

Type Classes and Constraint Handling Rules

Kevin Glynn, Peter J. Stuckey and Martin Sulzmann
Dept. of Computer Science and Software Engineering,
The University of Melbourne,
Parkville 3052, Australia.
{keving,pjs,sulzmann}@cs.mu.oz.au

Abstract

Type classes are an elegant extension to traditional, Hindley-Milner based typing systems. They are used in modern, typed languages such as Haskell to support controlled overloading of symbols. Haskell 98 supports only single-parameter and constructor type classes. Other extensions such as multi-parameter type classes are highly desired but are still not officially supported by Haskell. Subtle issues arise with extensions, which may lead to a loss of feasible type inference or ambiguous programs. A proper logical basis for type class systems seems to be missing. Such a basis would allow extensions to be characterised and studied rigorously. We propose to employ Constraint Handling Rules as a tool to study and develop type class systems in a uniform way.

1 Introduction

The Haskell language [PHA⁺99] provides one of the most advanced type systems in an industrial-strength language. Type classes are one of the most distinctive features of Haskell. The form of type classes found in Haskell 98 is restricted to single-parameter and constructor [Jon93] type classes. A rigorous treatment of the Haskell 98 type system can be found in [Jon99], it fills some serious gaps in the current specifications of Haskell.

Since the original papers [Kae88, WB89] on type classes, many researchers have studied extensions to the existing type class system [CHO92, JJM97]. In particular, multi-parameter type classes are a very desirable extension, see [JJM97] for an overview. However, multi-parameter type classes are still not officially supported by Haskell. As the authors note, design decisions need to be taken with great care in order to retain feasible type inference.

Existing implementations of Haskell use a dictionary passing translation to support type class overloading. This requires types to be unambiguous, otherwise an implementation can't know which dictionary should be passed. Unfortunately, even with the existing single-parameter class system, ambiguous types can occur. In his recent paper [Jon00], Jones extends type classes with functional dependencies to resolve ambiguity in the context of multi-parameter classes for certain cases. We find that type classes are still an active area of research and the debate about which features should be incorporated into future Haskell specifications is far from settled.

This work. In contrast to previous work, our foremost goal is not to propose yet another extension of type classes. The thesis of this paper is: *Constraint handling rules [Fru95] are the right way to understand type class constraints, and extensions to type classes. In particular, constraint handling rules help us understand the two main issues behind possible type class extensions: Feasible type inference and unambiguous programs.*

Feasible type inference is an important property which needs to be retained when considering type class extensions. Type inference should be decidable and should compute principal types.

Hindley-Milner types are characterized by constraints that are representable using Herbrand constraints, and solvable using well-understood constraint solvers such as Robinson’s unification algorithm. Clearly, type classes are simply another form of constraint system, extending the Herbrand constraints, that constrains the value that various type variables can take.

Constraint handling rules (CHRs) are a way of extending constraint solving from a well-understood underlying constraint domain to handle new forms of constraints. CHRs are a simple language and efficient implementations are available. They give a natural definition of a constraint solver for type classes, they clarify some issues about what meaning should be given to type class extensions and they give insight into problems such as ambiguity, overlapping instances of type classes and multi-parameter type classes. Moreover, CHRs allow us to specify the conditions under which feasible type inference is guaranteed.

For the purpose of this workshop paper, we explain our ideas by example rather than by giving a rigorous formal treatment.

Outline. Section 2 motivates our approach by reviewing limitations with Haskell’s type class system, which can be overcome with extensions defined by CHRs. In Section 3, we review the basic ideas behind CHRs. Type inference is described in Section 4. Section 5 states some sufficient conditions under which we achieve feasible type inference and unambiguous programs. In Section 6, we show how to express some (previously proposed in the literature) type class extensions in terms of CHRs. In Section 7, we show that CHRs prove to be useful for defining novel type class extensions. We conclude in Section 8.

2 Motivation

Type classes are an elegant extension to traditional, Hindley-Milner based typing systems. In addition to supporting controlled overloading of functions they allow programmers (and language designers) to identify related types. For example, when we make our new type an instance of the `Eq` class; not only are we adding the convenience of the `==` operator we are telling users of this type that its inhabitants are exact and identifiable. Note that if we choose not to make our type an instance of `Eq` we are also making a statement: inhabitants are representing values which are inexact and/or hard to identify.

