Type Debugging in the Hindley/Milner System with Overloading

Peter J. Stuckey *, Martin Sulzmann †and Jeremy Wazny ‡

August 20, 2003

Abstract

We introduce a novel approach for debugging ill-typed programs in the Hindley/Milner system. We map the typing problem for a program to a system of constraints each attached to program code that generates the constraints. We use reasoning about constraint satisfiability and implication to find minimal justifications of type errors, and to explain unexpected types that arise. Through an interactive process akin to declarative debugging, a user can track down exactly where a type error occurs. We are able to capture various extensions of the Hindley/Milner system such as type annotations and Haskell-style type class overloading. The approach has been implemented as part of the Chameleon system.

1 Introduction

Strongly typed languages provide the user with the convenience to significantly reduce the number of errors in a program. Well-typed programs can be guaranteed not to "go wrong" [24], with respect to a large number of potential problems. However, programs are often not well-typed, and therefore must be modified before they can be accepted. Unfortunately, it can be difficult to determine why a program has been rejected.

Traditional inference algorithms depend on a particular traversal of the syntax tree. Therefore, inference frequently reports errors at locations which are far away from the actual source of the problem. The programmer is forced to tackle the problem of correcting her program unaided. This can be a daunting task for even experienced programmers; beginners are often left bewildered.

Despite recent efforts, see e.g. [3, 23], we believe there remains a lot of scope for improvement. For example, previous works exclude Haskell-style overloading [31] which can be naturally handled by our approach (see e.g. Examples 2 and 7), and is a serious cause for consternation for beginning Haskell programmers.

The novelty of our approach lies in mapping the entire typing problem, including type classes and extensions, to a set of constraints. Program locations are attached to individual constraints. By employing some simple constraint reasoning steps we are able to narrow down the source of the type error. We demonstrate our approach via a simple example.

Example 1 Consider the following annotated program where we use numbers to refer to individual program locations.

*Department of Computer Science and Software Engineering, University of Melbourne, 3100, Australia, pjs@cs.mu.oz.au

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

‡Department of Computer Science and Software Engineering, University of Melbourne, 3100, Australia, jeremyrw@cs.mu.oz.au

```
p4 = (f2 'a'1)3
f7 True5 = True6
```

Each expression is translated into a set of constraints, each of which has an attached *justification* in this case a program location number, and a type variable representing its type. For example, expression `(f`$_2$ `'a'`$_1$`)`$_3$ is translated to constraint $(t_1 = Char)_1$, $f(t_2)_2$, $(t_2 = t_1 \rightarrow t_3)_3$ with type t_3. Note that we introduce for each function symbol `f` a predicate symbol f. So, for example $f(t_2)_2$ refers to an instance of function `f` at location 2 where t_2 refers to the particular instance type.

Each function definition is translated to a Constraint Handling Rule (CHR) [6]. More specifically, we make use of CHR *simplification* rules. For example, in case of the above program we find

$$\begin{array}{lcl} p(t_4) & \Longleftrightarrow & (t_1 = Char)_1, f(t_2)_2, (t_2 = t_1 \rightarrow t_3)_3, (t_4 = t_3)_4 \\ f(t_7) & \Longleftrightarrow & (t_5 = Bool)_5, (t_6 = Bool)_6, (t_7 = t_5 \rightarrow t_6)_7 \end{array}$$

The left-hand side of the first rule consists of the predicate $p(t_4)$ where t_4 refers to the type of function `p`. The right-hand side consists of the constraints generated out of the program text representing the type of `p`. The $\Longleftrightarrow$ symbol can be read as logical equivalence. Operationally the rules are read as defining replacements, you may replace the left hand side by the right hand side. We keep applying rules until no further rules are applicable.

For example, we can infer the type of expression `p`$_8$ by applying the above CHRs to the initial constraints store $p(t)_8$ where 8 stands for a hypothetical program location. Here is the final result.

$$\begin{array}{ll} & p(t)_8 \\ \longrightarrow & t = t_4, (t_1 = Char)_{\{1,8\}}, f(t_2)_{\{2,8\}}(t_2 = t_1 \rightarrow t_3)_{\{3,8\}}, \\ & (t_4 = t_3)_{\{4,8\}} \\ \longrightarrow & t = t_4, (t_1 = Char)_{\{1,8\}}, t_2 = t_7, (t_5 = Bool)_{\{5,2,8\}}, \\ & (t_2 = t_1 \rightarrow t_3)_{\{3,8\}}, (t_6 = Bool)_{\{6,2,8\}}, \\ & (t_7 = t_5 \rightarrow t_6)_{\{7,2,8\}}, (t_4 = t_3)_{\{4,8\}} \end{array}$$

In the first step, the constraint $p(t)_8$ matches the left hand side of the first CHR. We replace $p(t)_8$ by the right hand side. In addition, we add the matching equation $t = t_4$. Note how the justification from $p(t)_8$ is added to each justification set. In the final step, the constraint $f(t_2)_{\{2,8\}}$ matches the left hand side of the second CHR.

Note that constraints are unsatisfiable in the final store. Indeed, `p` is not well-typed. By collecting justifications attached to unsatisfiable constraints we are able to narrow down the possible source of the type error. A minimal unsatisfiable subset of the resulting constraints is $(t_1 = Char)_{\{1,8\}}$, $t_2 = t_7$, $(t_5 = Bool)_{\{5,2,8\}}$, $(t_7 = t_5 \rightarrow t_6)_{\{7,2,8\}}$, $(t_2 = t_1 \rightarrow t_3)_{\{3,8\}}$. Hence the system underlines the program location $\{1, 2, 3, 5, 7\}$ (ignoring 8 since we did not provide it in the program)

```
p = f 'a'
f True = True
```

indicating that the program must be changed in at least one of these locations to be corrected. □

Haack and Wells [9] in parallel proposed a very similar approach to that above, mapping the typing problem to Herbrand constraints. The advantage of using CHRs arises when we extend the approach to handle Haskell-style overloading [31].

Example 2 Consider the following program making use of the Haskell type class `Eq`.

```
data Erk = Erk
class (Eq a)1 where (==) ::  (a->a->Bool)2
instance (Eq a)3 => Eq [a]
f4 x5 = x6 ==7 [Erk]8
```

For simplicity, we left out the instance body. The important point to note is that we also annotate class and instance declarations. The translation to CHRs yields the following.

$$\begin{array}{rcl}
(==)(t) & \Longleftrightarrow & (Eq\ a)_1, (t = a \rightarrow a \rightarrow Bool)_2 \\
Eq\ [a] & \Longleftrightarrow & (Eq\ a)_3 \\
f(t_4) & \Longleftrightarrow & (t_x = t_5)_5, (t_x = t_6)_6, (t_8 = [Erk])_8, (==)(t_7)_7, t_7 = t_6 \rightarrow t_8 \rightarrow t_9, (t_4 = t_5 \rightarrow t_9)_4
\end{array}$$

The first rule represents the (justified) method of the class declaration. The second rule models the instance declaration. We have an instance of `Eq` on type $[a]$ iff we have an instance on type a. Note that the right-hand side of the rule is justified. We refer to [27] for a detailed discussion of how CHRs can be used to model Haskell-style type classes and its various extensions. The last rule (slightly simplified because of infix application) represents the type of `f`.

Type inference for expression $\mathtt{f}_{10}$ proceeds as follows. We use left-hand side symbols of CHRs to index rules.

$$\begin{array}{ll}
 & f(t)_{10} \\
\longrightarrow_f & t = t_4, (t_x = t_5)_{5,10}, (t_x = t_6)_{6,10}, (t_8 = [Erk])_{8,10}, (==)(t_7)_{7,10}, (t_7 = t_6 \rightarrow t_8 \rightarrow t_9)_{10}, \\
 & (t_4 = t_5 \rightarrow t_9)_{4,10} \\
\longrightarrow_{(==)} & t = t_4, (t_x = t_5)_{5,10}, (t_x = t_6)_{6,10}, (t_8 = [Erk])_{8,10}, t_7 = t', (Eq\ a)_{7,10,1}, \\
 & (t' = a \rightarrow a \rightarrow Bool)_{7,10,2}, (t_7 = t_6 \rightarrow t_8 \rightarrow t_9)_{10}, (t_4 = t_5 \rightarrow t_9)_{4,10} \\
\longrightarrow_{Eq} & t = t_4, (t_x = t_5)_{5,10}, (t_x = t_6)_{6,10}, (t_8 = [Erk])_{8,10}, t_7 = t', a = [a'], (Eq\ a')_{7,10,1,3}, \\
 & (t' = a \rightarrow a \rightarrow Bool)_{7,10,2}, (t_7 = t_6 \rightarrow t_8 \rightarrow t_9)_{10}, (t_4 = t_5 \rightarrow t_9)_{4,10}
\end{array}$$

Note that we rename rules before application. In the last step, we replace $(Eq\ a)_{7,10,2}$ by $a = [a'], (Eq\ a')_{7,10,1,3}$ because equality constraints in the store imply that a must be a list type.

We can build the type of `f` by simplifying constraints. That is, apply most general unifier to constraints in final store. This yields `Eq Erk => [Erk]->Bool`. Note that the Hugs system reports

```
ERROR test.hs:10 - Instance of Eq Erk required for definition of f
```

In our system we can ask for an explanation for why the offending class constraint appears. Instead of minimal unsatisfiable subsets we simply search for minimal implicants. Justifications attached to minimal implicants allows us to narrow down the possible source of the unexpected type.

For example, we are interested in why `f` has type `Eq Erk => a` for some `a`. The constraint

$$(t_8 = [Erk])_{8,10}, t_7 = t', a = [a'], (Eq\ a')_{7,10,1,3}, (t' = a \rightarrow a \rightarrow Bool)_{7,10,2}, (t_7 = t_6 \rightarrow t_8 \rightarrow t_9)_{10}$$

is a minimal implicant of $Eq\ Erk$. Hence, we know that locations $\{1, 2, 3, 7, 8\}$ are responsible for the occurrence of $Eq\ Erk$ locations $\{1, 2, 3, 7, 8\}$ are responsible for the occurrence of $Eq\ Erk$. The system reports the following.

```
data Erk = Erk
class Eq a where (==) ::  a->a->Bool
instance Eq a => Eq [a]
f x = x == [Erk]
```

□

In summary, our type inference algorithm generates equational and user-defined constraints out of expressions. Constraints are justified by the program location where they originated from. These locations are retained during CHR solving. Simple reasoning steps on constraints, such as finding minimal unsatisfiable subsets and minimal implicants, allows us to identify problematic program locations. The current paper is an extended version of [28].

