# Type Processing by Constraint Reasoning

Peter J. Stuckey[1,2], Martin Sulzmann[3], and Jeremy Wazny[2]

[1] NICTA Victoria Laboratory
[2] Department of Computer Science and Software Engineering
University of Melbourne, 3010 Australia. {pjs,jeremyrw}@cs.mu.oz.au
[3] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543 sulzmann@comp.nus.edu.sg

**Abstract.** Herbrand constraint solving or unification has long been understood as an efficient mechanism for type checking and inference for programs using Hindley/Milner types. If we step back from the particular solving mechanisms used for Hindley/Milner types, and understand type operations in terms of constraints we not only give a basis for handling Hindley/Milner extensions, but also gain insight into type reasoning even on pure Hindley/Milner types, particularly for type errors. In this paper we consider typing problems as constraint problems and show which constraint algorithms are required to support various typing questions. We use a light weight constraint reasoning formalism, Constraint Handling Rules, to generate suitable algorithms for many popular extensions to Hindley/Milner types. The algorithms we discuss are all implemented as part of the freely available Chameleon system.

## 1 Introduction

Hindley/Milner type checking and inference has long been understood as a process of solving Herbrand constraints, but typically the typing problem is not first mapped to a constraint problem and solved, instead a fixed algorithm, such as algorithm $\mathcal{W}$ using unification, is used to infer and check types. We argue that understanding a typing problem by first mapping it to a constraint problem gives us greater insight into the typing in the first place, in particular:

- Type inference corresponds to collecting the type constraints arising from an expression. An expression has no type if the resulting constraints are *unsatisfiable*.
- Type checking corresponds to checking that the declared type, considered as constraints, *implies* (that is has more information than) the inferred type (constraints collected from the definition).
- Type errors of various classes: ambiguity, subsumption errors; can all be explained better by reasoning on the type constraints.

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" [22], with respect to a large number of potential problems.

Typically type processing of a program either checks that types declared for each program construct are correct, or, better, infers the types for each program construct and checks that these inferred types are compatible with any declared types. If the checks succeed, the program is type correct and cannot "go wrong".

However, programs are often not well-typed, and therefore must be modified before they can be accepted. Another important role of the type processor is to help the author determine why a program has been rejected, what changes need to be made to the program for it to be type correct.

Traditional type 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 his program unaided. This can be a daunting task for even experienced programmers; beginners are often left bewildered.

Our thesis is that by mapping the entire typing problem to a set of constraints, we can use constraint reasoning to (a) concisely and efficiently implement the type processor and (b) accurately determine where errors may occur, and aid the programmer in correcting them. The Chameleon [32] system implements this for rich Hindley/Milner based type languages.

We demonstrate our approach via three examples. Note that throughout the paper we will adopt Haskell [11] style syntax in examples.

*Example 1.* Consider the following ill-typed program:

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

Here `error` is the standard Haskell function with type $\forall a.[Char] \rightarrow a$. GHC reports:

```
mdef.hs:4:
    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:[1]

[1] The currently available Chameleon system (July 2005) no longer supports these more detailed error messages, after extensions to other parts of the system. The feature will be re-enabled in the future. The results are given from an earlier version.

```
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 True = error "'a'"
          f c True z = error "'b'"
          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.

If we assume the actual error is that the `True` in the second definition should be a `'b'` through some copy-and-paste error, then it is clear that the GHC error message provides little help in discovering it. The Chameleon error certainly implicates the `True` in the problem and gives type information that should direct the programmer to the problem quickly.

As part of the diagnosis the system "colours" both the conflicting types and certain program locations. A program location which contributes to any of the reported conflicting types is highlighted in the same style as that type. Locations which contribute to multiple reported types are highlighted in a combination of the styles of the types they contribute to. (There are no such locations in the case above.)

The above example illustrates the fundamental problems with any traditional Hindley/Milner type inference like algorithms $\mathcal{W}$ [22]. The algorithms suffer from a bias derived from the way they traverse the abstract syntax tree (AST). The second problem is that being tied to unification, which is only one particular implementation of a constraint solving algorithm for tree constraints, they do not treat the problem solely as a constraint satisfaction problem.

The problems of explaining type errors are exacerbated when the type system becomes more complex. Type classes [34] are an important extension to Hindley/Milner types, allowing principled (non-parametric) overloading. But the extension introduces new classes of errors and complicates typing questions. Type classes are predicates over types, and now we have to admit that type processing is a form of reasoning over first order formulae about types.

*Example 2.* Consider the following program which is typical of the sort of mistake that beginners make. The base case `sum [] = []` should read `sum [] = 0`. The complexity of the reported error is compounded by Haskell's overloading of numbers.

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

sumLists = sum . map sum
```

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:4:
No instance for (Num [a]) arising from use of ‘sum’ at sum.hs:3
Possible cause: the monomorphism restriction applied to the following:
  sumLists :: [[[a]]] -> [a] (bound at sum.hs:3)
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)
```

The error message is completely misleading, except for the fact that the problem is there is no instance of Num [a]. The probable fix will not help.

For this program Chameleon reports the following:

```
sum.hs:4: ERROR: Missing instance
Instance:Num [a]: sum [] = []
                  sum (x:xs) = x + sum xs
```

This indicates that the demand for this instance arises from the interaction between [] on the first line of sum and (+) on the second. The actual source of the error is highlighted.

The advantages of using constraint reasoning extend as the type system becomes even more complex. Generalized Algebraic Data Types (GADTs) [3, 36] are one of the latest extensions of the concept of algebraic data types. They have attracted a lot of attention recently [24–26]. The novelty of GADTs is that the (result) types of constructor may differ. Thus, we may make use of additional type equality assumptions while typing the body of a pattern clause.

*Example 3.* Consider the following example of a GADT, using GHC style notation, where List a n represents a list of as of length n. Type constructors Z and S are used to represent numbers on the level of types.

```
data Z -- zero
data S n -- successor
data List a n where
  Nil :: List a Z
  Cons :: a -> List a m -> List a (S m)
```

We can now express much more complex behaviour of our functions, for example

```
map :: (a -> b) -> List a n -> List b n
map f Nil = Nil
map f (Cons a l) = Cons (f a) (map f l)
```

which guarantees that the map function returns a list of the same length as its input.

GADTs introduce more complicated typing problems because different bodies of the same function can have different types, since they act under different assumptions. This makes the job of reporting type errors much more difficult.

*Example 4.* Consider defining another GADT to encode addition among our (type) number representation.

```
data Sum l m n where
  Base :: Sum Z n n
  Step :: Sum l m n -> Sum (S l) m (S n)
```

We make use of the `Sum` type class to refine the type of the `append` function. Thus, we can state the desired property that the length of the output list equals the sum of the length of the two input lists.

```
append2 :: Sum l m n -> List a l -> List a m -> List a n
append2 Base Nil ys = Nil -- wrong!! should be ys
append2 (Step p) (Cons x xs) ys = Cons x (append p xs ys)
```

For this program GHC reports

```
append.hs:17:22:
    Couldn't match the rigid variable 'n' against 'Z'
      'n' is bound by the type signature for 'append2'
      Expected type: List a n
      Inferred type: List a Z
    In the definition of 'append2': append2 Base Nil ys = Nil
```

For this program Chameleon currently reports:

```
ERROR: Polymorphic type variable 'n' (from line 13, col. 56) instantiated by
append2 :: Sum l m n -> List a l -> List a m -> List a n
append2 Base Nil ys = Nil -- wrong!! should be ys
```

Here we can determine the actual locations that cause the subsumption error to occur. We could also give information on the assumptions made, though presently Chameleon does not. We aim in the future to produce something like:

```
append.hs:10: ERROR:  Inferred type does not subsume declared type
Problem: The variable 'm' makes the declared type too polymorphic
       Under the assumptions l = Z and m = n arising from
            append2 Base Nil ys = Nil
       Declared: Sum Z m m -> List a Z -> List a m -> List a m
       Inferred: Sum Z m m -> List a Z -> List a m -> List a Z
append2 Base Nil ys = Nil -- wrong!! should be ys
```

Our advantage is that we use a constraint-based system where we maintain information which constraints arise from which program parts. GHC effectively performs unification under a mixed prefix, hence, GHC only knows which 'branch' failed but not exactly where.