Unfortunately, in Haskell 98 it is impossible to enforce any restrictions on class membership. Worse, instance declarations are global and a program can only have one instance declaration for a type and a particular class. This requires that any instance declaration visible in a module is exported to all modules which import it, bypassing Haskell’s name hiding mechanisms. So, if a module declares `Bool` to be a member of the `Num` class then any importing module also treats `Booleans` as a member of `Num`. The writer of the importing module may be completely unaware that this has happened. Now ‘errors’ such as

```
import X()                -- X makes Bool an instance of Integral

cap_power :: (Num a,Integral b) => Bool -> b -> a -> b -> a
cap_power useLimit theLimit n p
  | useLimit && theLimit < p = n ^ useLimit  -- surely intended 'theLimit'
  | otherwise                = n ^ p
```

will compile just fine, though with bizarre consequences.

Many researchers have studied extensions which would make them more useful, while maintaining Haskell’s decidable type inference. In particular, extensions have been proposed to support multi parameter type classes and allow relationships amongst components of a type class other than the standard super/sub class relations. These extensions are ad-hoc, requiring special syntax and ‘hidden’ restrictions to be added to the language in order to guarantee principal types and a decidable type inference algorithm.

CHRs provide a framework which allows much more freedom for the programmer and language designer to specify constraints on type classes. These extra constraints can serve both to make more programs typable (since the more powerful constraints remove potential ambiguity of types in a program) and to make less programs typable (since the class constraints can restrict the use of types to those intended by the class writer).

CHRs will be described in the next section, but for the remainder of this section we will introduce some examples to give a feel for how CHRs can make the language both safer and more expressive.

2.1 Disjoint Classes

Haskell 98's support for type classes is too restrictive even without adding multi-parameter type classes to the language. For example, we would like to say that the `Integral` and `Fractional` type classes are disjoint. This cannot be expressed in current Haskell and hence the function

```
f x y = x / y + x 'div' y
```

has an inferred type of `f :: (Integral a, Fractional a) => a -> a -> a` rather than immediately causing a type error.

Disjoint classes can also be used to resolve ambiguities in a program. Overlapping type class instances are a thorny issue for Haskell implementations. In general, overlapping instances lead to unacceptable ambiguity in programs. But with improved class information it would sometimes be clear that overlapping instances do not really overlap at all. For example imagine a “has division operator” class `Dividable` defined as follows:

```
class Num a => Dividable a where dividedBy :: a -> a -> a

instance Fractional a => Dividable a where dividedBy = (/)
instance Integral a => Dividable a where dividedBy = div

halfish :: Dividable a => a -> a
halfish x = x 'dividedBy' 2
```

Although the instances appear to overlap, we know that the `Integral` and `Fractional` type classes are intended to be disjoint, therefore there should be no ambiguity here. Notice that this also implies a further extension to GHC's existing, experimental support for overlapping instances. Currently, GHC ignores the instance constraints when deciding if two instance declarations overlap.

A special case of disjoint sets allows us to specify instance declarations for certain types to be an error. For example, we could add constraints to the prelude so that programmers can't make types such as `Bool`, `Char` and function types instances of the `Num` class. This can be done directly for each type or we can declare a disjoint set for `Num`, e.g. `NotNum`, and make the types instances of this disjoint class.

2.2 Multi-Parameter Classes

As Jones [Jon00] points out, the need to extend type classes to a relation over types is well understood, and most Haskell implementations support multi-parameter type classes by non-standard extensions. However, in practice they haven't worked as well as was hoped. Many useful programs have ambiguous types or fall foul of syntactic restrictions required for feasible type inference. By allowing more expressive constraints between the elements of a class relation, these programs can be typed, and in addition the class designer has finer control over their use. Below, we show some useful relationships using the `Collects` class, as presented by Jones, as an example.

```

class Collects e ce where
  empty :: ce
  insert :: e -> ce -> ce
  member :: e -> ce -> Bool

```

Functional Dependencies Jones proposes to extend type classes with *Functional Dependencies* to support dependencies amongst Class components. These give the programmer the necessary control over the allowed relationships to allow additional programs to be typed. The `empty` method of `Collects` has an ambiguous type (since only `ce` is fixed and there may be more than one possible instance declaration which could be used). But if we say that the type of `e` is *dependent* on the type of `ce`, it is no longer ambiguous. CHR's subsume the expressiveness of functional dependencies. The remaining examples are not possible with only functional dependencies, however they are supported by CHR's.

Anti-symmetrical Relations The `Collects` class allows us to declare that `[a]` is a collection of `a`'s by an instance declaration for `Collects a [a]`. However, it would never be sensible to declare an instance for the reverse, i.e. `Collects [a] a`. CHR's can specify that the relationship is anti-symmetrical, making the latter instance declaration a type error.

Irreflexive Relations The `Collects` relation is also irreflexive, it wouldn't be sensible to allow a type to be a collection type for itself, i.e. `Collects τ_1 τ_2` where τ_1 and τ_2 are unifiable.

Many other relationships, such as transitivity or symmetry, can also be expressed with CHR's.