Our contributions are:

- We give a translation of the typing problem for Hindley/Milner which includes Haskell-style overloading into CHRs.
- We refine CHR solving by keeping track of justifications, i.e. program locations, attached to constraints.
- We provide formal results of soundness and completeness of our inference scheme via CHR solving.
- We identify some simple constraint reasoning steps as sufficient to locate the minimal set of contributing locations in case of a type error or unexpected result.
- Our approach is the first to:
  - explain the locations that lead to a type having a certain shape,
  - explain subsumption and ambiguity failure,
  - handle Haskell-style overloading (and indeed more complex type extensions [27]).
- We provide an interactive type debugger implementing the above ideas as part of the Chameleon system [30].

The rest of the paper is organized as follows. We first describe the debugging features supported by the Chameleon debugging system, with only informal explanations about how they are implemented. Then we give the formal underpinning to the system. In Section 3 we introduce types and constraints, followed by a formal definition of constraint solving in terms of Constraint Handling Rules (CHRs) in Section 4. We show how to translate a Hindley/Milner typing problem with Haskell-style overloading into a system of CHRs in Section 5, and then how we use this for type inference and checking in Section 6. Our main results are stated in Section 6.1. In Section 7, we discuss how simple constraint reasoning steps support type debugging of programs. Related work is discussed in Section 8. We conclude in Section 9. An implementation of our type debugger is available via [30].

2 The Chameleon Type Debugger

In this section we explain the features of the Chameleon type debugger. The Chameleon system [30] supports a Haskell-style language with its own overloading mechanism [27]. Currently, the Chameleon system does not allow for the debugging of errors in the definitions of type classes and instances.[1] Chameleon does of course allow us to debug the *usage* of classes and instances.

The debugger makes use of two kinds of constraint reasoning. In order to explain why a type error arises it determines minimal unsatisfiable subsets of a set of constraints. In order to explain why an expression has a certain type it determines a minimal implicant of a set of constraints. Details of these operations can be found in Section 7.

2.1 Error Explanation

We can ask for the type of an expression e using the command `type` e. If e has no type this displays the parts of the program which cause the error. It translates e into a set of constraints C. The constraint C is executed with respect to the CHRs P of the translated program, by exhaustively

[1] To do so requires a well-understood check of the confluence of the CHRs [1]. This is straightforward for CHRs arising from Haskell 98 classes and instances, but termination issues arise when arbitrary programmed type extensions are allowed. Currently, we also do not check for the monomorphism restriction and some other Haskell 98 specific context restrictions.

applying the rules in P to obtain a new constraint C'. We denote this execution by $C \longrightarrow_P^* C'$. If C' is satisfiable it displays the type of e. Otherwise the system determines a minimal unsatisfiable subset of C' (simply the first one detected) and displays the justifications for that set.

We support two basic approaches to displaying the justifications of an error.

Local explanation restricts attention to the expression for which the type error is detected, all locations outside this expression are ignored. If the expression is a single function name, we restrict attention to the function definition. Local explanation is useful for top-down exploratory style of type debugging which we believe is more natural. Indeed while using local explanations the system in fact simplifies all constraints arising from each other function, which considerably simplifies the calculation of minimal unsatisfiable subsets and minimal implicants.

Example 3 Returning to Example 1, using local explanation the CHR for $f(t)$ is treated as $f(t) \iff t = Bool \rightarrow Bool$ and the minimal unsatisfiable subset is $(t_1 = Char)_{\{1.8\}}$, $(t_2 = Bool \rightarrow Bool)_{\{2,8\}}$, $(t_2 = t_1 \rightarrow t_3)_{\{3,8\}}$. The resulting justification 1, 2, and 3 is the same as before (restricted to locations in p), but less constraints are generated. The resulting explanation is `p = f 'a'` □

Global explanation is more expensive to compute but allows the user to explore an error in a bottom-up manner. Here all the justification information is used to determine an error. The system highlights positions involved in one minimal unsatisfiable subset, and can highlight those positions that occur in all minimal unsatisfiable subsets differently from those not occurring in all.

Example 4 Consider the following program:

```
foldl f z []     = [z]
foldl f z (x:xs) = foldl f (f z x) xs
flip f x y = f y x
reverse = foldl (flip (:))  []
palin xs = reverse xs == xs
```

where `palin` is intended to be a function from a list to a Boolean value. It should return `True` if its argument is a palindrome. Hugs [16] reports the following:

```
ERROR Ex1.hs:6 - Type error in application
*** Expression     :  xs == reverse xs
*** Term           :  xs
*** Type           :  [a]
*** Does not match :  [[a]]
*** Because        :  unification would give infinite type
```

telling us no more than that there is an error within `palin`.

By using global error explanation, we can get an immediate picture of the program sites which interact to cause this error. In the following debugger query, one global explanation of the type error is given by the underlined code. Locations which appear in all minimal unsatisfiable subsets are underlined twice, while those which only appear in the selected minimal unsatisfiable subset are underlined once. Note that the "real" cause of the error does occur in all unsatisfiable subsets, and in our experience this is usually the case where there is one "real" error.

```
Ex1.hs> :set global
Ex1.hs> :type palin
type error - contributing locations
foldl f z []     = [z]
foldl f z (x:xs) = foldl f (f z x) xs
```

```
flip f x y = f y x
reverse = foldl (flip (:)) []
palin xs = reverse xs == xs
```

By starting from any of these sites, the programmer is able to work towards the true cause of the error - in any direction. In this case, the problem is that the first clause of `foldl` should not return a list, `[z]`, but rather `z`. Correspondingly, the offending location is double-underlined.

If we had taken a local approach to error explanation here, the result would have been specific to only the definition of `palin`. We would then have to proceed in a top-down fashion, from definition to definition, towards the offending expression.

□

The system naturally handles type class extensions that can be expressed by CHRs.

Example 5 Functional dependencies in class constraints are useful for preventing ambiguity. Consider a multi-parameter collection class `Collect a b` where type `b` is a collection of elements of type `a`. The class definition is

```
class Collect a b where
      empty ::  b
      insert ::  a -> b -> b
      member ::  a -> b -> Bool
```

As defined this class is flawed, since the type of `empty :: Collect a b => b` is ambiguous. Type variable `a` appears only in the constraint component. This leads to difficulties when implementing Haskell-style overloading [19]. Functional dependencies allow us to overcome this problem, by stating that `b` functionally determines `a`. Hugs [16] supports functional dependencies.

```
class Collect a b | b -> a where ...
```

This functional dependency can be expressed by a CHR *propagation rule*.

$$Collect\ a\ b,\ Collect\ a'\ b \Longrightarrow a = a'$$

The $\Longrightarrow$ symbol is read as logical implication. Operationally the rule is read as, if you have a match for the left hand side you may add the right hand side. The above rule states that if there are two `Collect` constraints with the same second argument, we enforce that their first arguments are identical.

Consider the following program which tries to check if a `Float` is a member of a collection of `Int`s.

```
f g x y = if member (x::Float) (insert (1::Int) y)
          then g x else y
```

The constraints for f imply $Collect\ Int\ t$ and $Collect\ Float\ t$ which causes the propagation CHR to fire adding the information that $Int = Float$ causing a type error to be detected. The justification of the error is reported as:

```
Ex9.hs> :type f
type error - contributing locations
f g x y = if member (x::Float) (insert (1::Int) y)
          then g x else y
rule(s) involved:  Collect a b, Collect a' b ==> a = a'
```

The system could be straightforwardly extended to report the source of the CHR involved—the functional dependency `b -> a`. □

Example 6 A strength of our system is to be able to support almost arbitrary type class extensions. This is made possible through the extensible type system [27] underlying our approach. Consider

```
f x y = x / y + x ‘div‘ y
```

The inferred type is `f :: (Integral a, Fractional a) => a -> a -> a` rather than immediately causing a type error. We would like to state that the `Integral` and `Fractional` classes must be disjoint. This can be expressed via the following CHR.

```
Integral a, Fractional a ⟹ False
```

Then, the type debugger reports the following.

```
Ex10.hs> :t f
type error - contributing locations
f x y = x / y + x ‘div‘ y
rule(s) involved:  Integral a, Fractional a ==> False
```

□

2.2 Type Explanation

Another important feature of the debugger is to explain how various types arise, even when there are no type errors. This allows the user to ask "why does this expression have such a type"? We can ask to explain the type t of expression e using the command `explain` (e) $(D \Rightarrow t)$. The system builds the constraints C for expression e and executes $C \longrightarrow_P^* C'$ and then checks whether $C' \supset \tilde{\exists}_{t_e}(t_e = t, D)$ where t_e is the type variable corresponding to e and $\tilde{\exists}_{t_e}$ quantifies everything except t_e. That is the inferred type for e is stronger than that we are asking to explain. If this is the case it determines a minimal subset of C' which causes the implication and displays the set of justifications for this set, in a global or local fashion just as for type error explanation.

In the following example we use this capability to explain an error arising from a missing instance.

Example 7 Consider the following program illustrating a classic beginners error with Haskell

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

The Hugs system generates the error

```
ERROR Ex11.hs:9 - Illegal Haskell 98 class constraint in inferred type
*** Expression :  sum
*** Type       :  Num [a] => [[a]] -> [a]
```

The inferred type has a class constraint that is non-variable and has no instance. This is completely opaque to a beginning Haskell programmer.

We can generate an explanation for the error by looking for a reason for (e.g. minimal set of constraints implying) the constraint `Num [a]`. Asking the type debugger

```
Ex11.hs> :explain sum (Num [_] => _)
sum [] = []
sum (x:xs) = x + sum xs
```

Clearly indicating the problem arises from the `[]` of the body of the first rule interacting with `+` and the recursive call to `sum`. □

Example 8 Returning to Example 5, the Hugs error message is

```
ERROR Ex9.hs:10 - Constraints are not consistent with
functional dependency
*** Constraint       :  Collects Float a
*** And constraint   :  Collects Int a
*** For class        :  Collects a b
*** Break dependency :  b -> a
```

This gives very little information to the programmer about the error. In our system we can ask where the constraints arise from:

```
Ex9.hs> :explain f (Collects Float _ => _)
f g x y = if member (x::Float) (insert (1::Int) y)
          then g x else y
```

Note that even though the constraint system is unsatisfiable, a minimal implicant correctly determines a useful justification of the constraint. □

2.3 Subsumption and Ambiguity Explanation

We also support two further forms of explanations which arise in the context of user-provided type annotations and Haskell-style overloading.

2.3.1 Subsumption Failure

Chameleon supports explicit type annotations of the form `f::C=>t`. We must ensure that the annotated type is "subsumed" by the inferred type. In other words, the annotated type cannot be more general than the (most general) inferred type. Our aim is to provide enough information to the user to resolve such a problem if it occurs.

When a subsumption error occurs, it is because a stronger type has been inferred than that provided. We highlight those locations in the program which cause this stronger type to be inferred.

Example 9 The following program has been annotated with a type which is too general.

