rho t_i)$, then the type at location i contains the variable a. Therefore, we report it as a candidate for a type annotation.

In Example 3 the only locations which include the variable `a` in their type are `mul :: a -> b -> c` and `10 :: a`, which are highlighted.

5 Related Work

Much effort has been spent on improving type errors and type error debugging facilities in functional programming languages [23, 2, 19, 15, 21, 8, 7, 16, 6, 22]. All of the work we are aware of has focused on plain Hindley-Milner type systems, without type class overloading. As such, many ideas may not be adequate or appropriate for Haskell.

The most direct approaches involve modifying the order in which unification is performed in traditional inference algorithms [13, 23]. One shortcoming of such variants is that the order of traversal must be fixed at some point, and as such, 'better' errors may be overlooked in favour of worse ones which are found earlier. Furthermore these algorithms report an error at a single site, and therefore overlook the contribution of other program locations to the conflict. By generating all constraints in advance, our approach is much more flexible in that it effectively allows for any sub-tree to be searched for errors before any other.

From our use of GHC [5], we have found it to do quite well at reporting subsumption errors. This is in contrast to other Haskell implementations like Hugs [10] and nhc98 [17], which discover the error without identifying its source in the program text. Indeed, it is possible that GHC implements a simplified version of our scheme where only user constraints have associated source locations [12]. Unfortunately, we have not found any formal description of the methods used by GHC. What's clear, though, is that without keeping additional information around, like the unsimplified, justified constraints we use, it would be difficult to do much more than what GHC already does. Too much source information is lost during the unification process. The fact that GHC's ambiguity error reports are significantly less descriptive than ours seems to bear this out.

The Helium [8] language and programming environment have been designed to make programming less confusing for beginners. Helium employs a constraint-based approach, similar to ours, but as a system for inexperienced programmers, it lacks support for important language features like type classes. The Chameleon system, by contrast, has been designed with a view to capturing all of Haskell's type system and potentially more. Available in Helium is a scheme which gives library writers some control over the order in which unification occurs when a function's type is checked [7]. A specialised error message can be invoked when a specific step fails. Adding more specialised error messages and better heuristics to our approach is an ongoing topic of research. It is already clear, though, that our system can be straightforwardly extended to support Helium-style scripted messages.

In addition to improved static error messages, a number of interactive debugging systems have been developed. Indeed, the Chameleon system itself has support for interactive type debugging [19]. Key features include: type inference at arbitrary program locations and explanations for erroneous-looking types (this functionality is related to the techniques underlying the subsumption and ambiguity error reporting mentioned in this paper.) Some other interactive tools which assist the user in finding type errors manually include [9, 2]. These allow the user to uncover the type at any program location. Through examination of the types of subexpressions, the user gradually works towards the source of the error. Identifiers with suspicious looking types are followed to their definition and further examined. Neither of these systems automate the process of finding conflicting program locations; the process must be directed at every step by the user.

6 Conclusion

Reporting type errors in languages with polymorphic type inference is a difficult problem. Often the error found and reported by a traditional type inference algorithm does not correspond to the actual mistake the programmer has made. These algorithms are at a disadvantage since they traverse programs in a fixed order and stop as soon as any problem is discovered.

By adopting a constraint-based approach to type inference we allow for better guesses as to the true source of a type error. We have shown the advantages of taking a wider view of type errors than traditional algorithms allow for, and the system we have presented can easily be extended to support more advanced reasoning about type errors. In addition to type errors, we can employ constraint reasoning techniques to report and explain subsumption, missing instance and ambiguity errors. Devising better heuristics for discerning the most likely source of an error and providing even better diagnosis is an ongoing area of research.

Acknowledgements

We'd like to thank the reviewers for their helpful comments and encouragement.

7 References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, volume 1330 of *LNCS*, pages 252–266. Springer-Verlag, 1997.

[2] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proc. of ICFP'01*, pages 193–204. ACM Press, 2001.

[3] G.J. Duck, S. Peyton-Jones, P.J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies, 2004. to appear.

[4] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer-Verlag, 1995.

[5] Glasgow haskell compiler home page. `http://www.haskell.org/ghc/`.

[6] C. Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Proc. of ESOP'03*, volume 2618 of *LNCS*, pages 284–301. Springer-Verlag, 2003.

[7] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 3–13. ACM Press, 2003.

[8] Helium home page. `http://www.cs.uu.nl/~afie/helium/`.

[9] F. Huch, O. Chitil, and A. Simon. Typeview: a tool for understanding type errors. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 63–69. Aachner Informatik-Berichte,, 2000.

[10] Hugs home page. `http://www.haskell.org/hugs/`.

[11] M. P. Jones. Type classes with functional dependencies. In *Proc. of 9th European Sympsosium on Programming Languages and Systems*, pages 230–244, 2000.

[12] S. Peyton Jones and M. Shields. Practical type inference for arbitrary-rank types, 2004. Under consideration for publication in J. Functional Programming.

[13] O. Lee and K. Yi. A generalized let-polymorphic type inference algorithm. Technical Memorandum ROPAS-2000-5, National Creative Research Center, Korea Advanced Institute of Science and Technology, March 2000.

[14] K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[15] B.J. McAdam. Generalising techniques for type debugging. In *Trends in Functional Programming*, pages 49–57, March 2000.

[16] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *Proc. of ICFP'03*, pages 15–26. ACM Press, 2003.

[17] nhc98 home page. haskell.org/nhc98/.

[18] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proc. of ICFP'02*, pages 167–178. ACM Press, 2002.

[19] P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Proc. of Haskell Workshop'03*, pages 72–83. ACM Press, 2003.

[20] M. Sulzmann and J. Wazny. Chameleon. `http://www.comp.nus.edu.sg/~sulzmann/chameleon`.

[21] M. Wand. Finding the source of type errors. In *Proc. of POPL'86*, pages 38–43. ACM Press, 1986.

[22] J. Yang, G. Michaelson, and P. Trinder. Explaining polymorphic types, 2001.

[23] J. Yang, J. Wells, P. Trinder, and G. Michaelson. Improved type error reporting. In *Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 71–86, 2000.