2.3 Constructor Classes

Type constructors, such as `List` (actually, `[]`) or `->`, are functions over types. Implicitly each type constructor has a *Kind* which specifies the number of argument types required to produce the result type.

Constructor classes, which are supported by Haskell 98, allow the programmer to write a class over type constructors. Then, any type constructor with the correct kind can be made an instance of the class, e.g. the definition of the `Functor` class in Haskell 98 is:

```

class Functor f where fmap :: (a -> b) -> f a -> f b

```

Instances can be provided for `[]` and `Tree` but not for `Int` or `->`, since they don't have the correct kind.

Instead of introducing a kind system to Haskell, we could use CHR's to express these constraints as multi-parameter class constraints. The CHR system can explicitly describe the required constraints, i.e. the type constructor is functional, surjective and correctly kinded.

3 Constraint Handling Rules

Constraint handling rules [Fru95] (CHR's) are a multi-headed concurrent constraint language for writing incremental constraint solvers. In effect, they define transitions from one constraint set to an equivalent constraint set. Transitions serve to simplify constraints and detect satisfiability and unsatisfiability. Efficient implementations of CHR's are available in the languages SICStus Prolog and Eclipse Prolog, and other implementations are currently being developed (e.g., for Java).

CHR's manipulate a constraint set in two parts: a global constraint in the language of the underlying solver, and a global set of primitive constraints defined only by constraint handling rules.

Constraint handling rules (*CHR rules*) are of two forms (though the first is sufficient)

$$\begin{array}{ll}
 \textit{simplification} & c_1, \dots, c_n \iff g \mid d_1, \dots, d_m \\
 \textit{propagation} & c_1, \dots, c_n \implies g \mid d_1, \dots, d_m
 \end{array}$$

In these rules c_1, \dots, c_n are CHR constraints, g is a Herbrand constraint, and d_1, \dots, d_m are either CHR or Herbrand constraints. The guard part g is optional. When it is omitted it is equivalent to $g \equiv \text{true}$. The simplification rule states that given a constraint set $\{c_1, \dots, c_n\}$ where g must hold, this set can be replaced by $\{d_1, \dots, d_m\}$. The propagation rule states that given a constraint set $\{c_1, \dots, c_n\}$ where g must hold, we should add $\{d_1, \dots, d_m\}$. A *CHR program* is a set of CHR rules.

More formally the logical interpretation of the rules is as follows. Let \bar{x} be the variables occurring in $\{c_1, \dots, c_n\}$, and \bar{y} (resp. \bar{z}) be the other variables occurring in the guard g (resp. rhs d_1, \dots, d_m) of the rule. We assume no local variables appear in both the guard and the rhs. The logical reading is

$$\begin{array}{ll} \text{simplification} & \forall \bar{x} (\exists \bar{y} \ g) \rightarrow (c_1 \wedge \dots \wedge c_n \leftrightarrow (\exists \bar{z} \ d_1 \wedge \dots \wedge d_m)) \\ \text{propagation} & \forall \bar{x} (\exists \bar{y} \ g) \rightarrow (c_1 \wedge \dots \wedge c_n \rightarrow (\exists \bar{z} \ d_1 \wedge \dots \wedge d_m)) \end{array}$$

The operational semantics is a transition system on a triple $\langle f, s, h \rangle_v$ of a conjunction of CHR and Herbrand constraints f , a conjunction of CHR constraints s and a conjunction of Herbrand constraints h , and a sequence of variables v . The logical reading of $\langle f, s, h \rangle_v$ is as $\exists \bar{y} f \wedge s \wedge h$ where y are the variables in the tuple not in v . Since the variable component v never changes we omit it for much of the presentation.

solve	$\langle d \wedge f, s, h \rangle \rightarrow \langle f, s, h' \rangle$	d is a Herbrand constraint, $\models h' \leftrightarrow h \wedge d$
introduce	$\langle d \wedge f, s, h \rangle \rightarrow \langle f, s \wedge d, h \rangle$	d is a CHR constraint
simplify	$\langle f, c' \wedge s, h \rangle \rightarrow_P \langle d \wedge f, s, h \wedge c = c' \rangle$	$c \iff g \mid d$ in P and $\models h \rightarrow \exists \bar{x} (c = c' \wedge g)$
propagate	$\langle f, c' \wedge s, h \rangle \rightarrow_P \langle d \wedge f, c' \wedge s, h \wedge c = c' \rangle$	$c \implies g \mid d$ in P and $\models h \rightarrow \exists \bar{x} (c = c' \wedge g)$

where \bar{x} are variables (assumed to be new) appearing in the CHR used. Note that the components of the triple are treated as conjunctions and the matching is modulo the idempotence, commutativity, and associativity of conjunction.