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

The inferred type of `insert` is `Ord a => a -> [a] -> [a]`. The `Ord` constraint is missing from the declared type. Chameleon reports:

```
insert.ch:1:  ERROR: inferred type does not subsume declared type
declared:  a -> [a] -> [a] inferred:  Ord(a) => a -> [a] -> [a]

insert ::  a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x ≥ y = y :  insert x ys
                | otherwise = x :  y :  ys
```

This indicates that the problem lies with the use of `>`. We cannot use `>` in `insert` with that declared type.

□

Note that typically this form of error reporting will cause the debugger to highlight only locations in a function's definition - we trust that the type declaration accurately reflects the programmer's intended type for the function, and we highlight the locations which disagree. It is possible however, that information in the declared type itself might contribute to the problem. In such a case, the problematic parts of the declared type will also be highlighted.

Example 10 Consider the following program:

```
f ::  a -> b -> (c,d)
f x y = x
```

The annotation given to `f` is too general. Clearly if `x` has type `(c,d)`, then it cannot also have type `a`, as declared. As such, there is a conflict between the type declaration and the program.

Chameleon reports:

```
f ::  a -> b -> (c, d)
f x y = x
```

This shows that the problem involves the two occurrences of `x` and the distinct types they have been given.

In this case either the type or the definition may be altered to resolve the problem. □

2.3.2 Ambiguity

In Haskell, a type scheme is ambiguous if there are variables which appear in its context, but not in its type. Typically these ambiguities are resolved by the programmer providing an explicit type declaration at some point in the program. We are interested in reporting program sites whose types include variables which appear within ambiguous constraints. By highlighting those locations, we can direct the program to the sites which either need to be reworked or type-annotated.

Example 11 Consider the following program which has an ambiguous type.

```
p u = (show x, f x)
      where x = read u
            y = x
            f = undefined
```

The inferred type of `p` is `(Read(b), Show(b)) => [Char] -> ([Char],a)`. Notice how variable `b` does not occur in the type component, only in the constraints, hence the type is ambiguous. To resolve this problem, the programmer must know which program sites are responsible for the ambiguous type variables. They can then be annotated with specific types - indicating the actual instances of `show` and `read` that are required.

When this program is loaded, Chameleon will report:

```
p u = (show x, f x)
      where x = read u
            y = x
            f = undefined
```

This indicates that some of those locations have to be altered to fix the program. For example, replacing `x = read u` by

```
x :: String
x = read u
```

would be sufficient. □

Note that Haskell's monomorphism restriction is not enforced in Chameleon. Consequently, the type of `x` in the above program is `forall a. Read a => a`. In Haskell it would have the monomorphic type `Read a => a`. This distinction is significant, since it means that the Chameleon type debugger may not find all locations whose types contain variables that Haskell would consider ambiguous.

Example 12 Returning to the previous program, we note that since `x` has a monomorphic type in Haskell, it would be possible to fix the ambiguity problem by annotating any occurrence of `x`. If Chameleon supported the monomorphism restriction, it would highlight the following sites when reporting the ambiguity error.

```
p u = (show x, f x)
      where x = read u
            y = x
            f = undefined
```

Indeed, adding a type annotation to the definition of `y` would resolve the problem in Haskell, but not in Chameleon. □

2.4 Referring to Local Variables

In order to track down type errors interactively it is often useful to be able to find the types and explain the types of variables local to a function definition. Current interactive Haskell systems only permit references to variables bound at the top-level. The debugger allows the syntax $f;r$ to refer to variable r local to the definition of f. If there are multiple equations (patterns) for the definition for f we can select one using the notation $f;n;r$ where n is an integer pattern number. By default if there are multiple equations, and no pattern number is given, the first where the local variable exists is used. Local variables inside nested scopes can also be referred to.

Example 13 Consider the program

```
f  (x:xs1)  _  = True
f  xs2      ys = let h xs3 = xs ++ xs
                     g ys = ys ++ xs
                 in h ys ++ g (xs ++ ys)
```

Then `f;1;xs` refers to $\texttt{xs}_1$, while `f;2;xs` refers to $\texttt{xs}_2$. By default `f;xs` refers to $\texttt{xs}_1$. In addition `f;2;h;xs` and `f;h;xs` refer to $\texttt{xs}_3$. □

2.5 Type Addition

While the `explain` command allows users to ask *why* a location has a type of a particular shape, the `declare` command allows users to ask *why not* of a location and a type. The `declare` f $(C \Rightarrow t)$ command adds constraints $x = t, C$ where x is the type of f to the CHR program defining f.

Example 14 Returning to Example 7, we can get another explanation for the erroneous type of `sum` by adding the expected type.

```
Ex11.hs> :declare sum ([Int] -> Int)
Ex11.hs> :type sum
type error - contributing locations
sum :: [Int] -> Int
sum [] = []
```

we are shown those locations which are in conflict with the newly declared type, as well as the parts of the added type which conflict. □

2.6 Source-Based Debugger Interface

Although an interactive debugging system provides users with the means to quickly pose a number of consecutive queries, narrowing in on the target, it might also be viewed as a slightly heavy handed interface to the debugger. An interactive system necessarily interrupts the typical, edit-compile programming cycle, which may be distracting. Furthermore, it may at times seem quite awkward to keep type exploration separate from the program source itself.

To this end we have provided an alternative means to interact with the debugger, by allowing for commands to appear naturally within the source program. At this time we have support for `type` e where e is an expression written e`::?`. And we support `explain` e $(D \Rightarrow t)$, where e is an expression and $D \Rightarrow t$ is a type scheme, written e`::?`D `=>` t.

Entire declarations can be queried by writing such a command at the same scope as the declaration (with a declaration name in place of an expression.) These queries are collected and processed in textual order. They do not effect the semantics of the program in which they are embedded, merely the compiler's output.

Example 15 Consider the following, modified, snippet of the program presented in Example 4.

```
reverse ::?
reverse = foldl (flip (:))  []
```

When we attempt to compile this code, using the non-interactive system, we would get, in addition to the usual type error message, the following output:

```
reverse ::  [a] -> [[a]]
```

Further modification of the program might lead to the following, which involves an `explain`-style query:

```
reverse ::?
reverse = (foldl (flip (:))  []) ::?  _ -> [[_]]
```

The corresponding output would be:

```
reverse ::  [a] -> [[a]]

foldl (flip (:))  [] ::?  _ -> [[_]]
because of:  foldl (flip (:)) []
```

□

2.7 Declarative Debugging Interface

Chameleon also includes a simplistic declarative debugging interface. We can invoke the declarative debugging interface on an expression e with a type error using the command `debug` e. The declarative debugger works like a declarative debugger for a logic program [26], localizing the error by finding a function whose type is wrong, but all the functions used in its definitions are correct. A similar feature is also provided by [15, 3].

$$\text{(Var)}\quad C,\Gamma \vdash v : \sigma \quad (v : \sigma \in \Gamma) \qquad\qquad \text{(Let)}\quad \frac{C,\Gamma \vdash e : \sigma \qquad C,\Gamma.x : \sigma \vdash e' : \tau'}{C,\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'}$$

$$\text{(Abs)}\quad \frac{C,\Gamma.x : t_1 \vdash e : t_2}{C,\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad\qquad \text{(App)}\quad \frac{C,\Gamma \vdash e_1 : t_1 \to t_2 \quad C,\Gamma \vdash e_2 : t_1}{C,\Gamma \vdash e_1 e_2 : t_2}$$

$$(\forall\ \text{Intro})\quad \frac{C \wedge D,\Gamma \vdash e : t \quad \bar{a} \notin fv(\Gamma, C)}{C,\Gamma \vdash e : \forall \bar{a}.D \Rightarrow t} \qquad\qquad (\forall\ \text{Elim})\quad \frac{C,\Gamma \vdash e : \forall \bar{a}.D \Rightarrow t' \quad F \models C \supset [\bar{t}/\bar{a}]D}{C,\Gamma \vdash e : [\bar{t}/\bar{a}]t'}$$

Figure 1: Hindley/Milner with Constraints

Example 16 Consider the program of Example 4 once more. The declarative debugger trace might be

```
Ex1.hs> :debug palin
reverse ::  [a] -> [[a]]
Ex1.hs:  is this type correct> n
flip ::  (a -> b -> c) -> b -> a -> c
Ex1.hs:  is this type correct> y
foldl ::  (a -> b -> a) -> a -> [b] -> [a]
Ex1.hs:  is this type correct> n
type error - contributing locations
foldl f z [] = [z]
foldl f z (x:xs) = foldl f (f z x) xs
```

□

The declarative debugging interface, chooses a minimal unsatisfiable subset, and asks the user about types of functions involved in this set, from the top down to discover where the error actually lies. It then shows the justifications of the error in this function. Note that since it uses a minimal unsatisfiable subset, it will never ask questions about functions not involved in this subset. This is not the case for the system of [3], since it does not determine minimal unsatisfiable subsets.

3 Types and Constraints

We consider an extension of the Hindley/Milner system with constraints.

Expressions	e	$::=$	$f \mid x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ f = e\ \mathsf{in}\ e$
Types	t	$::=$	$a \mid t \to t \mid T\ \bar{t}$
Type Schemes	σ	$::=$	$\tau \mid \forall \bar{\alpha}.C \Rightarrow t$
Constraints	C	$::=$	$t = t \mid U\ t \mid C \wedge C$

W.l.o.g., we assume that λ-bound and let-bound variables have been α-renamed to avoid name clashes. We commonly use $x, y, z, \ldots$ to refer to λ-bound variables and $f, g, h, \ldots$ to refer to user- and pre-defined functions. Both sets of variables are recorded in a variable environment Γ. Note that we consider Γ as an (ordered) list of elements, though we commonally use set notation. We denote by $\{x_1 : \sigma_1, \ldots, \sigma_n : t_n\}.\sigma : t$ the environment $\{x_1 : \sigma_1, \ldots, \sigma_n : t_n, \sigma : t\}$.

Our type language consists of variables a, type constructors T and type application, e.g. $T\ a$. We use common notation for writing function and list types. A type scheme is of the form $\forall \bar{a}.C \Rightarrow t$ where $\bar{a}$ refers to the set of bound variables, C is a set of constraints and t is a type. When C is omitted it is considered to be *True*.

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. A user-defined constraint is one of $U\ t_1\ \cdots\ t_n$ where U is a predicate symbol and $t_1, \ldots, t_n$ are types, or $p(t)$ where p is a predicate symbol and t a type. The reason for the two forms of user-defined constraints is simply to have different notation for user-defined constraints for indicating the types of functions, and user-defined constraints specifying some other program properties.

Conjunctions of constraints are often treated 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. [21].

We consider the standard Hindley/Milner system extended with constraints. The typing rules (Figure 1) are essentially the ones from HM(X) [25, 29]. In rule (Var), we assume that v either refers to a λ- or let-bound variable. In rule ($\forall$ Intro), we build type schemes by pushing in the "affected" constraints. Note that we slightly deviate from the standard HM(X) ($\forall$ Intro). However, the current rule is good enough for a lazy language. We refer to [25] for a detailed discussion. In rule ($\forall$ Elim), we assume that F refers to a first-order formula specifying relations among user-defined constraints, $\models$ denotes the model-theoretic entailment relation and $\supset$ stands for logical implication. We generally assume that F can be described by a set of CHRs.