As the examples illustrate, by translating type information to constraints with locations attached we can use constraint reasoning on the remaining constraint problem. The constraint reasoning maintains which locations caused any inferences it makes, and we can then use these locations to help report error messages much more precisely. In this paper we show how to translate complex typing problems to constraints and reason about the resulting typing problems.

The rest of the paper is organized as follows. In Section 2 we introduce our language for constraints, and the CHR formalism for constraint reasoning. We show how constraint algorithms for satisfiability and inference are expressible using CHRs. In Section 3 we show how we map a functional program to a CHR program defining the type constraints. We then in Section 4 examine typing in Hindley/Milner using our system, before considering reporting errors in Hindley/Milner in Section 5. We add type classes in Section 6, and show how that changes type inference and checking, and introduces new kinds of type errors. We then briefly consider further extensions such as functional dependencies, programmed type extensions and GADTs in Section 7. We conclude with a brief discussion of related work. Much of the technical underpinnings to material in this paper has appeared previously, and so we leave the presentation as quite informal. For more details the reader is referred to [6, 28–31, 35].

## 2 Constraints and CHRs

In this section we introduce constraints with location annotations, and our framework for constraint reasoning, Constraint Handling Rules.

We use notation $\bar{o}$ to refer to a sequence of objects $o$, usually types or variables. Out type language is standard, we assume type variables $a$, function types $t \to t$ and user definable data types $T\ \bar{t}$. We use common Haskell notation for writing function, pair, list types, etc.

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.

Conjunctions of constraints are sometimes written using a comma separator instead of the Boolean connective $\wedge$. 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. see e.g. [20]. We define $mgu(C)$ to return a most general unifier of the equations $C$.

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 $[1, 2, 3]$.

In addition to the Boolean operator $\wedge$ (conjunction), we make use of $\supset$ (implication) and $\leftrightarrow$ (equivalence) and quantifiers $\exists$ (existential) and $\forall$ (universal) to express conditions in formal statements, typing rules etc. We assume that *fv*$(o)$ computes the free variables not bound by any quantifier in an object $o$. We write $\bar{\exists}_o.F$ as a short-hand for $\exists fv(F) - fvo.F$ where $F$ is a first-order formula and $o$ is an object. Unless otherwise stated, we assume that formulae are implicitly universally quantified. We refer to [27] for more details on first-order logic.

**Constraint Handling Rules with Justifications** We will translate typing problems to a constraint problem where the meaning of the user-defined constraints is defined by Constraint Handling Rules (CHRs) [8]. CHRs manipulate a global set of primitive constraints, using rewrite rules of two forms

$$\begin{array}{rl}\text{simplification } (r1) & c_1, \ldots, c_n \iff d_1, \ldots, d_m \\ \text{propagation } (r2) & c_1, \ldots, c_n \implies d_1, \ldots, d_m\end{array}$$

where $c_1, \ldots, c_n$ are user-defined constraints, $d_1, \ldots, d_m$ are constraints, and *r1* and *r2* are labels by which we can refer to these rules. We will often omit rule labels when they are not necessary. A CHR *program* $P$ is a set of CHRs.

In our use of the rules, constraints occurring on the right hand side of rules have justifications attached. We extend the usual derivation steps of Constraint Handling Rules to maintain and extend these justifications.

A *simplification derivation step* applying a (renamed apart) rule instance $r \equiv c_1, \ldots, c_n \iff d_1, \ldots, d_m$ to a set of constraints $C$ is defined as follows. Let $E \subseteq C_e$ where $\theta = mgu(E)$. Let $D = \{c'_1, \ldots, c'_n\} \subseteq C_u$, and suppose there exists substitution $\sigma$ on variables in $r$ such that $\{\theta(c'_1), \ldots, \theta(c'_n)\} = \{\sigma(c_1), \ldots, \sigma(c_n)\}$, i.e. 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 $E \cup D$. Note that there may be multiple subsets of $C_e$ which satisfy the above condition and allow matching to occur. For our purposes, however, we require the subset $E$ to be *minimal*. i.e. no strict subset of $E$ can allow for a match. An algorithm for finding such an $E$ is detailed later in this section.

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 *beginning* of the original justification 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 right hand side only.

A *propagation derivation step* applying a (renamed apart) rule instance $r \equiv c_1, \ldots, c_n \implies d_1, \ldots, d_m$ 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. Abdennadher[1].

*Example 5.* Consider the following CHRs.

$$\begin{array}{l} g(t_4) \iff (t_1 = Char)_1, f(t_2)_2, (t_2 = t_1 \to t_3)_3, (t_4 = t_3)_4 \\ f(t_7) \iff (t_5 = Bool)_5, (t_6 = Bool)_6, (t_7 = t_5 \to t_6)_7 \end{array}$$

A CHR derivation from the goal $\mathsf{g}_8$ where 8 stands for a hypothetical program location, is shown below. To help the reader, we underline constraints involved in rule applications.

$$\begin{array}{l} \quad\ \underline{g(t)_8} \\ \longrightarrow t = t_4, (t_1 = Char)_{[8,1]}, \underline{f(t_2)_{[8,2]}}, (t_2 = t_1 \to t_3)_{[8,3]}, (t_4 = t_3)_{[8,4]} \\ \longrightarrow t = t_4, (t_1 = Char)_{[8,1]}, t_2 = t_7, (t_5 = Bool)_{[8,2,5]}, (t_6 = Bool)_{[8,2,6]}, \\ \quad\ (t_7 = t_5 \to t_6)_{[8,2,7]}, (t_2 = t_1 \to t_3)_{[8,3]}, (t_4 = t_3)_{[8,4]} \end{array}$$

Note that we have not bothered to rename any of the new constraints, since all the variables are already distinct, and no rule is applied more than once. In the first step, the constraint $g(t)_8$ matches the left hand side of the first CHR. We replace $g(t)_8$ by the right hand side. In addition, we add the matching equation $t = t_4$. Note how the justification from $g(t)_8$ is added to each justification set. Thus, by propagating justifications we retain the connection constraints and the program locations from which these constraints were originating from. In the final step, the constraint $f(t_2)_{[8,2]}$ matches the left hand side of the second CHR. Hence, we add $[8, 2]$ to the constraints on the right hand side of $\mathsf{g}$'s CHR.

