Modelling Scoped Instances with Constraint Handling Rules

author info removed for reviewing

–
–

Abstract

Haskell's class system provides a programmer with a mechanism to implicitly pass parameters to a function. A class predicate over some type variable in the type signature of a function induces the obligation for the caller to implicitly pass an appropriate instance of the class to the function. The class system is programmed by providing class instances for concrete types, thus providing, for each class, a unique mapping from types to instances. This mapping is used whenever an instance for a class predicate over some type is required. Choosing which instance to pass is solely based on the instantiated type of the class predicate.

Although this mechanism has proved to be powerful enough for modelling overloading and a plethora of other programming language concepts, it is still limited in the sense that multiple instances for a type cannot exist at the same time. Usually one can program around this limitation by introducing dummy types, which act as a key to map to additional instances; but this indirect way of allowing extra instances clutters a program and still is bound to the finite number of types statically available in a program. The latter restriction makes it impossible to dynamically construct instances, which, for example, depend on runtime values.

In this paper we lift these restrictions by means of local instances. Local instances allow us to shadow existing instances by new ones and to construct instances inside functions, using function arguments. We provide a translation of class and instance definitions to Constraint Handling Rules, making explicit the notion of "scope of an instance" and its role in context reduction for instances. We deal with the ambiguity of choosing between instances by using a framework for heuristically choosing between otherwise overlapping instances.

1. Introduction

The Haskell class system, originally introduced by both Wadler and Blott (Wadler and Blott 1988) and Kaes (Kaes 1988), offers a powerful abstraction mechanism for dealing with overloading (ad-hoc polymorphism). The basic idea is to restrict a polymorphic parameter by specifying that some predicates have to be satisfied when the function is called:

$f :: Eq\ a \Rightarrow a \to a \to Int$
$f = \lambda x\ y \to \textbf{if}\ x == y\ \textbf{then}\ 3\ \textbf{else}\ 4$

In this example the type signature for f specifies that values of any type a can be passed as arguments, provided the predicate $Eq\ a$ can be satisfied. Such predicates are introduced by *class declaration*s, as in the following simplified version of Haskell's *Eq* class declaration:

class *Eq a* **where**
  $(==) :: a \to a \to Bool$

The presence of such a class predicate in a type requires the availability of a collection of functions and values (here a collection with just one element) which can only be used on a type a for which the class predicate holds. A class declaration alone is not sufficient: *instance declarations* specify for which types the predicate actually can be satisfied, simultaneously providing an implementation for the functions and values as a witness for this:

instance *Eq Int* **where**
  $x == y = primEqInt\ x\ y$

instance *Eq Char* **where**
  $x == y = primEqChar\ x\ y$

Here the equality functions for *Int* and *Char* are implemented by the primitives *primEqInt* and *primEqChar*. One may look at this as if the compiler turns such instance declarations into records (called dictionaries) containing the functions as fields, and thus an explicit version of the internal machinery reads:

data $EqD\ a = EqD\ \{eqEqD :: a \to a \to Bool\}$ -- class Eq
$eqDInt = EqD\ primEqInt$ -- Eq Int
$eqDChar = EqD\ primEqChar$ -- Eq Char
$f :: EqD\ a \to a \to a \to Int$
$f = \lambda dEq\ x\ y \to \textbf{if}\ (eqEqD\ dEq)\ x\ y\ \textbf{then}\ 3\ \textbf{else}\ 4$

These definitions are used by the class system whenever the use of a value requires a class predicate to be satisfied. For example, the expression $f\ 3\ 4$ requires $Eq\ Int$ to hold, satisfied by the instance declaration for *Eq Int*. Additionally, an evidence based translation implicitly passes a dictionary for each predicate occurring in the type of the function: $f\ 3\ 4$ translates to $f\ eqDInt\ 3\ 4$.

For the purpose of this paper, the key design choices of the type class mechanism are:

- The class system chooses which instance is used to satisfy a predicate, and consequently which dictionary to pass implicitly.
- The class system makes its choice based on the type(s) over which a class predicate ranges.
- The only way a programmer can steer the class system is by providing class and instance declarations, and explicit type signatures for let-bound identifiers.