4 Constraint Handling Rules with Justifications

We will translate the typing problem to a constraint problem where the meaning of the user-defined constraints is defined by Constraint Handling Rules (CHRs) [6]. CHRs manipulate a global set of primitive constraints, using rewrite rules of two forms

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

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

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

In our use of the rules, constraints occurring on the right hand side of rules have attached justifications (program locations). We extend the usual derivation steps of Constraint Handling Rules 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 union of the justifications of $E \cup D$.

Then we create a new 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 shorthand 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 original justifications of each d_i. The other constraints (the equality constraints arising from the match) are given empty justifications. Indeed, this is sufficient. The connection to the original location in the program text is retained by propagating justifications to constraints on the rhs only.

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 repropagation see e.g. [1].

5 Translation to Constraint Handling Rules

Our approach to type inference follows [4] by translating the typing problem into a constraint problem. However, in contrast to [4] where translation results in a set of Horn clauses, we map the typing problem to a set of Constraint Handling Rules (CHRs) [6].

For each definition `f = e`, we introduce a CHR of the form $f(t, l) \iff C$. The type parameter t refers to the type of `f` whereas l refers to the set of types of λ-bound variables in scope. The reason for l is that we must ensure that λ-bound variables remain monomorphic. The constraint C contains the constraints generated out of expression `e` plus some additional constraints restricting l. We use list notation (on the level of types) to refer to the "set" of types of λ-bound variables. In order to avoid confusion with lists of values, we write $\langle l_1, \ldots, l_n \rangle$ to denote the list of types $l_1, \ldots, l_n$. We write $\langle l | r \rangle$ to denote the list of types with head l and tail r.

The following example provides some details about our translation scheme in case of nested definitions. The basic idea is to employ some form of λ-lifting on the level of types.

Example 17 Consider

```
k z = let h w = (w,z)
          f x = let g y = (x,y)
                in (g 1, g True, h 3)
      in f z
```

In a first attempt, a (partial) description of the resulting CHRs might look as follows. For simplicity, we leave out the constraints generated out of expressions. We commonly write t_x to denote the type of λ-bound variable `x`.

$$\begin{array}{lll} \text{(k)} & k(t,l) & \iff \quad l = \langle\rangle, \ldots \\ \text{(h)} & h(t,l) & \iff \quad l = \langle t_z \rangle, \ldots \\ \text{(f)} & f(t,l) & \iff \quad l = \langle t_z \rangle, \ldots \\ \text{(g)} & g(t,l) & \iff \quad l = \langle t_z, t_x \rangle, \ldots \end{array}$$

Note that the λ parameter l refers exactly to the set of types of all free (λ) variables in scope. Hence, at instantiation sites we need to specify correctly the set of types of λ-bound variables in

scope. Consider expression. `(g 1, g True, h 3)`. Among others, we generate (justifications are omitted for simplicity)

$$\begin{array}{l} g(t_1, l_1), l_1 = \langle t_z, t_x \rangle, t_1 = Int \rightarrow t'_1, \\ g(t_2, l_2), l_2 = \langle t_z, t_x \rangle, t_2 = Bool \rightarrow t'_2, \\ h(t_3, l_3), l_3 = \langle t_z \rangle, t_3 = Int \rightarrow t'_3, \ldots \end{array}$$

It seems unnecessary to keep track of the exact set of types of λ-variables for each function definition. Indeed, a simple trick allows us to maintain only a list of λ-variables for all functions. The set of types of lambda-bound variables in scope for function definitions is simply left "open". The set of types of lambda-bound variables at function instantiation sites corresponds to the "full" set of types of lambda-bound variables in scope. Our actual translation yields the following result.

$$\begin{array}{llll} \text{(k)} & k(t,l) & \Longleftrightarrow & l = r, t = t_1 \rightarrow t_2, f(t, l_1), l_1 = \langle t_z \rangle, t_1 = t_z \\ \text{(h)} & h(t,l) & \Longleftrightarrow & \underline{l = \langle t_z | r \rangle}, t = t_w \rightarrow (t_w, t_z) \\ \text{(f)} & f(t,l) & \Longleftrightarrow & \overline{l = \langle t_z | r \rangle}, t = (t'_1, t'_2, t'_3), g(t_1, l_1), \\ & & & l_1 = \langle t_z, t_x \rangle, t_1 = Int \rightarrow t'_1, \\ & & & g(t_2, l_2), l_2 = \langle t_z, t_x \rangle, t_2 = Bool \rightarrow t'_2, \\ & & & h(t_3, l_3), \underline{\underline{l_3 = \langle t_z, t_x \rangle}}, t_3 = Int \rightarrow t'_3 \\ \text{(g)} & g(t,l) & \Longleftrightarrow & l = \langle t_z, t_x | r \rangle, t = t_y \rightarrow (t_x, t_y) \end{array}$$

For example, in rule (h) we require that variable `z` is in scope plus possibly some more variables (see underlined constraint). Please observe that in rule (f), we pass in the (somewhat redundant) variable t_x as part of the l parameter at the instantiation site of `h` (see double-underlined constraint). There is no harm in doing so, because there is no reference to variable t_x on the right hand side of rule (h). □

The translation of the typing problem consists of two procedures for generating constraints out of expressions and generating CHRs for function definitions. We assume that individual expressions are annotated with unique numbers, i.e. program locations.

Constraint generation is formulated as a logical deduction system with clauses of the form $\Gamma, e \vdash_{Cons} (C \mid t)$ where environment Γ and expression e are input parameters and constraint C and type t are output parameters. See Figure 2 for details. For example, in rule (Var-f) we generate an "instantiation" constraint. The constraint $f(t,l), l = \langle t_{x_1}, \ldots, t_{x_n} \rangle$ demands on instance of `f` on type t where $(t_{x_1}, \ldots, t_{x_n})$ refers to the set of types of λ-bound variables in scope. The actual type of `f` will be described by a CHR where the set of types of λ-bound variables is left open. Note that the order of types of lambda-bound variables matters.

Generation of CHRs is formulated as logical deduction system with clauses of the form $\Gamma, e \vdash_{Cons} P$ where environment Γ and expression e are input parameters and the set P of CHRs is the output parameter. See Figure 3 for details.

In the following, we discuss how to adjust our translation scheme in case of some type extensions. For brevity we omit the (uninteresting) l argument for λ-bound variables, whose role is orthogonal to these extensions.

5.1 Type Annotations

We add explicit type annotations to our language of expressions.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \mathsf{let} \begin{array}{c} f :: \sigma \\ f = e \end{array} \mathsf{in}\ e$$

Note that $(e :: \sigma)$ can be viewed as syntactic sugar for

$$\mathsf{let} \begin{array}{c} f :: \sigma \\ f = e \end{array} \mathsf{in}\ f$$

$$\text{(Var-x)} \quad \frac{(x : t_1) \in \Gamma \quad t_2 \text{ fresh}}{\Gamma, x_l \vdash_{Cons} ((t_2 = t_1)_l \mid t_2)}$$

$$\text{(Var-f)} \quad \{x_1 : t_{x_1}, \ldots, x_n : t_{x_n}\}, f_l \vdash_{Cons} (f(t, l)_l, l = \langle t_{x_1}, \ldots, t_{x_n} \rangle \mid t)$$

$$\text{(Abs)} \quad \frac{\Gamma.x : t_1, e \vdash_{Cons} (C \mid t_2) \quad t_1, t_3, t_4 \text{ fresh}}{\Gamma, (\lambda x_{l_1}.e)_{l_2} \vdash_{Cons} (C, (t_3 = t_4 \rightarrow t_2)_{l_2}, (t_1 = t_4)_{l_1} \mid t_3)}$$

$$\text{(App)} \quad \frac{\Gamma, e_1 \vdash_{Cons} (C_1 \mid t_1) \quad \Gamma, e_2 \vdash_{Cons} (C_2 \mid t_2) \quad t_3 \text{ fresh}}{\Gamma, (e_1\ e_2)_l \vdash_{Cons} (C_1, C_2, (t_3 = t_1 \rightarrow t_2)_l \mid t_3)}$$

$$\text{(Let)} \quad \frac{\Gamma, e_2 \vdash_{Cons} (C \mid t)}{\Gamma, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Cons} (C \mid t)}$$

Figure 2: Justified Constraint Generation

$$\text{(Var)} \quad \Gamma, v \vdash_{Def} \emptyset$$

$$\text{(Abs)} \quad \frac{t \text{ fresh} \quad \Gamma.x : t, e \vdash_{Def} P}{\Gamma, \lambda x.e \vdash_{Def} P}$$

$$\text{(App)} \quad \frac{\Gamma, e_1 \vdash_{Def} P_1 \quad \Gamma, e_2 \vdash_{Def} P_2}{\Gamma, e_1\ e_2 \vdash_{Def} P_1 \cup P_2}$$

$$\text{(Let)} \quad \frac{\begin{array}{c}\Gamma, e_1 \vdash_{Def} P_1 \quad \Gamma, e_2 \vdash_{Def} P_2 \\ \Gamma, e_1 \vdash_{Cons} (C \mid t) \quad \Gamma = \{x_1 : t_1, \ldots, x_n : t_n\} \quad r \text{ fresh} \\ P = P_1 \cup P_2 \cup \{f(t, l) \Longleftrightarrow C, l =< t_1, \ldots . t_n | r >\}\end{array}}{\Gamma, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Def} P'}$$

Figure 3: Rule Generation for Hindley/Milner

where f is a fresh identifier. The typing rule for the additional language construct is standard.

$$\text{(Let-Annot)} \quad \frac{C, \Gamma \vdash e : \sigma \qquad C, \Gamma.f : \sigma \vdash e' : \tau'}{C, \Gamma \vdash \mathsf{let}\ \begin{array}{c} f :: \sigma \\ f = e \end{array}\ \mathsf{in}\ e' : \tau'}$$

As it is common, we often leave universal quantifiers implicit. That is, $f :: C \Rightarrow t$ is a short-hand for $f :: \forall \bar{a}.C \Rightarrow t$ where $\bar{a} = fv(C, t)$.