Because of the highly nondeterministic operational semantics, an important property of a CHR program is *confluence*, which demands that each possible order of rule applications leads to the same results (modulo renaming). That is, if $C \longrightarrow C'$ and $C \longrightarrow C''$, then $C' \longrightarrow^*_P D$ and $C'' \longrightarrow^*_P D'$ where $\bar{\exists}_C D \leftrightarrow \bar{\exists}_C D'$. We will demand that the CHR programs we use are confluent. Another important property is termination. A set $P$ of CHRs is *terminating* iff for each $C$ we find $D$ such that $C \longrightarrow^*_P D$. Again we will demand that the CHR programs we use are terminating.

A common restriction on a CHRs $C \iff D$ or $C \implies D$ is *range restriction*, that is, $fv(\phi(D)) \subseteq fv(\phi(C)$ where $\phi = mgu(D_e)$. Usually it holds because $fv(D) \subseteq fv(C)$. Range restrictedness essentially prevents new variables from being introduced by rules. We will also restrict attention to CHR programs where simplification rules are *single-headed*, that is, of the form $c \iff d_1, \ldots, d_m$.

Given a CHR program $P$ which is confluent, terminating, range-restricted and only includes single-headed simplification rules, we can define a number of constraint operations.

**Satisfiability** We use an open world assumption for satisfiability of CHR constraints, that is, we assume we can always add a new rule making a new fixed type constraint hold. In that case unsatisfiability can only result from the equations. We can check that $C$ is satisfiable by determining $C \longrightarrow^*_P D$ and checking that $D_e$ is satisfiable.

```
min_unsat($D$)
  $M$ := $\emptyset$
  while $satisfiable(M)$ {
    $C$ := $M$
    while $satisfiable(C)$
      { let $e \in D - C$; $C$ := $C \cup \{e\}$ }
    $D$ := $C$; $M$ := $M \cup \{e\}$ }
  return $M$
```
(a)

```
min_impl($D$, $\exists\bar{a}.F$)
  $M$ := $\emptyset$
  while $\neg implies(M, \exists\bar{a}.F)$ {
    $C$ := $M$
    while $\neg implies(C, \exists\bar{a}.F)$
      { let $e \in D - C$; $C$ := $C \cup \{e\}$ }
    $D$ := $C$; $M$ := $M \cup \{e\}$ }
  return $M$
```
(b)

**Fig. 1.** Constraint manipulation algorithms.

**Minimal Unsatisfiable Subsets** Given an unsatisfiable constraint $D$, we will be 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. An algorithm is shown in Figure 1(a)

*Example 6.* Consider the final constraint of Example 5. It is unsatisfiable, applying min_unsat to this constraint yields.

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

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

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 [9].

**Implication Testing** Given the restrictions on CHR programs defined above, we can show that they provide a canonical normal form (see [28] for details), that is, every equivalent constraint is mapped to an equivalent (modulo renaming) result. We can use an equivalence check to determine implication of $\bar{\exists}_V C \supset \bar{\exists}_V C'$, where we assume $C$ and $C'$ are renamed apart except for $V$, as follows. We execute $C \longrightarrow_P^* D$ and $C, C' \longrightarrow_P^* D'$, then check that $\phi(D_u)$ is a renaming of $\phi'(D'_u)$, where $\phi = mgu(D_e)$ and $\phi' = mgu(D'_e)$.

**Minimal Implicants** We are also interested in finding minimal systems of constraints that imply another constraint. Assume that $C \longrightarrow_P^* D$ where $\models D \supset \exists\bar{a}.F$. We want to identify a minimal subset $E$ of $D$ such that $\models E \supset \exists\bar{a}.F$. 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 $\neg implies(S, \exists\bar{a}.F)$. It is shown in Figure 1(b)

The test $implies(M, \exists\bar{a}.F)$ can be performed as follows. If $F$ is a system of equations only, we build $\phi = mgu(M_e)$ and $\phi' = mgu(M_e \wedge F)$ and check

| | | |
|---|---|---|
| Expressions | $e$ | $::= f_l \mid x_l \mid (\lambda x_l.e)_l \mid (e\ e)_l \mid (\mathsf{case}\ e\ \mathsf{of}\ ([(p_i \rightarrow e_i)_l]_{i \in I})_l)_l$ |
| Patterns | $p$ | $::= x_l \mid (K\ p...p)_l$ |
| Types | $t$ | $::= a \mid t \rightarrow t \mid T\ \bar{t}$ |
| Primitive Constraints | $at$ | $::= t = t \mid TC\ \bar{t}$ |
| Constraints | $C$ | $::= at \mid C \wedge C$ |
| Type Schemes | $\sigma$ | $::= t \mid \forall \bar{a}.C \Rightarrow t$ |
| Fun Decls | $fd$ | $::= f :: (C \Rightarrow t)_l \mid f_l = e$ |
| Data Decls | $dd$ | $::= \mathtt{data}\ T\ \bar{a} = K\ \bar{t}$ |
| Type Class Decls | $tc$ | $::= \mathsf{class}\ (C \Rightarrow TC\ \bar{a})_l\ \mathsf{where}\ m :: (C \Rightarrow t)_l \mid \mathsf{instance}\ C \Rightarrow (TC\ \bar{t})_l$ |
| Programs | $FP$ | $::= \epsilon \mid fd\ FP \mid dd\ FP \mid tc\ FP$ |

**Fig. 2.** Syntax of Programs

that $\phi(a) = \phi'(a)$ for all variables except those in $\bar{a}$. If $F$ includes user defined constraints, then for each user-defined constraint $c_i \in F_u$ we nondeterministically choose a user-defined constraint $c'_i \in M$. We then check that $implies(M, \exists \bar{a}.(F_e \wedge c_i = c'_i)$ 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 $c'_i = Ord\ b$).

## 3 Type Processing

Our approach to type processing follows [5] by translating the typing problem into a constraint problem and inferring and checking types by constraint operations. We map the type information to a set of Constraint Handling Rules (CHRs) [8], where the constraints are justified by program locations.

### 3.1 Expressions, Types and Constraints

The syntax of programs can be found in Figure 2. Using case expressions we can easily encode multiple-clause definitions: and if-then-else expressions which we will make use of in our examples. For brevity we omit nested function definitions and recursive functions. They are straightforward to handle, but messy; for a complete treatment see [35].

Note that our expressions are fully *labeled*, i.e. we label program locations with unique numbers. We indicate these labels by a subscript $l$ following the expression, as can be seen in the language description above. Labels will become important when generating constraints from a source program.

We assume that $K$ refers to constructors of user-defined data types. As usual patterns are assumed to be linear, i.e., each variable occurs at most once. In examples we will use pattern matching notation for convenience. Note that each pattern/action has a location, as well as a location for the list of all pattern/actions and a location for the case.

We assume data type declarations $\mathtt{data}\ T\ \bar{a} = K\ t_1\ \cdots\ t_n$ are preprocessed and the types of constructors $K : \forall \bar{a}.t_1 \rightarrow \cdots \rightarrow t_n \rightarrow T\ \bar{a}$ are recorded in the environment $E$.

Type schemes have an additional constraint component which allows us to restrict the set of type instances. We often refer to a type scheme as a type for short. Note that we consider $\forall \bar{a}.t$ as a short-hand for $\forall \bar{a}.True \Rightarrow t$.

We also ignore the bodies of instance declarations for brevity, they don't add any significant extra complication.

### 3.2 Constraint Generation from Expressions

The basic idea of our translation is that we map a functional program $FP$ to a CHR program $P$. For each function $f$ defined by the program $FP$ we introduce a unary predicate $f$ in $P$ such that the solutions of $f(t)$ are the types of $f$.