These design choices have led to, often ingenious, type level programming and class system extensions to support this, most notably

multiparameter type classes and functional dependencies. However, the basic mechanism of type based choice of an instance remains the same. For some programming problems this can lead to contrived code. For example, suppose we need to compare two *Int*s modulo 2 and still want to use the class system:

```
instance Eq Int where
  x == y = primEqInt (x `mod` 2) (y `mod` 2)
```

How do we instruct the class system to use this instance declaration instead of the previous, default, one? We can't, at least not without modifying the class *Eq* and its instances, or modifying the encoding of the *Int* value passed to (==). We further explore alternate *Int* encodings by means of a wrapper data type. Although both instances satisfy the predicate *Eq Int*, we have to choose between two dictionaries, without a means of making the choice: we have overlapping instances. We can avoid this by providing enough additional type information to make a choice:

```
data Default   = Default
data Mod2      = Mod2
data Wrap l a = W l a
instance Eq a ⇒ Eq (Wrap Default a) where
  W _ x == W _ y = x == y
instance Eq (Wrap Mod2 Int) where
  W _ x == W _ y = (x `mod` 2) == (y `mod` 2)
f :: Eq a ⇒ a → a → Int
f = λx y → if x == y then 3 else 4
v1 = f (W Default (2 :: Int)) (W Default 4)
v2 = f (W Mod2    (2 :: Int)) (W Mod2    4)
```

We wrap the *Int* value in the *Wrap* data type together with a value *Mod2* of which the type determines to use instance *Eq* (*Wrap Mod2 Int*). If we use local instances instead, the above clutter is avoided:

```
f' p q = let instance Eq Int where
               x == y = primEqInt (x `mod` 2) (y `mod` 2)
         in f p q
```

Function f' invokes f within the context of an overriding instance for *Eq Int*. The static nesting structure of **let** expressions provides the additional information for the class system to choose between the default *Eq Int* instance at the global level, and the local '*mod*`2 based *Eq Int* instance.

Local instances also allow us to tackle the so-called configuration problem (Kiselyov and Shan 2004) more directly. Configuration means the parameterization of code with additional data, without this being visible in the form of additional parameters in the configured code. For our example this means that we want to test for equality '*mod*`*n* for arbitrary *n* instead of fixed '*mod*`2, and solved for our *Wrap* example by:

```
data ModN n = ModN n
instance Eq (Wrap (ModN Int) Int) where
  W (ModN n) x == W _ y = (x `mod` n) == (y `mod` n)
f' n = f (W (ModN n) (2 :: Int)) (W (ModN n) 4)
```

However, Kiselyov's solution (Kiselyov and Shan 2004) differs from ours because of their use of phantom types, that is, data types with type parameters without a corresponding runtime value. For our *Wrap* example we then have:

```
data Mod2
data Wrap' l a = W' a
instance Eq (Wrap' Mod2 Int) where
  W' x == W' y = (x `mod` 2) == (y `mod` 2)
v2 = f (W' 2 :: Wrap' Mod2 Int) (W' 4)
```

We get rid of the *l* typed field in *Wrap'*, **data** *Mod2* no longer needs a constructor, and we pass type information via a type annotation instead of a value. Taking this route further, one can hide much of this mechanism with clever type level programming. However, the drawback is that we no longer can use *ModN* to pass additional data, because we no longer pass any runtime data along with the type: the resulting solution has become purely type based. This is a closed world, and as a result we then cannot take (for example) a commandline parameter and turn it into a configuration for the program.

Our solution using a local instance is as follows:

```
f' n p q = let instance Eq Int where
               x == y = primEqInt (x `mod` n) (y `mod` n)
           in f p q
```

The bottomline is that additional information must be passed to a function, and to solve this we either:

- wrap normal parameters with extra type related information steering the choice of an instance, possibly also including additional runtime available data;
- use phantom types instead to only pass type related information ((Kiselyov and Shan 2004), not further explained here);
- override directly by means of a local instance.

A solution without local instances always becomes cluttered because of the indirect way an instance is chosen via a type, whereas local instances circumvent such an indirection. This then is our motivation for looking into local instances.

In the next Section 2 we discuss what is problematic about scoped instances, and what the contribution of this paper is. Section 3 discusses previous work upon which we base ours, in Section 4 we provide the prerequisites for the technical part in Sections 5, 6, and 7. We conclude in Section 8 with discussion, related work and future work.

2. Local instances

A *scope* is a program location, distinguishable on the basis of its lexical nesting position. A *scoped instance* is an instance for which we know its defining scope. We distinguish between *global scope* as the lexically outermost scope; *local scope* is defined as non-global scope. In Haskell a normal module level instance corresponds to a globally scoped instance, or just global instance.