An important property of CHR programs is *confluence*. Confluence implies that the order of the transitions doesn't affect the final result. Two states $\langle f_1, s_1, h_1 \rangle_v$ and $\langle f_2, s_2, h_2 \rangle_v$ are *joinable* if there exists derivations $\langle f_1, s_1, h_1 \rangle_v \rightarrow_P^* \langle f_3, s_3, h_3 \rangle_v$ and $\langle f_2, s_2, h_2 \rangle_v \rightarrow_P^* \langle f_4, s_4, h_4 \rangle_v$ such that $\langle f_3, s_3, h_3 \rangle_v$ is a variant of $\langle f_4, s_4, h_4 \rangle_v$. Confluent CHR programs are guaranteed to be *consistent* (in the usual sense of a theory).

A CHR program P is confluent iff for each state $\langle f, s, h \rangle_v$, if $\langle f, s, h \rangle_v \rightarrow_P^* \langle f_1, s_1, h_1 \rangle_v$ and $\langle f, s, h \rangle_v \rightarrow_P^* \langle f_2, s_2, h_2 \rangle_v$ then $\langle f_1, s_1, h_1 \rangle_v$ and $\langle f_2, s_2, h_2 \rangle_v$ are joinable.

Importantly, for terminating CHR programs, confluence is *decidable* [AFM99] (although termination is not decidable). This is because for these programs, confluence is equivalent to local confluence which we can test by examining each *critical pair* of the program and seeing whether they are joinable. A critical pair of two rules

$$c_1 \wedge c'_1 \iff g_1 \mid d_1 \quad c_2 \wedge c'_2 \iff g_2 \mid d_2$$

is the pair of states $\langle c_1 \wedge d_2, \text{true}, g_1 \wedge g_2 \wedge c'_1 = c'_2 \rangle$ and $\langle c_2 \wedge d_1, \text{true}, g_1 \wedge g_2 \wedge c'_1 = c'_2 \rangle$ where $g_1 \wedge g_2 \wedge c'_1 = c'_2$ is satisfiable.

Deciding confluence requires that the CHR program is terminating. There are a number of syntactic restrictions on Haskell class and instance declarations that will assure us that the resulting CHR programs are terminating. There are also a number of other approaches to proving termination of CHR programs [Fru98].

4 Type Inference

Haskell is an implicitly typed language. The task of type inference is to infer a type for a given program or report error if the program is not typable. We identify the following three issues:

1. Class and instance declarations must be correct.
2. Type inference must generate a correct set of constraints which represent the possible solutions to the typing problem. The program is typable if the constraint problem is solvable.
3. Simplification of constraints is important for two reasons. Syntactically, it allows us to present type class constraints to the programmer in a more readable form. Operationally, simplification allows us to put type class constraints into a more efficient form. Type class constraints are translated into dictionaries. Hence, simplifying type class constraints may allow a more efficient translation. This form of simplification is known as context reduction in Haskell.

Type inference starts by processing all class and instance declarations, i.e. we translate all class and instance declarations into a CHR program. Then, type inference generates the constraints of the Haskell program and applies the CHR solving process. In CHR, a constraint C is solvable if the derivation from C does not lead to a constraint including *False*. Simplification may be invoked if necessary. The following three sections expand on the three issues.

4.1 Class and instance declarations

In this section we show how to translate class and instance definitions into CHRs.

Class definitions

A class definition

$$\text{class } (d_1, \dots, d_m) \Rightarrow C x_1 \dots x_n \text{ where } \dots$$

constrains any instance of the class C to also satisfy the class constraints d_1, \dots, d_m . Hence the corresponding CHR is

$$C x_1 \dots x_n \Longrightarrow d_1, \dots, d_m$$

Example 1. Consider the standard prelude definitions of `Ord` and its translation:

$$\text{class Eq t => Ord t where } \dots \quad \text{Ord t} \Longrightarrow \text{Eq t} \quad (S1)$$

Whenever we assert the $\text{Ord } t$ constraint we must also satisfy the $\text{Eq } t$ constraint. \diamond

Instance definitions

An instance definition

$$\text{instance } (d_1, \dots, d_m) \Rightarrow C t_1 \dots t_n \text{ where } \dots$$

maintains that tuple $t_1 \dots t_n$ is an instance of C if the constraints d_1, \dots, d_m are also satisfied. This corresponds to a simplification rule.

$$C t_1 \dots t_n \Longleftrightarrow d_1, \dots, d_m$$

Example 2. The instances of Ord and Eq for Lists and their translations are:

$$\begin{aligned} \text{instance Eq t => Eq [t] where } \dots \quad \text{Eq [t]} &\Longleftrightarrow \text{Eq t} & (S2) \\ \text{instance Ord t => Ord [t] where } \dots \quad \text{Ord [t]} &\Longleftrightarrow \text{Ord t} & (S3) \end{aligned}$$

This means that we can prove that a type $[t]$ is an instance of the class Eq or Ord if and only if we can prove that t is an instance. \diamond

Checking instance definitions

An instance declaration must be compatible with the class definition. An instance declaration is correct in this sense if the resulting CHR program is confluent.