Constraint generation is formulated as a logical deduction system with clauses of the form $E, \Gamma, e \vdash_{Cons} (F \mid t)$ where the environment $E$ of all pre-defined functions, environment $\Gamma$ of lambda-bound variables, and expression $e$ are input parameters and constraint $C$ and type $t$ are output parameters.

Each individual sub-expression gives rise to a constraint which is justified by the location attached to this sub-expression. See Figure 3 for details. In rule (Var-x) we simply look up the type of a $\lambda$-bound variable in $\Gamma$. The rule (Var-x) creates a renamed apart copy of the type of a predefined let-bound function. In rule (Var-f) we generate an "instantiation" constraint, to represent the type of a let-defined function, we use the notation $[\overline{b/a}]$ to define a substitution replacing each $a \in \bar{a}$ by the corresponding $b \in \bar{b}$. In rule (Case) we first equate the types of all pattern/actions, and then treat the remainder like an application. In rule (Pat) we make use of auxiliary judgments of the form $p \vdash_{Cons} \forall \bar{b}.(C \mid t \mid \Gamma_p)$ which we use to generate types and constraints from patterns, as well as to extend the type environment with newly bound variables. The other rules are straightforward.

## 4 Hindley/Milner Types

We begin by restricting ourselves to programs without type classes or instances. This leaves us in the case of pure Hindley/Milner types. Generation of CHRs is straightforward by iteration over the program. The function definition $f = e$ generates the rule $f(t) \iff C$ where $E, \emptyset, e \vdash_{Cons} (C, t)$. This defines the predicate $f$ encoding the type of function $f$. The function declaration $f : (C \Rightarrow t)_l$ generates the rule $f_a(t') \iff C_l \wedge (t = t')_l$. This defines the predicate $f_a$ encoding the annotated type of $f$.

*Example 7.* For example the (location annotated) program

```
(g = (f₂ 'a'₁)₃)₄
(f True₅ = True₆)₇
```

is translated to (after some simplification[2]):

$$\begin{aligned} g(t_4) &\iff (t_1 = Char)_1, f(t_2)_2, (t_2 = t_1 \to t_3)_3, (t_4 = t_3)_4 \\ f(t_7) &\iff (t_5 = Bool)_5, (t_6 = Bool)_6, (t_7 = t_5 \to t_6)_7 \end{aligned}$$

[2] The desugared definition of $f$ is $f = \lambda x.(\texttt{case}\ x\ \texttt{of}\ True \to True)$ creating a much bigger but equivalent set of constraints.

$$\text{(Var-x)}\qquad \frac{(x : t) \in \Gamma \quad t_l \text{ fresh}}{E, \Gamma, x_l \vdash_{Cons} ((t_l = t)_l \mid t')}$$

$$\text{(Var-p)}\qquad \frac{f : \forall \bar{a}.C \Rightarrow t \in E \quad \bar{b}, t_l \text{ fresh}}{E, \Gamma, f_l \vdash_{Cons} ([\overline{b/a}]C)_l \wedge (t_l = [\overline{b/a}]t)_l \mid t')}$$

$$\text{(Var-f)}\qquad \frac{f : \sigma \notin E \quad t_l \text{ fresh}}{E, \Gamma, f_l \vdash_{Cons} (f(t_l)_l \mid t_l)}$$

$$\text{(Abs)}\qquad \frac{E, \Gamma.x : t_{l_1}, e \vdash_{Cons} (C \mid t) \quad t_{l_1}, t_{l_2}, t' \text{ fresh}}{E, \Gamma, (\lambda x_{l_1}.e)_{l_2} \vdash_{Cons} (C \wedge (t_{l_2} = t' \to t)_{l_2} \wedge (t_{l_1} = t')_{l_1} \mid t_{l_2})}$$

$$\text{(App)}\qquad \frac{E, \Gamma, e_1 \vdash_{Cons} (C_1 \mid t_1) \quad \Gamma, e_2 \vdash_{Cons} (C_2 \mid t_2) \quad t_l \text{ fresh}}{E, \Gamma, (e_1\ e_2)_l \vdash_{Cons} (C_1 \wedge C_2 \wedge (t_1 = t_2 \to t_l)_l \mid t_l)}$$

$$\text{(Case)}\qquad \frac{\begin{array}{c} E, \Gamma, e \vdash_{Cons} (C_e \mid t_e) \\ E, \Gamma, (p_i \to e_i) \vdash_{Cons} (C_i \mid t_i) \quad \text{for } i \in I \\ C \equiv \bigwedge_{i \in I}((t_{l_1} = t_i)_{l_1} \wedge C_i) \wedge (t_{l_1} = t_e \to t_{l_2})_{l_2} \quad t_{l_1}, t_{l_2} \text{ fresh} \end{array}}{E, \Gamma, (\textsf{case } e \textsf{ of } ([p_i \to e_i]_{i \in I})_{l_1})_{l_2} \vdash_{Cons} (C \mid t_{l_2})}$$

$$\text{(Pat)}\qquad \frac{\begin{array}{c} p \vdash_{Cons} (C_p \mid t_p \mid \Gamma') \quad E, \Gamma \cup \Gamma', e \vdash_{Cons} (C_e \mid t_e) \\ C \equiv C_p \wedge C_e \wedge (t_l = t_p \to t_e)_l \quad t_l \text{ fresh} \end{array}}{E, \Gamma, (p \to e)_l \vdash_{Cons} (C \mid t_l)}$$

$$\text{(Pat-Var)}\qquad \frac{t \text{ fresh}}{x_l \vdash_{Cons} (True \mid t_l \mid \{x : t_l\})}$$

$$\text{(Pat-K)}\qquad \frac{\begin{array}{c} p_i \vdash_{Cons} (t_{p_i} \mid C_{p_i} \mid \Gamma_{p_i}) \quad \text{for } i = 1, ..., n \\ K : \forall \bar{a}.t_K \quad \Gamma_p = \Gamma \cup \bigcup_{i=1,\ldots,n} \Gamma_{p_i} \quad t_l \text{ fresh} \\ C' \equiv (t_K = t_{p_1} \to ... \to t_{p_n} \to t_l)_l \wedge \bigwedge_{i \in \{1,..,n\}} C_{p_i} \end{array}}{(K\ p_1\ ...\ p_n)_l \vdash_{Cons} (C' \mid t_l \mid \Gamma_p)}$$

**Fig. 3.** Justified Constraint Generation

*Example 8.* The (location annotated) program

```
h :: (Int -> (Int,Int))₁
(h x₂ = (x₃, x₄)₅)₆
```

is translated to

$$\begin{array}{ll} h_a(t_1) & \Longleftrightarrow (t_1 = Int \to (Int, Int))_1 \\ h(t_6) & \Longleftrightarrow (t_2 = t_x)_2, (t_3 = t_x)_3, (t_4 = t_x)_4, (t_5 = (t_3, t_4))_5, (t_6 = t_2 \to t_5)_6 \end{array}$$

In this framework it is now easy to see the correspondences between typing questions and constraint algorithms.

**Type Inference** Type inference for an expression $e$ corresponds to building a canonical normal form of the constraint generated from $e$. Type inference of a function $f$ simply involves executing the goal $f(t) \longrightarrow_P^* C$. The type of $f$ is $\bar{\exists}_t C$.

*Example 9.* For the program of Example 7, if we wish to infer the type of $g$, we determine $C$ such that $g(t) \longrightarrow_P^* C$. The generated constraint is shown in Example 5. Since the resulting constraints are not satisfiable $g$ has no type.