2.1 The problem

Although the idea of scoped (or local) instances already exists for some time and is available in a limited form known as "implicit parameters" (Lewis et al. 2000), to our knowledge scoped instances are neither available nor described previously. We can see the following areas where the interaction between scoped instances, context reduction, and type inference is problematic:

- Principal typing.
- The location in the program where the need to prove a predicate arises.
- Instances requiring other instances as their context.
- Interaction with type signatures.

Before proceeding, we point out that all these areas share the same problem with instances: how to choose between them. Such a choice already is an issue for 'normal' overlapping instances. The main underlying idea of this paper is to take this choice away from the language definition because it cannot be made unambiguously anyway, and to put it in the hands of the programmer. Naturally, default situations still can be handled without programmer intervention, but when necessary the programmer can specify exactly what a language tries to do automatically, without burdening the programmer with known default situations.

We now further discuss the list of problems mentioned above.

Principal typing Wadler & Blott observed (Wadler and Blott 1988) that allowing the following program led to loss of principal typing:

```
f = let instance Eq Int
        instance Eq Char
    in (==)
```

In his type system he could derive either $f :: Int \to Int \to Bool$ or $f :: Char \to Char \to Bool$ because of the presence of the two instances, but not $f :: Eq\ a \Rightarrow a \to a \to Bool$. The instantiation of the type of $(==) :: Eq\ a \Rightarrow a \to a \to Bool$ combined with the satisfaction of the instantiated predicate $Eq\ a$ by either $Eq\ Int$ or $Eq\ Char$ leads to his possible derivations. However, (==) does not even use the corresponding dictionaries, so this interaction seems unnecessary and artificial. Part of the problem lies with a check for satisfiability when the type of a value is established, usually as part of a generalization step for **let** bindings. Such a check also conflicts with modularity because later additional instances may be defined, and therefore this check is usually omitted. Even for the following variant the two instances play no role and could well have been omitted:

```
f x y = let instance Eq Int
            instance Eq Char
        in x == y
```

However, the key observation is that as soon as a polymorphic type is instantiated with a more concrete type, and the necessity to satisfy a predicate over this type arises, such local instances play a role:

```
f x y = let instance Eq Int    -- now used!
        in x == y ∧ x == 3
```

In $x == y \wedge x == 3$ the need for $Eq\ Int$ arises within the scope of the local $Eq\ Int$ instance, and its satisfaction thus must take the presence of this instance into account.[1] In terms of a solution for scoped instances, this means that the obligation to prove a predicate like $Eq\ Int$, has to be annotated with the scope in which that happens. And, although we lose principal typing, our solution for dealing with scoped instances offers a mechanism to steer context reduction for the qualified part of an inferred type. Therefore, scoped instances, when combined with type signatures (more about this later), in practice offer a way to explicitly deal with the consequence of the loss of principal typing.

Location The above observation complicates matters, as demonstrated by the following example; y's type is not generalized over in function g because it still is free in g's enclosing environment:

```
f x = let g y = let instance Eq Int
                in x == y          -- (*)
      in g 3
```

[1] We ignore the fact that 3 gives rise to a *Num* predicate.

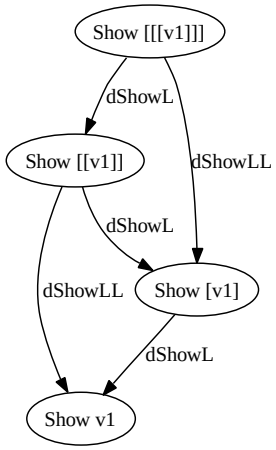

Figure 1. Context reduction for showTable

We only get to know the type of y together with f's parameter x, which means that context reduction for the Eq predicate over that type arising at (*) has to be done together with context reduction for f, on a more global level. There we find out that both y and x are of type Int, and that the nested $Eq\ Int$ instance can be used. Context reduction then still has to know about the local instance declaration, even for proofs on a more global level.