Example 3. If the instance declaration for `Ord [t]` didn't require that `t` was also in class `Ord`, i.e.

$$\text{instance Ord [t] where ... Ord [t] } \iff \text{True} \quad (S4)$$

We would have a non-confluent CHR program, since `Ord [t]` has another derivation:

$$\text{Ord [t]} \mapsto_{S1} \text{Ord [t]} \wedge \text{Eq [t]} \mapsto_{S4} \text{Eq [t]} \mapsto_{S2} \text{Eq t}$$

◇

After generating a CHR program we can (assuming that it's terminating) check that it is confluent. If it isn't confluent then there is an error in the instance declarations, otherwise we can safely use the CHR program for type inference.

4.2 Solving Type Class Constraints

With this view of type classes simply as constraints defined by CHR rules, the solving process is obvious: generate the constraints of the program text and apply the CHR solving process.

Example 4. Consider the Haskell function

$$\text{f g h = c where a = tail g; b = init h; c = a < b}$$

the constraints generated are

$$\begin{aligned} t_a = [t_1] \wedge t_g = [t_1] \wedge t_b = [t_2] \wedge t_h = [t_2] \wedge t_f = t_g \mapsto t_h \mapsto t_c \wedge \\ \text{Ord } t_3 \wedge t_3 \mapsto t_3 \mapsto \text{Bool} = t_a \mapsto t_b \mapsto t_c \end{aligned} \quad (1)$$

Note that we use subscript notation to associate expressions and their inferred type. After simplification through unification we obtain

$$t_a = t_g = t_b = t_h = t_3 = [t_1] \wedge t_2 = t_1 \wedge t_f = [t_1] \mapsto [t_1] \mapsto \text{Bool} \wedge \text{Ord } [t_1] \quad (2)$$

If we now apply the constraint handling rules above to the constraint `Ord [t1]` we obtain the following derivation:

$$\begin{aligned} \text{Ord [t}_4] \iff \text{Ord t}_4 \quad (S3) & \mapsto \langle \text{Ord [t}_1], \text{true}, \text{true} \rangle_{\{t_1\}} \\ & \mapsto \langle \text{true}, \text{Ord [t}_1], \text{true} \rangle_{\{t_1\}} \\ & \mapsto \langle t_1 = t_4 \wedge \text{Ord t}_4, \text{Ord [t}_1], \text{true} \rangle_{\{t_1\}} \\ \text{Ord t}_4 \implies \text{Eq t}_5 \quad (S1) & \mapsto^* \langle \text{true}, \text{Ord t}_4, t_1 = t_4 \rangle_{\{t_1\}} \\ & \mapsto \langle t_5 = t_4 \wedge \text{Eq t}_5, \text{Ord t}_4, t_1 = t_4 \rangle_{\{t_1\}} \\ & \mapsto^* \langle \text{true}, \text{Eq t}_5 \wedge \text{Ord t}_4, t_1 = t_4 \wedge t_4 = t_5 \rangle_{\{t_1\}} \end{aligned}$$

Since such derivations are laborious, from now on we will use simplified derivations where the tuple is represented as a single conjunction and unification is applied to remove extra variables. The corresponding derivation is

$$\text{Ord [t}_1] \mapsto_{S3} \text{Ord t}_1 \mapsto_{S1} \text{Ord t}_1 \wedge \text{Eq t}_1$$

Since the CHR program is confluent, all alternative rewritings must give the same result, for example

$$\text{Ord [t}_1] \mapsto_{S1} \text{Ord [t}_1] \wedge \text{Eq [t}_1] \mapsto_{S2} \text{Ord [t}_1] \wedge \text{Eq t}_1 \mapsto_{S3} \text{Ord t}_1 \wedge \text{Eq t}_1$$

gives the same answer. ◇

4.3 Presenting Type Class Constraints

CHRs are also simplification rules, they replace a (type class) constraint by a simpler, equivalent form. When we want to present a type definition to the user we wish to have the “simplest” possible form. This is contrary to the usual solving methodology that adds redundant information to simplify the detection of unsatisfiability. To this end it is worth adding a separate simplification phase for presenting constrained types to users. This too can be represented by a (disjoint) CHR program.