Type inference for type annotations generates two CHRs instead of one.

$$\text{(Let-Annot)}\quad \frac{\begin{array}{c}\Gamma, e_1 \vdash_{Cons} (C \mid t) \quad \Gamma = \{x_1 : t_1, \ldots, x_n : t_n\} \quad r \text{ fresh} \\ \Gamma, e_2 \vdash_{Def} P \\ P' = P \cup \{f_a(t,l) \Longleftrightarrow (t = t_a)_i, (C_a)_i\} \cup \\ \{f(t,l) \Longleftrightarrow f_a(t,l), C, l =< t_1, \ldots . t_n | r >\}\end{array}}{\Gamma, \mathsf{let} \begin{array}{l}(f :: C_a \Rightarrow t_a)_i \\ f = e_1\end{array} \mathsf{in}\ e_2 \vdash_{Def} P'}$$

We will need to check that the annotated type (represented by f_a) is "subsumed" by the inferred type (represented by f). Details will be discussed in Section 6.2.

Example 18 Consider the program

```
(g ::  [Char] -> Bool)1
g8 ((x2:3)4xs5)6 = True7
```

The CHR generated for g from the annotation is simply

$$g_a(x) \Longleftrightarrow (x = [Char] \rightarrow Bool)_1$$

The CHR generated for g is

$$\begin{array}{rl} g(t_8) \Longleftrightarrow & (t_2 = t_x)_2, (t_3 = a \rightarrow [a] \rightarrow [a])_3, \\ & (t_3 = t_2 \rightarrow t_4)_4, (t_5 = t_{xs})_5, (t_4 = t_5 \rightarrow t_6)_6, \\ & (t_7 = Bool)_7, (t_8 = t_6 \rightarrow t_7)_8, g_a(t_8) \end{array}$$

□

5.2 Multiple Clauses

The (possibly multiple) definitions for a single function are joined using another CHR. If there are m definitions of f, numbered $f_{i_1}, \ldots, f_{i_m}$ then the final CHR rule for f is

$$f(x) \Longleftrightarrow f_{i_1}(x), \ldots, f_{i_m}(x)$$

Note the lack of justifications, which will be collected from the rules for $f_{i_1}, \ldots, f_{i_n}$.

5.3 Recursive Functions

We consider recursive functions.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \mathsf{letmono}\ f = e\, \mathsf{in}\ e \mid \mathsf{letpoly} \begin{array}{l} f :: \sigma \\ f = e \end{array} \mathsf{in}\ e$$

For simplicity, we introduce distinct language construct for monomorphic and polymorphic recursive functions. Note that we require explicit type annotations for polymorphic recursive functions to ensure decidable type inference [14]. Note that in our implementation and example programs, we only make use of one "let" construct. We can distinguish between monomorphic, polymorphic recursive and other let-defined functions by a simple dependency analysis and checking for the presence of explicit type annotations.

The typing rules are standard.

$$\text{(Let-Mono)} \qquad \frac{C,\Gamma \vdash et \qquad C,\Gamma.f : t \vdash e' : t'}{C,\Gamma \vdash \mathsf{letmono}\ f = e\,\mathsf{in}\ e' : t'}$$

$$\text{(Let-Poly)} \qquad \frac{C,\Gamma.f : \sigma \vdash e : \sigma \qquad C,\Gamma.f : \sigma \vdash e' : \tau'}{C,\Gamma \vdash \mathsf{letpoly}\ \begin{array}{c} f :: \sigma \\ f = e \end{array}\ \mathsf{in}\ e' : \tau'}$$

We illustrate the necessary changes for rule generation by some examples. If we were to naively apply the scheme outlined before to the translation of recursive programs, our type inference procedure would become undecidable. In short, a CHR derivation, in such a situation, would never terminate.

Example 19 Consider the program:

```
f (x, y) = g (x, y)
g (x, y) = f (y, x)
```

Applying the standard translation, we would generate something like the following.

$$\begin{array}{lcl} f(t_f) & \Longleftrightarrow & g(t_g), t_g = (t_x, t_y) \rightarrow t_r, t_f = (t_x, t_y) \rightarrow t_r \\ g(t_g) & \Longleftrightarrow & f(t_f), t_f = (t_y, t_x) \rightarrow t_r, t_g = (t_x, t_y) \rightarrow t_r \end{array}$$

Note that these rules are simplified versions of what our translation scheme would actually generate. If we were to attempt a CHR derivation involving either rule, it is clear that it would never terminate. □

To circumvent this problem, we enforce monomorphic recursion for functions without type annotations in Chameleon. Consequently, this allows us to replace the user constraints involved in cycles with monomorphic types. We do this by unfolding cyclical user constraints.

Example 20 We return to the two CHRs generated above, with the knowledge that the calls to f and g within their bodies are involved in a cycle.

We begin with the rule for f above, and apply the simplification rule for g to the rhs constraints, obtaining the following:

$$\begin{array}{lcl} f(t_f) & \Longleftrightarrow & \mathbf{f(t'_f), t'_f = (t'_y, t'_x) \rightarrow t'_r, t'_g = (t'_x, t'_y) \rightarrow t'_r,} \\ & & \mathbf{t_g = t'_g}, t_g = (t_x, t_y) \rightarrow t_r, t_f = (t_x, t_y) \rightarrow t_r \end{array}$$

The newly added constraints are shown in boldface. The constraint $t_g = t'_g$ represents the matching of the type in the g user constraint and the type in the head of the g rule. Finally, to break the cycle we replace the call to f with a constraint asserting that t'_f is the same type as we have already found for f - hence the monomorphism.

$$\begin{array}{lcl} f(t_f) & \Longleftrightarrow & \mathbf{t'_f = t_f}, t'_f = (t'_y, t'_x) \rightarrow t'_r, t'_g = (t'_x, t'_y) \rightarrow t'_r, \\ & & t_g = t'_g, t_g = (t_x, t_y) \rightarrow t_r, t_f = (t_x, t_y) \rightarrow t_r \end{array}$$

The same procedure would then be carried out for the g rule. □

We are able to type polymorphic recursive programs, given that the programmer has supplied sufficient type declarations.

Example 21 Consider the function:

```
f ::  [a] -> Bool
f [] = True
f (x:xs) = f [xs]
```

Note that the type annotation is necessary. In such a case, our translation (simplified) yields the following.

$$\begin{array}{lcl} f_a(t) & \Longleftrightarrow & t = [a] \rightarrow Bool \\ f(t) & \Longleftrightarrow & f_1(t), f_2(t) \\ f_1(t) & \Longleftrightarrow & t = [a] \rightarrow Bool \\ f_2(t) & \Longleftrightarrow & t = [a] \rightarrow t_1, t_2 = [[a]] \rightarrow t_1, f_a(t_2) \end{array}$$

To break the cycle, we employ the annotated CHR for the recursive call. □

In general, rule generation for polymorphic recursive function is the same as for type annotated let definitions. In case of monomorphic recursive functions, we need to unfold CHRs.[2] Constraint generation needs to adjusted in case of rule (Var-f) in Figure 2. We need to distinguish between function identifiers with and without type annotation.

$$\text{(Var-f)} \quad \frac{f \text{ has no type annotation}}{\{x_1 : t_{x_1}, \ldots, x_n : t_{x_n}\}, f_l \vdash_{Cons} (f(t,l)_l, l = \langle t_{x_1}, \ldots, t_{x_n} \rangle \mid t)}$$

$$\text{(Var-f-Annot)} \quad \frac{f \text{ has a type annotation}}{\{x_1 : t_{x_1}, \ldots, x_n : t_{x_n}\}, f_l \vdash_{Cons} (f_a(t,l)_l, l = \langle t_{x_1}, \ldots, t_{x_n} \rangle \mid t)}$$

5.4 Overloading

For an in-depth treatment of the translation of Haskell-style class and instance declarations to CHRs we refer the interested reader to [27, 8].

The translation from declarations to CHRs is

$$\begin{array}{ll} \texttt{instance}\ (C\ \texttt{=>}\ TC\ t)_{l_0}\ \texttt{where} & TC\ t \Longleftrightarrow C_{l_0} \\ \quad f\ \texttt{=}\ e & \\ \texttt{class}\ (C\ \texttt{=>}\ TC\ x)_{l_1}\ \texttt{where} & TC\ x \Longrightarrow C_{l_1} \\ \quad f\ \texttt{::}\ (C \Rightarrow t)_{l_2} & f_a(y) \Longleftrightarrow y = t, C_{l_2}, (TC\ x)_{l_2} \end{array}$$

The appropriate location (l_0, l_1 or l_2) is added as justification to all constraints on the right hand side. A missing constraint C is treated as *True*. Note that we use upper-case letters for user-defined type class constraints, and lower-case letters for user-defined constraints referring to function definitions.

Example 22 Given the class and instance declarations below,

```
class (Eq a)50 where
    (==) ::  a -> a -> Bool51
class (Eq a => Ord a)52 where
    (>) ::  a -> a -> Bool53
instance (Ord a => Ord [a])54 where
    []     > _      = False
```

[2] In fact, in our implementation we perform unfolding on the fly. That is, the CHR solver detects and breaks cycles.

```
    (_:_)   > []      = True
    (x:xs) > (y:ys) = x > y || x == y && xs > ys
instance (Ord Bool)55 where
    True > False = True
    _    > _     = False
```

we generate the following CHRs

$$
\begin{array}{rcl}
Eq\ a & \Longrightarrow & True \\
eq_a(t_{51}) & \Longleftrightarrow & (t_{51} = a \rightarrow a \rightarrow Bool)_{51}, (Eq\ a)_{51} \\
Ord\ a & \Longrightarrow & (Eq\ a)_{52} \\
gt_a(t_{53}) & \Longleftrightarrow & (t_{53} = a \rightarrow a \rightarrow Bool)_{53}, (Ord\ a)_{53} \\
Ord\ [a] & \Longleftrightarrow & (Ord\ a)_{54} \\
Ord\ Bool & \Longleftrightarrow & True
\end{array}
$$

We assume *eq* represents the type of *Eq*'s member function `(==)` and *gt* represents the type of *Ord*'s member function `(>)`. Note that CHRs arising from the type annotations appearing in the classes have a missing implicit class constraint added.

Note we would also generate constraints for the code defining the instance methods for `>` and check this versus the annotation constraints for *gt*. □

The proof of correctness of these rules in modeling the class constraints can be found in [8]. Note that our type debugging approach also immediately extends to more complicated approaches to overloading that can be expressed as CHRs [27].

We generally assume that CHRs are confluent. A set of CHRs is *confluent* if any sequence of derivation steps leads to the same final constraint store. This condition holds trivially for CHRs generated from the Hindley/Milner subset of our language. The same is true for any valid set of Haskell 98 [10] class and instance declarations.

6 Type Inference via CHR Solving

Consider type inference for a function definition `f = e`. We execute the goal $f(t, l)$ using the CHR program P created, i.e. $f(t, l) \longrightarrow_P^* C$, and build ϕ, the most general unifier of C_e (the set of equations in C). Let $\bar{a} = fv(\phi C_u) \setminus fv(\phi l)$ (C_u refers to the set of user-defined constraints in C). These are the variables we will quantify over; we specifically exclude types of λ-bound variables. We can then build the type scheme, $f :: \forall \bar{a}\ \phi C_u \Rightarrow \phi t$.