*Example 10.* Consider the program in Example 8. The goal $h(t) \longrightarrow_P^* C_1$ generates the constraint

$$C_1 \equiv t = t_6 \wedge (t_2 = t_x)_2 \wedge (t_3 = t_x)_3 \wedge (t_4 = t_x)_4 \wedge (t_5 = (t_3, t_4))_5 \wedge (t_6 = t_2 \rightarrow t_5)_6$$

which is satisfiable. A simplified equivalent constraint to $\bar{\exists}_t C_1$ is $\exists t_x . t = t_x \rightarrow (t_x, t_x)$ which we report as the type of $h$ as $h : \forall t_x . t_x \rightarrow (t_x, t_x)$.

**Type Checking** Type checking of a function definition $f = e$ with respect to its declared type $f : C \Rightarrow t$ requires us to test implication. Since we have two constraints defining the inferred and declared type we simply need to check implication. Let $f(t) \longrightarrow_P^* C$ and $f_a(t) \longrightarrow_P^* C'$. Then the declared type is correct if $\bar{\exists}_t C' \supset \bar{\exists}_t C$. We can use the implication checking algorithm discussed in Section 2.

*Example 11.* Consider the program in Example 8. The goal $h_a(t) \longrightarrow_P^* C_2$ generates

$$\begin{aligned} C_2 &\equiv (t = Int \rightarrow (Int, Int)_1 \\ C_1 \wedge C_2 &\equiv (t = Int \rightarrow (Int, Int))_1 \wedge C_1 \end{aligned}$$

The corresponding substitutions are identical on $t$. Hence the declared type is correct.

## 5 Type Error Reporting

The most important insight we gain from understanding typing problems as constraint problems in the case of pure Hindley/Milner types is what to do when it goes wrong! Since we have mapped typing questions to constraint questions, we immediately have more insight into why failure occurred. In this section we consider what it means about the corresponding constraint problem when type inference or type checking fails. We then use this to define better error messages.

### 5.1 Failure of Type Inference

A program is ill-typed if the constraints on its type are unsatisfiable. Before the program can be run, it must be modified, but obviously any such modification must actually fix the problem at hand. Our task then, is to report the type error in such a way that the programmer is directed towards the locations in the source code which are potentially the source of the error, and if modified appropriately, would fix the program.

Suppose type inference fails for a function $f$, then we have an unsatisfiable set of constraints $C$ arising from $f(t) \longrightarrow_P^* C$. The key insight we obtain from the constraint view is this:

*A type error results from a minimal unsatisfiable set of constraints.*

We dont need to consider all constraints in $C$ to have a type error. Hence we should report errors as minimal unsatisfiable sets of constraints. Note there are many possible minimal unsatisfiable sets, and different sets will generate different error reports (see [35] for examples).

We can find $M$ a single minimal unsatisfiable subset of $C$, employing the algorithm of Section 2 (we will just take the first generated). Such a set represents a "smallest" type error, and the corresponding locations give a smallest collection of program locations which caused the error. The simplest scheme for reporting an error is simply to highlight all the locations of the source text which make up a type error.

*Example 12.* When we try to infer a type for $g$ in Example 7 we obtain the constraints shown in Example 5. Since these are unsatisfiable we find a minimal unsatisfiable subset as shown in Example 6. The set of locations involved are $\{1, 2, 3, 5, 7, 8\}$ The type error can be reported as

```
g = f 'a'
f True = True
```

This indicates a conflict between the application of `f` to `'a'` in `g`, and `f`'s pattern. Importantly, because we have used a minimal unsatisfiable subset of the inconsistent constraints, we have only highlighted the locations which are actually involved in the error; the `True` in the body of `f` is not part of the conflict, and therefore not highlighted.

Note that we do not highlight applications since they have no explicit tokens in the source program. We leave it to the user to understand when we highlight a function position we may also refer to its application.

To remain efficient, we only consider a single minimal unsatisfiable subset of constraints at a time. Given the number of constraints generated during inference, calculating all minimal unsatisfiable subsets is simply not feasible. As mentioned in Section 2, however, it is inexpensive to find any constraints which appear in all minimal unsatisfiable subsets.

For type error reporting purposes, finding a non-empty intersection of all minimal unsatisfiable subsets is significant, since those constraints correspond to source locations which are part of every type conflict present. These common locations are much more likely to be the actual source of the mistake.

*Example 13.* The following simple program, where functions `toUpper` and `toLower` are standard Haskell functions both with type $Char \rightarrow Char$, is ill-typed.

$(\texttt{f}\ \texttt{x}_2\ \texttt{=}\ (\texttt{if}_3\ \texttt{x}_4\ \texttt{then}_5\ (\texttt{toUpper}_6\ \texttt{x}_7)_8\ \texttt{else}_9\ (\texttt{toLower}_{10}\ \texttt{x}_{11})_{12})_1$

It's plain to see that there is a conflict between the use of `x` at type *Bool* in the conditional, and at type *Char* in both branches of the `if-then-else`. Hence, there are two minimal unsatisfiable subsets of the above constraints. Common to both of these are the (location annotated) constraints listed below.

$$(t_3 = Bool)_3, (t_4 = t_1)_4,$$

This strongly suggests that the real source of the mistake in this program lies at location 3 or 4. We might report this by highlighting the source text as follows.

```
f x = (if x then (toUpper x) else (toLower x))
```

Indeed, changing the expression at location 4 to something like `x > 'm'` would resolve both type conflicts, whereas changing either of the two branches would only fix one.

Just highlighting the locations causing a type error is not very informative. Usually type errors are reported as an incompatibility of two types, an expected type and an inferred type. Given our much more detailed constraint viewpoint we can do better. 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 from the minimal unsatisfiable set to report the type conflict about
2. Find the types that conflict at that location
   – Assign each a colour and determine which locations contribute to it
3. Diagnose the error in terms of the conflicting types at the chosen location. Highlight each location involved in the colours of the types it contributes to.

Although we can pick any location, we have found that usually the highest location in the abstract syntax tree occuring in the minimal unsatisfiable subset leads to the clearest error messages. If $l$ is the highest location appearing in $M$, we remove all equations added by location $l$ to obtain $M'$. Now $M'$ is satisfiable (since we have removed at least one equation from a minimal unsatisfiable set) and we can use it to determine the types reported. We will choose locations $l'$ to report the types of depending on the kind of location $l$. Importantly if $\phi = mgu(M')$ and $\phi(t_{l'}) = t'$ we report the type of location $l'$ as $t'$ and highlight the locations from $M'$ of a minimal implicant of $t_{l'} = t'$.

We can define a specific type error for each different kind of location. For brevity we just give an example, see [31, 35] for more details.

*Example 14.* For a location corresponding to incompatible types for pattern/actions in a case $([p_i \rightarrow e_i]_{i \in I})_l$ we remove all the constraints of the form $(t_l = t_i)_l$ occuring in $M$. We now report the types $t_i$ of each pattern/action entry $p_i \rightarrow e_i$ as defined by $M'$.

Example 1 is an example of incompatible types of a pattern/actions (in the desugared version). For this example, we remove the equations forcing each clause for `f` to have the same type. We then determine the type of each clause independently, and find minimal implicants of these types. By highlighting each type and its implicant locations in the same color we can see why the types arose.

Note that the types only consider constraints in the minimal unsatisfiable subset, so that the type of the first alternative is reported as `Char -> a -> b -> c` rather than `Char -> a -> Bool -> c` which we might expect.