Instances with context The combination of instances requiring other instances in their context, overlapping instances, and local instances makes it rather difficult to predict what combination of instances is used to satisfy $Show\ [[[v_1]]]$, arising in *showTable*, in the following example for some type variable v_1:

```
class Show a where
  show :: a → [Char]
instance Show Char                          -- dShowChar
instance Show a ⇒ Show [a]                  -- dShowL
instance Show [Char]                        -- dShowCharL
showTable hdr tbl
   = let instance Show a ⇒ Show [[a]]       -- dShowLL
     in show (map (λx → x ++ x) hdr : tbl)
main = showTable ["Name","DOB"] [["G","19830511"]
                                ,["A","19830208"]]
```

What is the type of *showTable*, and along which path in Figure 1 do we reduce context, if we do context reduction at all?

```
showTable :: Show [[[a]]] ⇒ [[a]] → [[[a]]] → [Char]
showTable :: Show [a]     ⇒ [[a]] → [[[a]]] → [Char]
showTable :: Show a       ⇒ [[a]] → [[[a]]] → [Char]
```

The first type opts for no context reduction at all, whereas the second only uses the local instance *dShowLL*, and the third has several alternative paths available to arrive at $Show\ v_1$. From the above problems we conclude the following:

- The proving machinery, declared instances and predicates all need to be aware of (their) scope.
- The choice offered by local instances complicates making automatic choices during context reduction.

Type signatures The role of type signatures becomes more significant:

$$
\begin{aligned}
&e_1 :: \mathit{Int} \to \mathit{Bool}\\
&e_1\ x = x == 2\\
&e_2 :: \mathit{Eq\ Int} \Rightarrow \mathit{Int} \to \mathit{Bool}\\
&e_2\ x = x == 2\\
&f\ x = \textbf{let instance}\ \mathit{Eq\ Int}\\
&\qquad \textbf{in}\ \ e_1\ x \wedge e_2\ x
\end{aligned}
$$

Although currently not allowed (Peyton Jones 2003), the type signature for e_2 implies that the decision which *Eq Int* to use is delayed until e_2 is called: in f the local instance is used for e_2, the global one for e_1. The presence of a class predicate in the type signature for e_2 introduces a scoped instance for the body e_2. Without local instances this makes no difference, because only one *Eq Int* is present.

2.2 Our contribution

The awareness of scope and complexity of choice leads us to a solution which does not attempt to do the impossible, which is always automatically making the right choice between instances. Instead we aim for a design where ultimately the programmer has complete control over which instances are to be used. In this paper we therefore propose a three stage process for dealing with making choices for local instances separately:

- Translate class and instance declarations to an equivalent Constraint Handling Rule (CHR) representation, taking scope into account.
- Solve constraints using the CHR representation, thereby generating all possible solution alternatives for context reduction.
- Choose between solution alternatives by means of a separate framework.

We focus on the translation to CHRs and the choice between solution alternatives as we feel that this is the novelty of our approach. We do *not* deal with a type system in which our solution is to be embedded, nor make claims about formal properties of our solution, although we will point out what needs to be done in this area (Section 8). We emphasize that our solution should be seen as a proposed solution, not as the proven correct solution within a theoretical framework. We present part of our work in terms of Haskell, much in the spirit of "Typing Haskell in Haskell" (Jones 1999). A prototype system has been built and more extensively described by [redacted], and has been integrated into the [redacted] compiler [redacted].

We omit the description of a type system in which our context reduction takes place; although type inference and context reduction interact, these are largely separate issues. Our approach can be part of any standard Hindley-Milner type inference algorithm (Damas and Milner 1982) or be embedded in more recent type inference algorithms which are already CHR based (Sulzmann 1998; Stuckey and Sulzmann 2002).

In summary, our contribution is:

- Model scope for class predicates by means of CHR, and provide a different translation to CHRs as compared to Sulzmann et al. .
- Provide a framework for expressing different context reduction strategies, applied to scoped instances in particular. The intention is to make the use of this framework available to the programmer, but this has not been done yet.
- A prototype for the above, integrated in a Haskell compiler.

3. Related work

CHRs (Frühwirth 1998; Frühwirth and Abdennadher 2003) as used by Sulzmann et al. (Sulzmann 1998; Stuckey and Sulzmann 2002) to describe and formalize Haskell's type system. Our use of CHRs for the class system is slightly different because we postpone any decision making for instances to a later stage. We also have to be more precise in dealing with type signatures. As we focus on the class system exclusively, we ignore the use of CHRs for the underlying type system itself.

Techniques for improving type and class error messages are also constraint based (Heeren and Hage 2005; Heeren 2005; Heeren and IJzendoorn 2005). Their solution first solves constraints and then uses a representation of the solution to produce error messages; we use similar techniques to separately deal with choosing between instances.