Note that in our scheme we are a bit more "lazy" in detecting type errors compared to other formulations.

Example 23 Consider

```
e = let f = True True
    in False
```

Our (simplified) translation to CHRs yields

$$
\begin{array}{rcl}
f(t) & \Longleftrightarrow & t_1 = Bool, t_1 = t_2 \rightarrow t_3, t_2 = Bool, t_3 = t \\
e(t) & \Longleftrightarrow & t = Bool
\end{array}
$$

For simplicity, we omit justifications and the l parameter. Note that type inference for expression `e` succeeds, although function `f` is ill-typed. There is no occurrence of `f` in the let body, hence we never execute the CHR belonging to `f`. In a traditional approach, type inference for `e` proceeds by

first inferring the type of `f` immediately detecting that `f` is not well-typed. Note that our approach is still type-safe for a lazy language. Additionally, we could require that all defined functions must be type correct, by simply executing the corresponding CHRs. □

In the following example, we give a precise account of the interplay between CHR solving and justifications attached to constraints.

Example 24 Consider the following program, together with the class and instance declarations of Example 22

```
lteq10 x1 y2 = (not7 ((x4 >3)5 y6)8)9
not ::  Bool -> Bool16
```

where `(>)` is part of the `Ord` class. The translation process yields (again for simplicity we ignore the λ-bound variables argument):

$$\begin{array}{lcl} lteq(t_{10}) & \Longleftrightarrow & (t_1 = t_x)_1, (t_2 = t_y)_2, gt(t_3,)_3, (t_4 = t_x)_4, \\ & & (t_3 = t_4 \rightarrow t_5)_5, (t_6 = t_y)_6, not(t_7)_7, \\ & & (t_5 = t_6 \rightarrow t_8)_8, (t_7 = t_8 \rightarrow t_9)_9, \\ & & (t_{10} = t_1 \rightarrow t_2 \rightarrow t_9)_{10} \\ not(t_{16}) & \Longleftrightarrow & (t_{16} = Bool \rightarrow Bool)_{16} \end{array}$$

These rules are generated directly from the program text.

Type inference for `(lteq`$_{22}$ `[w`$_{17}$`]`$_{18}$`)`$_{23}$ generates the constraints

$$(t_{17} = t_w)_{17}, (t_{18} = [t_{17}])_{18}, lteq(t_{22})_{22}, (t_{22} = t_{18} \rightarrow t_{23})_{23}$$

This is the initial constraint which we run the CHR program on. For the first step, we find $E = \emptyset$ and $D = \{lteq(t_{22})_{22}\}$ means we can apply the first rule above leading to

$$\begin{array}{l} (t_{17} = t_w)_{17}, (t_{18} = [t_{17}])_{18}, t_{10} = t_{22}, (t_1 = t_x)_{\{1,22\}}, \\ (t_2 = t_y)_{\{2,22\}}, gt(t_3)_{\{3,22\}}, (t_4 = t_x)_{\{4,22\}}, (t_3 = t_4 \rightarrow t_5)_{\{5,22\}}, \\ (t_6 = t_y)_{\{6,22\}}, not(t_7)_{\{7,22}, (t_5 = t_6 \rightarrow t_8)_{\{8,22\}}, \\ (t_7 = t_8 \rightarrow t_9)_{\{9,22\}}, (t_{10} = t_1 \rightarrow t_2 \rightarrow t_9)_{\{10,22\}}, \\ (t_{22} = t_{18} \rightarrow t_{23})_{23} \end{array}$$

For brevity, we show the whole derivation in a simplified form, just showing $\theta(C_u) \wedge t_{23} = \theta(t_{23})$ where θ is the mgu of C_e for C at each step, and omit justifications. That is, we only show the user-defined constraints and the top type variable t_{23}, under the effect of the equations in C ignoring justifications.

$$\begin{array}{rl} & lteq([t_w] \rightarrow t_{23}), t_{23} = t_{23} \\ \longrightarrow_{lteq} & not(t_8 \rightarrow t_9), gt([t_w] \rightarrow t_2 \rightarrow t_8), (t_{23} = t_2 \rightarrow t_9) \\ \longrightarrow_{not} & gt([t_w] \rightarrow t_2 \rightarrow Bool), t_{23} = t_2 \rightarrow Bool \\ \longrightarrow_{gt} & Ord\ [t_w], t_{23} = [t_w] \rightarrow Bool \\ \longrightarrow_{Ord\ [a]} & Ord\ t_w, t_{23} = [t_w] \rightarrow Bool \\ \longrightarrow_{Ord\ a} & Ord\ t_w, Eq\ t_w, t_{23} = [t_w] \rightarrow Bool \end{array}$$

In other words the type inferred for the original expression is `(Ord a, Eq a) => [a] -> Bool`. Note that we are more "verbose" than e.g. Hugs [16] which would report `Ord a => [a] -> Bool`. Clearly, the constraint `Eq a` is "redundant", since every instance of `Ord` must be an instance of `Eq` as specified by the class declaration for `Ord`. In [27], we show how to remove such redundant constraints. However, for type debugging purposes it is desirable to keep all constraints for better type explanations.

The fourth step in the derivation is the only one requiring a non-empty set E of equations to justify the match. The constraint $D = \{(Ord\ a_1)_{\{3,22,53\}}\}$ matches the left hand side of the rule $Ord\ [a_2] \Longleftrightarrow (Ord\ a_2)_{54}$. The minimal set of equations $E \subseteq C$ where $\theta = mgu(E)$ is such that $\theta(a_1)$ has the form $[t']$ is

$$\begin{array}{l}(t_1 = t_x)_{\{1,22\}}, (t_{53} = a_1 \rightarrow a_1 \rightarrow Bool)_{\{3,22,53\}}, t_3 = t_{53},\\ (t_4 = t_x)_{\{4,22\}}, (t_3 = t_4 \rightarrow t_5)_{\{5,22\}}, t_{10} = t_{22}, (t_{18} = [t_{17}])_{18},\\ (t_{10} = t_1 \rightarrow t_2 \rightarrow t_9)_{\{10,22\}}, (t_{22} = t_{18} \rightarrow t_{23})_{23}\end{array}$$

The total justifications of $E \cup D$ are $\{1, 3, 4, 5, 10, 18, 22, 23, 53\}$. Hence we replace the constraint $(Ord\ a_1)_{\{3,22,53\}}$ by

$$[a_2] = a_1, (Ord\ a_2)_{\{1,3,4,5,10,18,22,23,53,54\}}$$

□

The following examples shows that some simple constraint reasoning steps allow us to identify all those program locations which actually contribute to the final result.

Example 25 Consider the following partially annotated program.

$\texttt{f}_1\ \texttt{g}_2\ \texttt{x}_3\texttt{= ((g}_4\ \texttt{True}_5\texttt{)}_6\texttt{, (g}_7\ \texttt{x}_8\texttt{)}_9\texttt{, (g}_{10}\ \texttt{True}_{11}\texttt{)}_{12}\texttt{)}$

The (simplified) translation to CHRs yields

$$\begin{array}{lll} f(t_1) & \Longleftrightarrow & (t_2 = t_g)_2, (t_3 = t_x)_3, (t_1 = t_2 \rightarrow t_3 \rightarrow (t_6, t_9, t_{12}))_1,\\ & & (t_4 = t_g)_4, (t_5 = True)_5, (t_4 = t_5 \rightarrow t_6)_6,\\ & & (t_7 = t_g)_7, (t_8 = t_x)_8, (t_7 = t_8 \rightarrow t_9)_9,\\ & & (t_{10} = t_g)_{10}, (t_{11} = True)_{11}, (t_{10} = t_{11} \rightarrow t_{12})_{12}\end{array}$$

We find that

$$\left(\begin{array}{l}(t_2 = t_g)_2, (t_3 = t_x)_3, (t_1 = t_2 \rightarrow t_3 \rightarrow (t_6, t_9, t_{12}))_1,\\ (t_4 = t_g)_4, (t_5 = True)_5, (t_4 = t_5 \rightarrow t_6)_6,\\ (t_7 = t_g)_7, (t_8 = t_x)_8, (t_7 = t_8 \rightarrow t_9)_9\end{array}\right) \supset \bar{\exists}_{t_1} C$$

where C refers to the right-hand side of the above rule. That is, constraints $(t_{10} = t_g)_{10}, (t_{11} = True)_{11}, (t_{10} = t_{11} \rightarrow t_{12})_{12}$ are not necessary to establish the final result. Indeed, we can replace locations $\{10, 11, 12\}$ by `undefined :: a` without changing the final result. □

Another interesting point to note is that our type inference scheme for polymorphic recursive function is more relaxed compared to the one found in some other established type checkers.

Example 26 Consider the following program. For simplicity, we omit justifications.

```
e :: Bool
e = g
f :: Bool -> a
f = g
g = f e
```

In the case of Hugs, the following error reported is:

```
ERROR "mr.hs":8 - Type error in function binding
*** Term           : f
*** Type           : Bool
*** Does not match : Bool -> a
```

The problem reported here stems from the fact that within the mutually recursive binding group consisting of `e, f` and `g`, `f` is assigned two ununifiable types, *Bool* and $Bool \rightarrow a$: the first because it must have the same type as `g`, which according to `e` must be *Bool*; and the second because of its type declaration.

Our translation scheme is more liberal than this, in that `g`'s type within `e` and `f` may be different. Essentially, we only require that the type of a variable be identical at all locations within the mutually recursive subgroup if a type declaration has been provided for that variable.

(Simplified) Translation of the above program to CHRs yields.

$$\begin{array}{lcl} e_a(t) & \Longleftrightarrow & t = Bool \\ e(t) & \Longleftrightarrow & g(t) \\ f_a(t) & \Longleftrightarrow & t = Bool \rightarrow a \\ f(t) & \Longleftrightarrow & g(t) \\ g(t) & \Longleftrightarrow & e_a(t_e), f_a(t_f), t_f = t_e \rightarrow Bool \end{array}$$

It is clear from the above that there are no cycles present amongst these rules. We can use them to successfully infer a type for any of the variables in the program.

In fact, our translation scheme is similar to [18] where type inference of binding groups proceeds as follows:

1. Extend the type environment with the type signatures. In this case `f::forall a. Bool -> a` and `e::Bool`.
2. Do type inference on the bindings without type signatures, in this case `g = f e`. Do generalisation too, and extend the environment, giving `g :: forall a. a`.
3. Now, and only now, do type inference on the bindings with signatures.