## 5.2 Failure of Type Checking

When type checking fails for $f : C \rightarrow t; f = e$ we have that $f(t) \longrightarrow_P^* C$ and $f_a(t) \longrightarrow_P^* C'$ and its not the case that $\bar{\exists}_t C' \supset \bar{\exists}_t C$. If $\bar{\exists}_t C$ is *False* we have a failure of type inference and can report a problem as in the previous subsection. Otherwise we can choose any constraint in $\bar{\exists}_t C$ not implied by $\bar{\exists}_t C'$.

There are choices in how to do this. Currently Chameleon chooses in the following way. We consider the substitutions $\phi = mgu(C')$ and $\phi' = mgu(C \wedge C')$. Choose a variable $a \in \phi(t)$ where $\phi'(a) \neq a$. We then determine the minimal subset $D$ of $C$ such that $D \wedge C' \supset a = \phi'(a)$. We make use of the min_impl algorithm described in Section 2 to find $D$. This describes a minimal reason why the variable $a$ was bound in the inferred type.

*Example 15.* Consider the following modification of the program in Example 8

$\texttt{h :: (a -> (a,b))}_1$
$\texttt{(h x}_2\texttt{ = (x}_3\texttt{, x}_4\texttt{)}_5\texttt{)}_6$

is translated to

$$\begin{array}{ll} h_a(t_1) & \Longleftrightarrow (t_1 = (a \rightarrow (a,b)))_1 \\ h(t_6) & \Longleftrightarrow (t_2 = t_x)_2, (t_3 = t_x)_3, (t_4 = t_x)_4, (t_5 = (t_3, t_4))_5, (t_6 = t_2 \rightarrow t_5)_6 \end{array}$$

We find that $\phi = \{t \mapsto a \rightarrow (a,b)\}$ while $\phi' = \{t \mapsto a \rightarrow (a,a), t_x \mapsto a, b \mapsto a\}$. We find $\phi'(b) \neq b$. We determine a minimal implicant of $C \wedge C'$ for $b = a$, which is $\{(t_1 = (a \rightarrow (a,b)))_1, (t_6 = t_2 \rightarrow t_5)_6, (t_5 = (t_3, t_4))_5, (t_2 = t_x)_2, (t_4 = t_x)_4, \}$. The resulting set of locations are highlighted.

```
h.hs:2: ERROR: Inferred type does not subsume declared type
Declared: forall a,b. a -> (a,b)
Inferred: forall a. a -> (a,a)
Problem : The variable ‘b’ makes the declared type too polymorphic
          h x = (x, x)
```

The GHC error message explains the same problem in inferred and declared type but can't point us at any location that caused the problem.

## 6 Type Class Overloading

Type classes and instances are a popular extension of Hindley/Milner types that give controlled overloading. We now extend our notion of constraints to incorporate classes and instances. Again we use CHRs to encode their meaning. We then revisit the typing questions once more. The class declaration and instance declaration generate the following CHRs:

| | |
|---|---|
| $\mathsf{class}\ (C \Rightarrow TC\ \bar{a})_{l_1}\ \mathsf{where}\, m :: (D \Rightarrow t)_{l_2}$ | $TC\ \bar{a} \Longrightarrow C_{l_1}$ |
| | $m_a(t) \Longleftrightarrow t = t_{l_2}, D_{l_2}, (TC\ \bar{a})_{l_2}$ |
| $\mathsf{instance}\ E \Rightarrow (TC\ \bar{t})_{l_3}$ | $TC\ \bar{t} \Longleftrightarrow E_{l_3}$ |

The first rule ensures the super-class constraint hold, that is if $TC\ \bar{a}$ then the super class constraints $C$ also hold. The location annotation ensure we see they arise from the class declaration. The second rule defines the type of the method $m$. The third rule encodes the proof that an instance is available through the instance rules. We omit instance method declarations for simplicity, they simply create more type checking.

*Example 16.* The table below shows class and instance declarations below and their translation (where we ignore instance method declarations for brevity).

| | |
|---|---|
| `class (Eq a)`$_1$ `where` | $Eq\ a \Longrightarrow True_1$ |
| `    (==) :: (a -> a -> Bool)`$_2$ | $(==)(t_2) \Longleftrightarrow (t_2 = a \to a \to Bool)_2, (Eq\ a)_2$ |
| `class (Eq a => Ord a)`$_3$ `where` | $Ord\ a \Longrightarrow (Eq\ a)_3$ |
| `    (>) :: (a -> a -> Bool)`$_4$ | $(>)(t_4) \Longleftrightarrow (t_4 = a \to a \to Bool)_4, (Ord\ a)_4$ |
| `instance (Ord a => Ord [a])`$_5$ | $Ord\ [a] \Longleftrightarrow (Ord\ a)_5$ |
| `instance (Ord Bool)`$_6$ | $Ord\ Bool \Longleftrightarrow True_6$ |

Note that the super-class relation encoded by the CHR states that each occurrence of $Ord\ a$ implies $(Eq\ a)_3$. Note that right hand sides of CHRs generated are justified, so we can keep track which rules were involved when inspecting justifications attached to constraints.

Our assumptions on CHRs require that the CHRs generated from class and instance declarations are confluent, terminating, range-restricted and single-headed simplification. The last two properties are easy to check, and, fortunately, the first two properties are guaranteed by the conditions imposed on Haskell type classes. An in-depth discussion can be found in [6].

Type inference in the presence of type classes and instances works as follows. To infer the type of $f$ we determine $f(t) \longrightarrow_P^* C$ and give the inferred type of $f$ as $f :: \phi(C_u) \Rightarrow \phi(t)$ where $\phi = mgu(C_e)$.

## 6.1 Failure of Type Inference

As in the pure Hindley/Milner case a failure of type inference can give an unsatisfiable set of constraints $C$. Unsatisfiability of a set of constraints can only arise through an unsatisfiable set of term equations, since the assumption is that the classes and instances follow an open world assumption, another instance could be added at any time in order to satisfy any remaining class constraints. Hence we can use the same mechanisms as for pure Hindley/Milner.

But there are two new kinds of type error that can now occur.

**Missing Instance Error** In Haskell 98, type classes are single-parameter and each argument of a type class appearing within a functions type $\phi(C_u)$, must be a single variable $(a)$. A non-conforming constraint is one whose arguments have not been reduced to this form, indicating that there is a *missing instance* error.

For a missing instance error to occur a class constraint $T\ a$ must occur in $C$ such that $\phi(a)$ is not a variable. We can determine the reason this constraint occurs in $C$ using minimal implications. Let $L$ be the set of locations occurring on the constraint $(T\ a)_L$. Now $\phi(a)$ is not a variable (or variable applied to arguments) so it is a term with top level type constructor $K$ of arity $n$ say. We determine the minimal implicants in $C$ of $\exists \bar{y}.a = K\ \bar{y}$. Collecting the locations $L'$ of this minimal implicant with the locations $L$ introducing $T\ a$ we have the reasons why the missing instance is involved.

*Example 17.* Re-examining Example 2 from the introduction: the inferred type is $sum :: Num\ [a] \Rightarrow [[a]] \rightarrow [a]$. The missing instance $Num\ [a]$ arise from an initial class constraint $Num\ b$ introduced by `+` and $b = [a]$ which is implied by the result of `sum` in the first definition arising from the `[]` on the right of the equality. Hence we obtain error message shown in Example 2.