The presentation rules for class definition

$$\text{class } (d_1, \dots, d_m) \Rightarrow C x_1 \dots x_n \text{ where } \dots$$

are of the form $C x_1 \dots x_n, d_i \iff C x_1 \dots x_n$ which removes the redundant constraints for presentation.

Example 5. An example class definition and its corresponding presentation rule are:

$$\text{class Eq } t \Rightarrow \text{Ord } t \text{ where } \dots \quad \text{Ord } t, \text{Eq } t \iff \text{Ord } t \quad \text{mlabelP1}$$

In presenting the answer to Example 4 we obtain the type $f :: \text{Ord } t \Rightarrow t \rightarrow t \rightarrow \text{Bool}$ since $\text{Ord } t_1, \text{Eq } t_1 \mapsto_{P1} \text{Ord } t_1$. \diamond

5 Properties of Type Class Systems

CHRs allow us to characterize under which conditions we retain feasible type inference and unambiguous programs.

5.1 Feasible Type Inference

Feasible type inference must be decidable and yield principal types. In CHR, type inference is decidable if the CHR program is terminating. If we restrict instance and class definitions to those allowed by the Haskell report and GHC’s multi-parameter type class extensions then the resulting CHR program is always terminating.

Principality means that for a given Haskell program the inferred type subsumes all other types we could possibly give to this program. CHRs preserve principal types because they only ever map constraints to logically equivalent constraints. Note that principal types are not syntactically unique (in Example 4, (1) and (2) are both possible principal types) but since the CHR program is confluent, we will always present the same type.

Confluence of the CHR program is a vital property. It guarantees that the CHR program is consistent, so we can meaningfully talk about unsatisfiable type constraints, and it guarantees that instance definitions satisfy class definitions.

Type Signatures

Type signatures allow the user to declare that variables have a certain type. They are an optional form of program documentation but necessary, for example, to retain decidability in the case of polymorphic recursion. In the presence of type signatures, we are moving from a type inference problem to a type reconstruction problem. This shift implies that our constraint solver also needs to handle entailment among constraints. Decidable constraint entailment is often difficult to establish. As in [Jon99], we find that the conditions of Haskell 98 are sufficient to guarantee that entailment is decidable.

5.2 Unambiguous Types

In the Haskell framework each function must have an unambiguous type. We can understand the notion of *unambiguity* in terms of CHRs, so that later when we extend the type system to use more complex CHRs we retain this property.

Suppose the inferred type for function f is $f :: D \Rightarrow \tau$ then the type for f is *unambiguous* for CHR program P if, for renaming to new variables ρ , $P \models D \wedge \rho(D) \wedge \tau = \rho(\tau) \rightarrow \alpha = \rho(\alpha)$ for each variable $\alpha \in \text{vars}(D \Rightarrow \tau)$.

We have a sound check for the unambiguity of a type using the CHR program P by seeing if $D \wedge \rho(D) \wedge \tau = \rho(\tau) \rightsquigarrow_P C$ where $\models C \rightarrow \alpha = \rho(\alpha)$. This check is complete for Haskell 98 programs, ignoring Haskell's Numeric defaulting mechanism. We conjecture that it is complete for other interesting Haskell extensions, such as Functional Dependencies.

6 Understanding Type Classes Extensions

There are a number of extensions to type classes that have been proposed in the literature. These can be understood in the uniform framework of CHRs. By unifying the representation of different extensions we can gain insight into what kinds of extensions are feasible.

Functional Dependencies

Jones [Jon00] proposes an extension of multi-parameter type classes to include functional dependencies among class arguments. From a CHR point of view, functional dependencies among variables in a type class just extend the proof requirements for an instance. They are expressible straightforwardly using CHRs.

A class definition with functional dependencies has the form

$$\text{class } (d_1, \dots, d_m) \Rightarrow C x_1 \dots x_n \mid fd_1, \dots, fd_k \text{ where } \dots$$

where fd_i is a *functional dependency* of the form $(x_{i_1}, \dots, x_{i_k}) \rightsquigarrow x_{i_0}$. The functional dependency asserts that given fixed values of x_{i_1}, \dots, x_{i_k} then there is only one value of x_{i_0} for which the class constraint $C x_1 \dots x_n$ can hold.

The CHR translation creates a propagation rule for each functional dependency of the form

$$C x_1 \dots x_n, C y_1 \dots y_n \Longrightarrow x_{i_1} = y_{i_1} \wedge \dots \wedge x_{i_k} = y_{i_k} \mid x_{i_0} = y_{i_0}$$

This CHR enforces the functional dependency.