□

6.1 Main Results

We can state soundness and completeness of type inference for the Hindley/Milner system as described in Figure 1. We assume that the type of predefined functions is recorded as a CHR. For example, `map :: (a->b)->[a]->[b]` is represented by

$$map(t, l) \Longleftrightarrow t = (a \rightarrow b) \rightarrow ([a] \rightarrow [b]).$$

Theorem 1 (Soundness and Completeness) *Let P_1 be a set of CHRs describing all predefined functions, Γ be an environment containing all free variables, e be an expression and t be a type. Then, we have that $\Gamma \vdash e : t$ iff $\Gamma, e \vdash_{Cons} (C \mid t')$ for some constraint C and type t' and $\Gamma, e \vdash_{Def} P_2$ for some set P_2 of CHRs such that $C \longrightarrow^*_{P_1 \cup P_2} D$ and $\phi t' = t$ with ϕ m.g.u. of D where we consider $fv(\Gamma)$ as Skolem constants.*

Proof: Here is a proof sketch. More details will be added later.

1. The statement is easy to verify for the simply-typed fragment. We generate the correct constraints out of expression.
2. Consider let: W.l.o.g., we assume that all nested lets are lambda-lifted. Almost, direct analogy to our translation. We simply need a slightly stronger induction hypothesis where we compare type schemes.

□

Note that in the above theorem we did not make use of any other CHRs besides those arising through our rule generation algorithm. Soundness holds in general for arbitrary type extensions expressible in terms of CHRs. In general, we need to impose some sufficient conditions on CHRs (e.g. termination and confluence which are trivially satisfied by the CHRs describing the standard Hindley/Milner system) to guarantee completeness. For example, in [27] we have identified a class of CHRs suitable for modeling Haskell-style overloading which enjoy complete type inference. There are certainly further classes of "complete" CHRs. We plan to pursue this topic in future work.

An important property of our inference scheme is that by inspecting justifications attached to minimal implicants in the final store we can identify all "essential" program locations (see Example 25). In the following theorem, we make use of a function $just(D)$ to extract all the justifications attached to constraints in D.

Theorem 2 (Minimal Contributing Locations) *Let Γ be an environment, e be an expression, P be a set of CHRs, C and D be constraints and t be a type such that $\Gamma, e \vdash_{Cons} (C \mid t)$, $\Gamma, e \vdash_{Def} P$ and $C \longrightarrow_P^* D$. For any subset $D' \subseteq C$ where $[\![P]\!] \models D' \supset \bar{\exists}_t D$ we set $J = just(D) \backslash just(D')$. We replace all locations J in e by "undefined". The resulting expression is denoted e'. Then, $\Gamma, e' \vdash_{Cons} (C' \mid t')$ for some constraint C' and type t' such that $[\![P]\!] \models \bar{\exists}_{fv(\Gamma)}\ D \leftrightarrow \bar{\exists}_{fv(\Gamma)}\ D'$ where $C' \longrightarrow_P^* D'$.*

Proof: Here is a proof sketch. More details will be added later.

1. First consider simply-typed fragment (only primitive functions, i.e. local explanation only) straightforward here because
   - we know constraint generation is correct,
   - assumptions ensure that we can leave out all non-relevant constraints (which originate from non-relevant locations which in turn are replaced by undefined which in turn yield (t, True), i.e. does not contribute anything)
2. Extends to let-polymorphism. We simply assume global explanation.

□

The above theorem justifies our debugging methods in case of a type error and unexpected result.

Corollary 1 *Identifying minimal unsatisfiable constraints is sufficient for error explanation.*

Corollary 2 *Identifying minimal implicants is sufficient for type explanation.*

We refer to Section 7 for a description of the two essential constraint operations.

6.2 Subsumption and Unambiguity Checking

Note that in case of type annotations, type inference needs to perform a subsumption check. We compare a type annotation for function f versus the function definition by simply testing the following. We begin with the inferred type $\forall \bar{a}.C \Rightarrow t$, and the annotated type $\forall \bar{a}'.C' \Rightarrow t'$. W.l.o.g., we assume that t and t' are variables. For example, $\forall \bar{a}.C \Rightarrow t$ can always be transformed into the equivalent type scheme $\forall \bar{a}, a.(C, t = a) \Rightarrow a$ where a is a fresh variable. We build constraints $C_1 \equiv C', t' = a, a = a'$, and $C_2 \equiv C_1, C, t = a'$ and run them both in the solver. That is, $C_1 \longrightarrow_P^* D_1$ and $C_1 \longrightarrow_P^* D_1$. Let $V = fv(C', t', \forall \bar{a}.C \Rightarrow t)$. These are the variables which we must ensure are not more general in the declared type than in the inferred. For the check to

succeed, D_1 and D_2 must be equivalent wrt V. That is, $\models (\bar{\exists}_V D_1) \leftrightarrow (\bar{\exists}_V D_2)$. This is the method outlined in [27]. There, we prove correctness of our method and identify some sufficient conditions under which the subsumption check is complete.

This check might fail for two reasons. There is either an unmatched user-defined constraints in D_2 or D_2 contains additional type information about variables V. Note that a third reason might be that constraint D_2 is unsatisfiable. However, such a situation will be handled by our error explanation mechanism (i.e. computing minimal unsatisfiable constraints).

In the first case, we simply use the justifications of the unmatched constraint to highlight the possible source of the subsumption error (see Example 9).

In the second case, let ϕ_1 be m.g.u. of D_1 and ϕ_2 be m.g.u of D_2. We then find a set of equations which represent the additional type information present in D_2, namely $\{a = \phi_2 a \mid a \in V, \models \exists fv(\phi_1 a).\phi_1 a = \phi_2 a\}$. For each such equation, we find a minimal subset of D_2 which implies it. See Section 7.2 for the algorithm which does this.

Alternatively, we could build

$$\{a = \tau_3 \mid a \in V, \models \exists fv(\phi_1 a).\phi_1 a = \phi_2 a, \tau_3 \text{ anti-unifier of } \phi_1 a \text{ and } \phi_2 a\}$$

and continue with the same reasoning steps. We leave a detailed discussion for future work.

Example 27 Consider the following program repeated from 2.3.1, but now with individual locations annotated:

```
f1 ::  a2 -> b3 -> (c5,d6)4
f7 x8 y9 = x10
```

We would generate the following (simplified) CHRs from this program:

$$\begin{array}{rcl} f(t_0) & \Longleftrightarrow & f_a(t_1)_1, f_i(t_7)_7, (t_0 = t_1)_0, (t_0 = t_7)_0 \\ f_a(t_1) & \Longleftrightarrow & (t_1 = a \rightarrow b \rightarrow p)_1,\ (p = (c,d))_4,\ (a = t_2)_2,\ (b = t_3)_3,\ (c = t_5)_5,\ (d = t_6)_6 \\ f_i(t_7) & \Longleftrightarrow & (t_7 = e \rightarrow f \rightarrow g)_7,\ (e = t_8)_8,\ (f = t_9)_9, (g = t_8)_{10} \end{array}$$

Let $f_a(t_1) \longrightarrow C_a$ and $f(t_0) \longrightarrow C_i$.
Consequently, we build the type schemes: $f_a :: \forall fv(C_a).C_a \Rightarrow t_1$, and $f_i :: \forall fv(C_i).C_i \Rightarrow t_0$. We can now perform the subsumption check.

We build constraints, $C_1 = C_a, t_1 = a, a = a'$ and $C_2 = C_1, C_i, t_0 = a'$, and let $\bar{a} = fv(C_a) \cup \{a, a'\}$. Let $C_1 \longrightarrow D_1$ and $C_2 \longrightarrow D_2$, and $\phi_1 = mguD_1$, $\phi_2 = mguD_2$.

We then begin looking for variables in $\bar{a}$ which are more instantiated in the inferred than in the declared constraints. We will eventually find that $\phi_1(t_2) = t_2$, but $\phi_2(t_2) = (t_4, t_5)$. To explain this, we must find the minimal subset of D_2 which implies that $t_2 = (t_4, t_5)$. This set is: $(t_1 = a \rightarrow b \rightarrow p)_1, (a = t_2)_2, (p = (c,d))_4, (c = t_5)_5, (d = t_6)_6, (t_7 = e \rightarrow f \rightarrow g)_7, (e = t_8)_8, (g = t_8)_{10}, (t_0 = t_1)_0, (t_0 = t_7)_0$

We highlight locations: 1,2,4,5,6,7,8,10 as below:

```
f1 ::  a2 -> b3 -> (c5,d6)4
f7 x8 y9 = x10
```

□

In case of overloading, we must also ensure that type schemes are unambiguous. Not enforcing this condition would make the semantics of such programs non-deterministic. For Haskell 98 this equates to checking that variables appearing within the context of the type scheme must also appear within the type. In [27], we have identified a general criteria to test for unambiguity of type schemes. Given a type scheme $\forall \bar{a}.C \Rightarrow t$, we build a renaming ρ of variables $\bar{a}$, run

$C, \rho C, t = \rho t \longrightarrow^* D$, and check if $\models D \rightarrow a = \rho a$, for all a in $\bar{a}$. In other words, we are checking that instantiating variables in the type component also instantiates all variables in the constraints. For details, we refer to [27].

Variables, $\bar{a}$ for which this does not hold, are ambiguous. Let A be the set of all ambiguous variables. We report program sites whose type contains an ambiguous type variable by computing the locations of all variables $v \in fv(D)$ such that $\models D \supset v = t'$ and $\exists a \in fv(t') \wedge a \in \bar{a}$ for some type t'.

Example 28 Consider the following program, which has the ambiguous type $(Show\ a, Read\ a) \Rightarrow String \rightarrow b$:

```
p u1 = (p3 (show5 (read7 u8)6)4)2
```

The whole process is as follows. First we generate the following (simplified) rule from the program above. The rules for `read` and `show` (defined elsewhere) are also shown below.

$$\begin{array}{rcl} p(t) & \Longleftrightarrow & show(t_5), read(t_7), t_3 = t, t_3 = t_4 \rightarrow t_2, t_5 = t_6 \rightarrow t_4, t_7 = t_8 \rightarrow t_6, t_8 = t_1, t = t_1 \rightarrow t_2 \\ read(t) & \Longleftrightarrow & t = String \rightarrow a, Read\ a \\ show(t) & \Longleftrightarrow & t = a \rightarrow String, Show\ a \end{array}$$

Inferring `p`'s type, we perform the derivation $p(t) \longrightarrow^* C$ where

$$\begin{array}{ll} C = \{ & t_7 = String \rightarrow a', Read\ a', t_5 = a \rightarrow String, Show\ a, t_3 = t, t_3 = t_4 \rightarrow t_2, t_5 = t_6 \rightarrow t_4, \\ & t_7 = t_8 \rightarrow t_6, t_8 = t_1, t = t_1 \rightarrow t_2 \} \end{array}$$

In this (simplified) case we build the type scheme $\forall \bar{a}.C \Rightarrow t$, where $\bar{a} = fv(C)$. We now run our unambiguity checking algorithm to determine which of the quantified type variables are ambiguous and appear in an ambiguous constraint. In our example, we find that variables a and a' are ambiguous. The following locations have types which include either of these variables: 5 ($t_5 = a \rightarrow String$), 6 ($t_6 = a$) and 7 ($t_7 = String \rightarrow a$). The debugger would display:

```
p u1 = (p3 (show5 (read7 u8)6)4)2
```

This indicates that we could fix this program by providing more type information at locations 5, 6 or 7. The following is one possibility.

```
p u = p ((show::Int->String) (read u))
```

□

7 Constraint Operations

The type debugger make use of two essential manipulations of the constraints generated from the CHR derivation: finding a minimal unsatisfiable subset of an unsatisfiable constraint set, and finding a minimal subset that implies some give constraint (which may be used if the constraints are satisfiable or unsatisfiable). Based on Theorem 2, justifications attached to those minimal sets refer to problematic program locations.