**Ambiguity Error** An important restriction usually made on types is that they are *unambiguous*. A type $\bar{\exists}_t C$ is unambiguous if fixing $t$ fixes all the existentially quantified variables in $\bar{\exists}_t C$. Programs with ambiguous types can lead to operationally nondeterministic behaviour.

We can use CHRs to check unambiguity as follows. A type $\bar{\exists}_t C$ is unambiguous if $\rho$ is a renaming of $fv(C)$ we determine $C \wedge \rho(C) \wedge t = \rho(t) \longrightarrow_P^* D$. If $D \supset a = \rho(a)$ for all $a \in fv(C)$ then $f$ is unambiguous.

In reporting ambiguity we highlight the locations where the ambiguous variable is part of the type, since each such location could be improved to remove the ambiguity. For ease of reporting we only consider only a variable $a \in \bar{a} = fv(\phi(C_u))$ where $\phi = mgu(C_e)$. If the type $\bar{\exists}_t C$ is ambiguous, then the test must fail for one of these variables. Examining a location variable $t_l$ we can see if $a$ occurs in its type, if $a \in fv(\phi(t_l))$. We highlight all locations where this test succeeds.

*Example 18.* Consider the following program, where $read :: Read\ a \Rightarrow [Char] \rightarrow a$ and $show :: Show\ a \Rightarrow a \rightarrow [Char]$,

```
f x y z = show (if x then read y else read z)
```

The inferred type is ambiguous since the type $a$ of `read y` and `read z` does not appear in the type of $f$. GHC reports the error as follows

```
amb.hs:3:26:
    Ambiguous type variable ‘a’ in the constraints:
      ‘Read a’ arising from use of ‘read’ at amb.hs:3:26-29
      ‘Show a’ arising from use of ‘show’ at amb.hs:3:10-13
    Probable fix: add a type signature that fixes these type variable(s)
```

Chameleon highlights the positions where the type variable $a$ appears as part of the type:

```
ambig.ch:9: ERROR: Inferred type scheme is ambiguous:
Type scheme: forall a. (Read a, Show a) => Bool -> [Char] -> [Char] -> [Char]
Suggestion: Ambiguity can be resolved at these locations
            f x y z = show (if x then read y else read z)
```

illustrating that the ambiguity can be removed by type annotations on either call to read, or on the if-then-else. Note how effectively GHC picks just one instance of `read` to concentrate on.

### 6.2 Failure of Type Checking

Now type checking can fail in a new way, since the types are no longer simply sets of equations. We now have to consider that a type class constraint is not implied. This ends up actually easier than the case for equations.

Recall that $f(t) \longrightarrow_P^* C$ and $f_a(t) \longrightarrow_P^* C'$. Let $\phi = mgu(C'_e)$ and $\phi' = mgu(C_e \wedge C'_e)$. Suppose we have a constraint $\phi'(T\ \bar{a}) \in \phi'(C_u)$ such that $\phi'(T\ \bar{a}) \notin \phi(C_u)$. Suppose $(T\ \bar{a})_L$ is the location annotated version of this constraint in $C_u$, we highlight the locations $L$ which causes the unmatched class constraint to arise.

*Example 19.* Consider the following program

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

The inferred type is $Ord\ a \wedge Eq\ a \Rightarrow a \rightarrow Bool$, while the declared type is $Eq\ a \Rightarrow a \rightarrow Bool$. We determine the locations that cause the $Ord\ a$ class constraint to arise, and highlight them.

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 unmatched 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 [16] and nhc98 [23], however, report the error without identifying the program locations responsible.

## 7 Extended Type Systems

We now consider further extensions to Hindley/Milner types and how they can be incorporated. Chameleon [32] supports all the features we discuss below.

## 7.1 Functional Dependencies

Functional dependencies [17] are an important extension for multi-parameter type classes. They allow the programmer to specify depencies between arguments of multi-parameter type classes, and hence improve type inference. Classes with functional dependencies are translated using additional CHRs for each functional dependency, and for each instance and functional dependency.

*Example 20.* The following type class models a collection relationship $ce$ is a collection of $e$s.

```
class (Collects ce e)₁ | (ce -> e)₂ where
    empty :: ce
    insert :: e -> ce -> ce
instance (Collects Integer Bool)₃ where ...
instance Eq a => (Collects [a] a)₄ where ...
```

The functional dependency $ce \rightarrow e$ states that there is at most one element type $e$ for each collection type $ce$. Without this `empty` is ambiguous.

The additional CHRs are

$$\begin{aligned} Collects\ ce\ e1, Collects\ ce\ e2 &\Longrightarrow (e1 = e2)_2 \\ Collects\ Integer\ b &\Longrightarrow (b = Bool)_{[2,3]} \\ Collects\ [a]\ b &\Longrightarrow (b = a)_{[2,4]} \end{aligned}$$

The first enforces the functional dependency on two *Collects* constraints with the same collection type. The last two are improvement rules for each instance. Once we know the collection type is *Integer*, we know the element type is *Bool*, and once we know the collection type is $[a]$ we know the element type is $a$.

We can show (see [6]) that Haskell programs with functional dependencies satisfying the restrictions in [17] lead to confluent, terminating, range-restricted and single-headed simplication programs, so our type framework is usable without modification. The only new difficulty arises in error reporting. Functional dependencies can create unsatisfiable constraints where the location $l$ occuring in a justification but $t_l$ does not appear in the constraints. We overcome this by reporting the error on the usage of the functional dependency.

*Example 21.* The function

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

is incorrect since we cannot have a *Bool* and *Char* in the same collection. 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 generalizing 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)
```

Note here we have multiple "colour" highlighting. The calls to `insert` both generate *Collects* constraint and define the types of variables $e1$ and $e2$ so they are highlighted in both ways.

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.

## 7.2 Adhoc type improvements

In Chameleon the user is allowed to write their own CHRs, which become part of the program $P$. This can be used to improve type inference and checking.

*Example 22.* Consider the following class and instance building a `zip`-like function `zipall` for zipping an arbitrary number of arguments:

```
class Zip a b c | c -> b, c -> a where
    zipall :: [a] -> [b] -> c
instance Zip a b [(a,b)] where zipall = zip
instance Zip (a,b) c e => Zip a b ([c]->e) where
    zipall as bs cs = zipall (zip as bs) cs
```

As it stands type inference for

```
e = head (zipall ['a','b'] [True,False] ['c'])
```

will return $e :: \forall a. Zip\ (Char, Bool)\ Char\ [a] \Rightarrow a$. We can add an improving propagation rules to enforce that whenever the third argument of a *Zip* constraint is a list type, rather than a function type, it is a list of pairs of the first two. In Chameleon format this is