Example 6. Returning to the collection class example, but now adding a functional dependency. We have the following rule and (simplified) CHR:

$$\text{class Collects } e \ ce \mid ce \rightsquigarrow e \text{ where } \dots \quad \text{Collects } e \ ce, \text{Collects } f \ ce \Longrightarrow f = e \quad (T1)$$

Note now that the type for `empty` is unambiguous because

$$\text{Collects } e \ ce \wedge \text{Collects } e' \ ce' \wedge ce = ce' \rightsquigarrow_{T1} \text{Collects } e \ ce \wedge \text{Collects } e' \ ce' \wedge ce = ce' \wedge e = e'$$

◇

Example 7. The type checking/inference for

```
f x y c = insert x z where z = insert y c
```

where `insert :: Collects e ce => e -> ce -> ce` gives

$$\begin{aligned}
& \text{Collects } e \text{ ce} \wedge e \mapsto ce \mapsto ce = t_y \mapsto t_c \mapsto t_z \wedge \text{Collects } e' \text{ ce}' \wedge e' \mapsto ce' \mapsto ce' = t_x \mapsto t_z \mapsto r \\
\equiv & \text{Collects } e \text{ ce} \wedge \text{Collects } e' \text{ ce} \wedge t_c = t_z = ce' = r = ce \wedge t_y = e \wedge t_x = e' \\
\mapsto_{T1} & \text{Collects } e \text{ ce} \wedge \text{Collects } e' \text{ ce} \wedge t_c = t_z = ce' = r = ce \wedge t_y = e \wedge t_x = e' \wedge e = e' \\
\equiv & \text{Collects } e \text{ ce} \wedge t_c = t_z = ce' = r = ce \wedge t_x = t_y = e
\end{aligned}$$

The type inferred is `f :: Collects e ce => e -> e -> ce -> ce` as expected. \diamond

This view of functional dependencies as CHR's clarifies one of the questions that Jones poses in the end of [Jon00]. Given the declarations

$$\begin{aligned}
& \text{class } U \text{ a b} \mid a \rightsquigarrow b \text{ where...} \\
& \text{class } U \text{ a b} \Rightarrow V \text{ a b} \text{ where...}
\end{aligned}$$

in Jones' framework, from the constraints $U \text{ a b} \wedge V \text{ a c}$ it cannot be inferred that $b = c$. The CHR rules support the automatic inference of inherited functional dependencies Consider the following example:

$$U \text{ a b} \wedge V \text{ a c} \rightsquigarrow U \text{ a b} \wedge V \text{ a c} \wedge U \text{ a c} \rightsquigarrow U \text{ a c} \wedge V \text{ a c} \wedge b = c$$

Constructor Classes

Type constructors are simply a functional relation among types. We can understand them simply using CHR's, this is simply a matter of replacing constructor expressions $f \text{ e}$ (lets say $\equiv fe$) by explicit kind constraints $\text{Kind}^* \text{->}^* f \text{ e } fe$. The class constraints need to satisfy appropriate properties (functionality, surjectiveness) which can be expressed with CHR's, as well as the kind constraints. For example

$$\begin{aligned}
\text{functional} & \quad \text{Kind}^* \text{->}^* f \text{ e } fe, \text{Kind}^* \text{->}^* f \text{ e } fe' \implies fe = fe' \\
\text{surjective} & \quad \text{Kind}^* \text{->}^* f \text{ e } fe, \text{Kind}^* \text{->}^* f' \text{ e}' fe \implies f = f', e = e' \\
\text{kinding} & \quad \text{Kind}^* \text{->}^* f \text{ e } fe, \text{Kind}^* f \iff \text{False}
\end{aligned}$$

Clearly constructor classes can be expressed using CHR's, and hence we have a more uniform understanding of their meaning and use. The presentation of constructor class constraints to the user might be preferable with the usual notation, but this is simply a matter of presentation.

7 Further Extensions to Types Classes Using CHR's

Given we can use CHR's to specify existing type class extensions, an immediate question is what other new extensions can we express in terms of CHR's.

Disjointness of Type Classes

The example in Section 2 illustrates how it may be useful to have additional constraints on the instances of a class. With this disjointness we may be able to have a weaker definition of non-overlapping instances.

Example 8. A CHR expressing that the `Integral` and `Fractional` type classes are disjoint is simply.

$$\text{Integral } t, \text{Fractional } t \iff \text{False} \quad (IF)$$

If we translate the two instance declarations for *Dividable* from the motivation we obtain:

$$\begin{aligned} \text{Dividable } t &\iff \text{Integral } t \\ \text{Dividable } t &\iff \text{Fractional } t \end{aligned}$$

Clearly the resulting CHRs are not confluent, since there are two disjoint replacements for *Dividable t*. But we could weaken the simplification rules to

$$\begin{aligned} \text{Dividable } t, \text{Integral } t &\iff \text{Integral } t \\ \text{Dividable } t, \text{Fractional } t &\iff \text{Fractional } t \end{aligned}$$

which together with (*IF*) give a confluent system. Note that with this reading, we can remove a *Dividable t* constraint if we already know that *t* is in *Integral* or *Fractional* but we cannot simply replace the *Dividable t* with one of these constraints. \diamond

Another extension is to allow negative information about type classes.