7.1 Minimal Unsatisfiable Subsets

Assume type inference fails. That is, we have that $C \longrightarrow_P^* D$ for some constraint C and D where D is unsatisfiable. For D to be unsatisfiable it must be that D_e is unsatisfiable, since user-defined constraints only contribute new equations.

We are interested in finding a minimal subset E of D_e such that E is unsatisfiable. An unsatisfiable set is *minimal* if the removal of any constraint from that set leaves it satisfiable. The Chameleon system simply finds an arbitrary minimal unsatisfiable subset. We also determine which constraints in this set are present in *all* minimal unsatisfiable subsets.

We can naively determine minimal unsatisfiable subsets by testing each possible subset. This is impractical. Using an incremental equation solver (as all unification algorithms are) we can quickly determine a minimal unsatisfiable subset of D by adding the equations one at a time and detecting the first time the set is unsatisfiable. The last added equation must be involved in the minimal unsatisfiable subset. Applying this principle repeatedly results in:

```
min_unsat(D)
  M := ∅
  while satisfiable(M) {
    C := M
    while satisfiable(C)
      { let e ∈ D − C; C := C ∪ {e} }
    D := C; M := M ∪ {e} }
  return M
```

We can straightforwardly determine which constraints $e \in M$ must occur in all minimal unsatisfiable subsets, since this is exactly those where $D - \{e\}$ is satisfiable. The complexity (for both checks) is $O(|D|^2)$ using an incremental unification algorithm. A detailed analysis of the problem of finding all minimal unsatisfiable constraints can be found in [7].

Ultimately, we are interested in the justifications attached to minimal unsatisfiable constraints. This will allow us to identify problematic locations in the program text.

Example 29 Consider the final constraint of Example 1.

$$(t_1 = Char)_{\{1,8\}}, t_2 = t_7, (t_5 = Bool)_{\{5,2,8\}}, (t_6 = Bool)_{\{6,2,8\}},$$
$$(t_7 = t_5 \rightarrow t_6)_{\{7,2,8\}}, (t_2 = t_1 \rightarrow t_3)_{\{3,8\}}, (t_4 = t_3)_{\{4,8\}}$$

The system of constraints is detected as unsatisfiable as the second last constraint $(t_2 = t_1 \rightarrow t_3)_{\{3,8\}}$ is added. Hence $(t_4 = t_3)_{\{4,8\}}$ can be excluded from consideration. Solving from the beginning, starting with $(t_2 = t_1 \rightarrow t_3)_{\{3,8\}}$, unsatisfiability is detected at $(t_7 = t_5 \rightarrow t_6)_{\{7,2,8\}}$. In the next iteration, starting with $(t_7 = t_5 \rightarrow t_6)_{\{7,2,8\}}$ and $(t_2 = t_1 \rightarrow t_3)_{\{3,8\}}$, unsatisfiability is detected at $(t_5 = Bool)_{\{5,2,8\}}$. Therefore, $(t_6 = Bool)_{\{6,2,8\}}$ can be excluded. The final result M is

$$(t_1 = Char)_{\{1,8\}}, t_2 = t_7, (t_5 = Bool)_{\{5,2,8\}},$$
$$(t_7 = t_5 \rightarrow t_6)_{\{7,2,8\}}, (t_2 = t_1 \rightarrow t_3)_{\{3,8\}}$$

Note that M is the only minimal unsatisfiable constraint for this example. □

7.2 Minimal Implicants

We are also interested in finding minimal systems of constraints that ensure that a type has a certain shape.

Assume that $C \longrightarrow_P^* D$ where $\models D \supset \exists \bar{a}.F$ unexpectedly, where F is a conjunction of equations. We want to identify a minimal subset E of D such that $\models E \supset \exists \bar{a}.E$. The algorithm for finding minimal implicants is highly related to that for minimal unsatisfiable subsets.

The code for min_impl is identical to min_unsat except the test $satisfiable(S)$ is replaced by $\neg implies(S, \exists \bar{a}.D')$.

```
min_impl(D)
   M := ∅
   while ¬implies(M, ∃ā.D′) {
      C := M
      while ¬implies(C, ∃ā.D′)
         { let e ∈ D − C; C := C ∪ {e} }
      D := C; M := M ∪ {e} }
   return M
```

The test $implies(M, \exists\bar{a}.D')$ can be performed as follows. If D' is a system of equations only, we first add M_e to an incremental equation solver, and then add D them and check that no variable apart from those in $\bar{a}$ is further bound from the state with M.

If D' includes user defined constraints, then for each user-defined constraint $c_i \in D'_u$ we nondeterministically choose a user-defined constraint $c'_i \in M$. We then check that $implies(M, \exists\bar{a}.(D'_e \cup \{c_i = c'_i\})$ holds as above. We need to check all possible choices for c'_i (although we can omit those which obviously lead to failure, e.g. $c_i = Eq\ a$ and $c'_i = Ord\ b$).

8 Related Work

The most conservative approach to improving type error information involves modifying the order in which substitutions take place within traditional inference algorithms. The standard algorithm, $\mathcal{W}$, tends to find errors too late in its traversal of a program [20, 33], since it delays substitutions until as late as possible. $\mathcal{W}$ has been generalized [20] so that the point at which substitutions are applied can be varied. Despite this, there are cases where it is not clear which variation provides the most appropriate error report. Moreover, all of these algorithms suffer from a left-to-right bias when discovering errors during abstract syntax tree (AST) traversal.

One way to overcome this problem, as we have seen, is to avoid the standard inference algorithms altogether and focus directly on the constraints involved. Although our work bears a strong resemblance to [11, 12, 13], our aims are different. We attempt to explain errors involving advanced type system features, such as overloading, whereas [13], who are developing a beginner-friendly version of Haskell, choose to ignore such features by design. Furthermore, they focus on producing non-interactive error messages, and do not consider mechanisms for providing type explanations.

In [22], graphs are used to represent type information, again, independently of any particular program traversal. This work allows generation of potentially more useful type error messages, again without any opportunity for user interaction.

A number of "error explanation systems" [2, 5, 32] allow the user to examine the process by which specific types are inferred. By essentially recording the effects of the inference procedure on specific types a step at a time, a complete history can be built up. One common shortcoming of such systems is the excessive size of explanations. Although complete, such explanations are full of repetitive and redundant information which can be a burden to deal with. Furthermore, since these systems are layered on top of an existing inference algorithm, they suffer from the same AST traversal bias. In contrast, when asked to explain why an expression has a particular type, our system finds precisely those locations which have contributed.

Chitil [3] describes a compositional type explanation system based on the idea of principal typings [17]. In his system a user can explore the types of subexpressions by manually navigating through the inference tree. This is very similar to our form of declarative debugging (Section 2.7). Note that our form of type explanation allows us to automatically identify contributing program locations.

Independently, Haack and Wells [9] also discuss finding of minimal unsatisfiable subsets which allows them to find problematic program locations. However, they only consider error explanations. That is, in their system it is not possible to explain why functions have a type of a certain shape.

Furthermore, their approach applies to the Hindley/Milner system only whereas our approach is applicable to Haskell-style type classes and its various extensions.

9 Conclusion

We have presented a flexible type debugging scheme for Hindley/Milner typable programs which also includes Haskell-style overloading. The central idea of our approach is to translate the typing problem to a constraint problem, i.e. a set of CHRs. We have demonstrated that CHRs is a sufficiently rich constraint language to encode the typing problem for a wide range of Hindley/Milner style system. Type inference is phrased in terms of CHR solving. Our approach has the advantage that we are not dependent on a fixed traversal of the abstract syntax tree. Constraints can be processed in arbitrary order which makes a flexible traversal of the syntax tree possible.

In case of a type error (or unexpected result), we find minimal unsatisfiable constraints (minimal implicants). Justifications, i.e. program locations, attached to constraints allow us to identify problematic program expressions. The approach has been fully implemented [30].

There is much further work to do in improving the system. This includes adding features such as: allowing the user to trace the CHR type inference derivation, and explaining each step in the derivation, and using the minimal unsatisfiable subsets to generate better error messages. In particular, we plan to include some heuristics to catch common errors. The Helium [13] programming environment includes a database of common mistakes which is searched for a match when a type error occurs. This allows meaningful error messages and suggestions on how to fix the error to be presented. Using minimal unsatisfiable subsets to search in the database should allow us to detect more generic common mistakes.

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] 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 *Proc. of ICFP'01*, pages 193–204. ACM Press, 2001.

[4] B. Demoen, M. García de la Banda, and P. J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proc. of the 22nd Australian Computer Science Conference*, pages 217–228. Springer-Verlag, 1999.

[5] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

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

[7] M. García de la Banda, P.J. Stuckey, and J. Wazny. Finding all minimal unsatisfiable constraints. In *Proc. of PPDP'03*. ACM Press, 2003. To appear.

[8] K. Glynn, P. J. Stuckey, and M. Sulzmann. Type classes and constraint handling rules. In *Workshop on Rule-Based Constraint Reasoning and Programming*, 2000. http://xxx.lanl.gov/abs/cs.PL/0006034.

[9] 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.

[10] Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.

[11] B. Heeren and J. Hage. Parametric type inferencing for Helium. Technical Report UU-CS-2002-035, Utrecht University, 2002.

[12] B. Heeren, J. Hage, and D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Utrecht University, 2002.

[13] Helium home page. http://www.cs.uu.nl/ afie/helium/.

[14] Fritz Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.

[15] 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.

[16] Hugs home page. haskell.org/hugs/.

[17] . Jim. What are principal typings and what are they good for? In ACM Press, editor, *Proc. of POPL'96*, pages 42–53, 1996.

[18] M. Jones. Typing haskell in haskell. In *Haskell Workshop*, September 1999.

[19] M. P. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, Department of Computer Science, September 1993.

[20] 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.

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

[22] B.J. McAdam. Graphs for recording type information. Technical Report ECS-LFCS-99-415, The University of Edinburgh, 1999.

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

[24] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[25] M. Odersky, M. Sulzmann, and M Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[26] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

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

[28] P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *Proc. of Haskell Workshop'03*. ACM Press, 2003. To appear.

[29] M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

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

[31] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76. ACM Press, 1989.

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

[33] 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.