```
rule Zip a b [c] ==> c = (a,b)
```

With this rule we infer $e :: ((Char, Bool), Char)$ as expected.

Arbitrary rule additions may break confluence, termination, range-restrictedness and single-headed simplification (the last two of which we can check). Currently we assume the user enforces confluence and termination. We can handle type error reporting with adhoc rules using the same approach as for functional dependencies, choosing the last rule fired. This does highlight the future need for better error reporting by explaining a sequence of CHR rule firings.

### 7.3 Extended Algebraic Data Types

Guarded algebraic data types illustrated in Example 4 significantly complicate type processing. Chameleon supports GADTs through a more generalized form, Extended Algebraic Data Types (EADTs) [33] which also generalizes existential types. EADTs extend the translation to constraints to include quantified implication constraints of the form

$$\text{ImpConstraints } F ::= C \mid \forall \bar{b}.(C \supset \exists \bar{a}.F) \mid F \wedge F$$

This makes type inference in general impossible, since there may be an infinite number of maximal types, so we concentrate on type checking. Essentially the type checking procedure must check that the implication constraint $\forall \bar{b}.(C \supset \exists \bar{a}.F)$ is implied by the declared type $C'$. In checking this implication we effectively check if $C \wedge C' \supset \exists \bar{a}.F$. If the implication fails we have a subsumption error like that illustrated in Example 4. See [35] for an extended discussion of type errors for EADTs.

## 8 Related Work

The starting point for this work was [5] which translated Hindley/Milner types to a set of Horn clauses rather than CHRs. The advantage of CHRs is that we can easily accommodate more advanced type extensions like type classes. Another difference to [5] is that we attach justifications to constraints to keep track of program locations.

Despite recent efforts [4, 21, 15, 10], we believe there remains a lot of scope for improving the quality of type error diagnoses. For example, almost all other work we are aware of has focused on the plain Hindley/Milner type system and excludes features like type-class overloading [34] which are critical in languages like Haskell and Clean (the one exception is the recent paper by Heeren and Hage [13]).

The standard algorithm, $\mathcal{W}$, tends to find errors too late in its traversal of a program [18, 37]. $\mathcal{W}$ has been generalised [18] 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 [12, 14, 15], our aims are different. We attempt to explain errors involving advanced type system features, such as overloading, whereas the Helium system [15], which is based on a beginner-friendly variant of Haskell, omits such features by design. Their focus has been on inventing heuristics which allow them to present type errors from a more useful perspective, as well as automatically suggesting "probable fixes." More recently [13] they propose extending their source language with so-called 'type class directives', which provide restrictions on certain forms of type classes (such as making $Num\ [a]$ illegal). These can be straightforwardly encoded using Chameleon rules.

Closest to our work is probably that of Haack and Wells [10] who also, independently, propose using minimal unsatisfiable subsets of typing constraints to identify problematic program locations. The main difference between their work and ours is that they focus entirely on the standard Hindley/Milner system, limiting their constraint domain to equations, and only report errors by highlighting the locations involved. Another limitation of their proposal is that it lacks any way to generate type explanations, which we do by finding minimal implicants. Such a facility is necessary for explaining subsumption errors.

Another related research direction is error explanation systems [7, 2], which allow the user to examine the process by which specific types are inferred for program variables. By essentially recording the effects of the inference procedure on types a step at a time, a complete history can be built up. Unfortunately, a common shortcoming of such systems is the excessive size of of explanations. Although complete, such explanations are full of repetitive and redundant information which can be a burden to sort through. Furthermore, since these systems are layered on top of an existing inference algorithm, they suffer from the same left-to-right bias when discovering errors.

## 9 Conclusion

We have presented a flexible type processing system for Hindley/Milner types and extensions which naturally supports advanced type error reporting and reasoning techniques. The central idea of our approach is to translate the typing problem to a constraint problem, i.e. a set of constraints where function relations are expressed in terms of CHRs. Individual constraints are justified by the location of their origin. During CHR solving we retain these locations. CHRs are a sufficiently rich constraint language to encode the typing problem for a wide range of extensions of the Hindley/Milner system such as type-class overloading and functional dependencies. The techniques explained in this paper have all been implemented as part of the Chameleon system [32] which is freely available.

The basic machinery we use here can also be used in an interactive type debugging framework (see [29, 30]). Clearly as type systems become more and more complicated these interactive forms of type debugging, which for example can explain why a function has an inferred type of a certain shape, become much important. We can also straightforwardly extend our approach to create specialised error messages for library functions or CHR rules in the manner of Helium (see [35] for details).

By lifting type algorithms from adhoc specialized algorithms to generic constraint reasoning algorithms our approach offers the advantages of uniformity (allowing easier handling of extensions) as well as a clear semantics (which for example allowed us to give the first proof of the soundness and completeness of Jones functional dependency restrictions [6]). As types become more complicated, we need to make use of the existing deep understanding of constraints and first order predicate logic, in order to handle them correctly. Typing problems will also inevitably push us to develop new constraint algorithms, for example constraint abduction [19] seems required for inference of GADTs.

## 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. J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.
4. O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proc. of ICFP'01*, pages 193–204. ACM Press, 2001.
5. 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.
6. G. J. Duck, S. Peyton-Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.
7. D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.
8. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer-Verlag, 1995.
9. M. García de la Banda, P.J. Stuckey, and J. Wazny. Finding all minimal unsatisfiable constraints. In *Proc. of PPDP'03*, pages 32–43. ACM Press, 2003.
10. 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.
11. Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.
12. B. Heeren and J. Hage. Parametric type inferencing for Helium. Technical Report UU-CS-2002-035, Utrecht University, 2002.
13. B. Heeren and J. Hage. Type class directives. In *Proc. of PADL 2005*, 2005.
14. B. Heeren, J. Hage, and D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Utrecht University, 2002.
15. Helium home page. `http://www.cs.uu.nl/~afie/helium/`.
16. Hugs home page. `http://www.haskell.org/hugs/`.
17. M. P. Jones. Type classes with functional dependencies. In *Proc. ESOP'00*, volume 1782 of *LNCS*, pages 230–244. Springer-Verlag, March 2000.
18. 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.
19. M.J. Maher. Herbrand constraint abduction. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 397–406. IEEE Computer Society, 2005.
20. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction.* MIT Press, 1998.
21. B.J. McAdam. Generalising techniques for type debugging. In *Trends in Functional Programming*, pages 49–57, March 2000.

22. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.
23. nhc98 home page. haskell.org/nhc98/.
24. H. Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proc. of ICFP'05*, pages 54–65. ACM Press, 2005.
25. F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Proc. of POPL'04*, pages 89–98. ACM Press, January 2004.
26. T. Sheard and E. Pasalic. Meta-programming with built-in type equality. In *Fourth International Workshop on Logical Frameworks and Meta-Languages*, 2004.
27. J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.
28. P.J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems*, 27(6):1216–1269, 2005.
29. P.J. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon type debugger (tool demonstration). In M. Ronsse, editor, *Proceedings of the Fifth International Workshop on Automated Debugging*, pages 247–260, 2003. http://arxiv.org/html/cs.SE/0309027.
30. P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In J. Juring, editor, *Proceedings of the ACM SIGPLAN 2003 Haskell Workshop*, pages 72–83. ACM Press, 2003.
31. P.J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Proceedings of the ACM SIGPLAN 2004 Haskell Workshop*, pages 80–91. ACM Press, 2004.
32. M. Sulzmann and J. Wazny. Chameleon. `http://www.comp.nus.edu.sg/~sulzmann/chameleon`.
33. M. Sulzmann, J. Wazny, and P.J. Stuckey. A framework for extended algebraic data types. In P. Wadler and M. Hagiya, editors, *Proceedings of 8th International Symposium on Functional and Logic Programming*, number 3945 in LNCS, pages 47–64. Springer-Verlag, April 2006.
34. 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.
35. J. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions.* PhD thesis, University of Melbourne, 2006. `http://www.comp.nus.edu.sg/~sulzmann/chameleon/thesis.ps.gz`.
36. H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. of POPL'03*, pages 224–235. ACM Press, 2003.
37. 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.