Example 9. The intention of the `Num` class is to describe numeric types. We might insist that functional types are never numbers by adding the rule

$$\text{Num } (s \mapsto t) \iff \text{False}$$

Or we might form a `NotNum` class meant to indicate types which cannot be numbers, where functional types are in this class, expressed by the rules

$$\begin{aligned} \text{NotNum } t, \text{Num } t &\iff \text{False} \\ \text{NotNum } (s \mapsto t) &\iff \text{True} \end{aligned}$$

Then, in either case, the declaration `instance Num (a->b) where ...` will cause an error to be detected since the resulting CHR program is not confluent. \diamond

In general there are considerable problems supporting overlapping class instances. The key thing to understand is that *confluence* of the resulting CHRs gives the behaviour we expect. If the resulting CHR program is not confluent, then there is an error with the program's class and instance definitions. If the CHR program is confluent it doesn't mean there are not problems, but at least the correctness of types is not affected by the overlapping instances.

Example 10. Consider the program

```
instance A t => C Bool t where ...
instance B s => C s Int where ...
```

The question is what should happen for class constraint `C Bool Int`. If `Bool` is in `B` and `Int` is in `A`, then it is clear that the constraint holds, we must simply choose which instance's methods to use. If we allow overlapping class instances, this leaves the fundamental problem of how to make this choice. \diamond

Record Types

We can define an extensible record type class by using a set of types for labels, and two class constraints:

```
class Rec r l b where
  select :: r -> l -> b          -- access record r, b = r.l
  update :: r -> l -> b -> r    -- update record r so r.l = b
class Rec r2 l b => Ext r1 l b r2 where
  extend :: r1 -> l -> b -> r2
```

The *Rec* constraint constrains r to be a record type containing element labeled l of type b . The *Ext* constraint constrains r_2 to be a type obtained by extending r_1 with a new element labelled l of type b . There are some obvious rules we want to hold in order to enforce type correctness.

class-defn	$Ext\ r_1\ l\ b\ r_2 \implies Rec\ r_2\ l\ b$
functionality	$Rec\ r\ l\ b_1, Rec\ r\ l\ b_2 \implies b_1 = b_2$
false-extension	$Ext\ r_1\ l\ b_1\ r_2, Rec\ r_1\ l\ b_2 \iff False$
extension	$Ext\ r_1\ l_1\ b_1\ r_2, Rec\ r_2\ l_2\ b_2 \implies l_1 \neq l_2 \mid Rec\ r_1\ l_2\ b_2$

8 Conclusion

We have demonstrated that constraint handling rules are a useful tool in understanding type class systems. Several existing type class systems can be expressed in terms of CHR rules. Surprisingly, constructor classes and multi-parameter classes which have been considered to be orthogonal extensions are both expressible in terms of CHR. Other useful extensions such as disjoint classes can also naturally be expressed in CHR. Feasible type inference and unambiguity are important issues in the design of a type class system. CHRs allows us to characterize sufficient conditions under which we can retain both properties. We conclude that CHRs offer a natural way to study type class systems. In this work, the development was rather motivated by examples and intuition. We are currently working on a more formal treatment which we will report separately.

References

- [AFM99] S. Abdennadher, T. Fruehwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(3):133–166, 1999.
- [CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 170–191. ACM Press, June 1992.
- [Fru95] T. Fruehwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
- [Fru98] T. Fruehwirth. A declarative language for constraint systems: Theory and practice of constraint handling rules, 1998. Habilitation.
- [JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.
- [Jon93] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, pages 52–61. ACM Press, June 1993.
- [Jon99] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, September 1999.
- [Jon00] Mark P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Proceedings of the 9th European Symposium on Programming Languages and Systems, ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2000.
- [Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *ESOP'88 Programming Languages and Systems*, volume 300 of *Lecture Notes in Computer Science*, pages 131–141. Springer-Verlag, 1988.
- [PHA⁺99] SL Peyton Jones, RJM Hughes, L Augustsson, D Barton, B Boutel, W Burton, J Fasel, K Hammond, R Hinze, P Hudak, T Johnsson, MP Jones, J Launchbury, E Meijer, J Peterson, A Reid, C Runciman, and PL Wadler. Report on the programming language Haskell 98, February 1999. <http://www.haskell.org/definition/>.
- [WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.