Design alternatives for a class system, in particular context reduction, were explored by Peyton Jones et al. (Peyton Jones et al. 1997). Our framework for choices between instances allows to express such alternative context reduction strategies.

GHC (Marlow 2004) allows some form of programming the predicate proving machinery by means of pragmas, for example for choosing the most specific instance in case of overlapping instances.

Named instances (Kahl and Scheffczyk 2001; Scheffzcyk 2001) allow explicit named references to instances. This is a natural extension for being explicit in the choice between multiple instances, also experimented with in [redacted].

Closest to our work are modular type classes (Dreyer et al. 2007). Module signatures correspond to classes, modules to class instances: the class system is built on top of the module system. Their scoping mechanism is bound to an explicit import (*using*) mechanism for modules, similar to naming instances. Only explicitly imported instances partake in context reduction. Our work differs in that the notion of scope participates in choosing between otherwise overlapping instances; more information is taken into account, and we aim at programmability of the decision process between instances.

Furthermore, there is a large body of work on type classes (Jones 1993, 2000; Lewis et al. 2000; Chakravarty et al. 2005b,a), some of which is described in terms of CHRs (Sulzmann 1998; Stuckey and Sulzmann 2002; Stuckey et al. 2004; Sulzmann et al. 2006, 2007). Most of these are relatively independent of scoped instances.

4. Preliminaries

Terms, types and predicates Figure 2 shows the standard term and type language we assume for our discussion of scoped instances. We have included Figure 2 to clarify the minimal expected context of our work, but we will not explicitly use it any further after this section. Although the syntax allows local classes, we inhibit this. We see no reason to inhibit instances nested within instances. For simplicity we only consider single parameter type classes. Overbar notation $\bar{x}$ denotes (possibly empty) sequences of x's, comma denotes concatenation, i lowercase variables, and I uppercase constructors.

Constraints We use the following forms of constraint C, parameterized with predicate ϖ and *info* used to postprocess and choose between *Reduction* alternatives:

Value expressions:

e	$::= int$	literals
	$\mid i$	program variable
	$\mid e\ e$	application
	$\mid$ **let** $\overline{d}$ **in** e	local definitions
	$\mid \lambda i \rightarrow e$	abstraction

Declarations of bindings:

d_{sig}	$::= i :: \sigma$	value type signature
d_{val}	$::= i = e$	value binding
d	$::= d_{sig} \mid d_{val}$	
	$\mid$ **class** $\overline{\pi} \Rightarrow \pi$ **where** $\overline{d_{sig}}$	class
	$\mid$ **instance** $\overline{\pi} \Rightarrow \pi$ **where** $\overline{d_{val}}$	class instance

Types:

τ	$::= Int$	literals
	$\mid v$	variable
	$\mid \tau \rightarrow \tau$	abstraction
σ	$::= \forall \overline{v}.\overline{\pi} \Rightarrow \tau$	type scheme

Predicates:

π	$::= I\ \tau$	class predicate

Figure 2. Terms and types

$$\boxed{\Gamma; \overline{\pi} \Vdash_e \overline{\pi}}$$

$$\frac{\overline{\pi}_1 \supseteq \overline{\pi}_2}{\Gamma; \overline{\pi}_1 \Vdash_e \overline{\pi}_2}\ \text{MONO} \qquad \frac{\Gamma; \overline{\pi}_1 \Vdash_e \overline{\pi}_2 \quad \Gamma; \overline{\pi}_2 \Vdash_e \overline{\pi}_3}{\Gamma; \overline{\pi}_1 \Vdash_e \overline{\pi}_3}\ \text{TRANS}$$

$$\frac{\Gamma; \overline{\pi}_1 \Vdash_e \overline{\pi}_2}{\Gamma; S\ \overline{\pi}_1 \Vdash_e S\ \overline{\pi}_2}\ \text{CLOSED} \qquad \frac{\textbf{class}\ \overline{\pi}_2 \Rightarrow \pi_3 \in \Gamma \quad \Gamma; \overline{\pi}_1 \Vdash_e \pi_3}{\Gamma; \overline{\pi}_1 \Vdash_e \overline{\pi}_2}\ \text{CLASS}$$

$$\frac{\textbf{instance}\ \overline{\pi}_3 \Rightarrow \pi_2 \in \Gamma \quad \Gamma; \overline{\pi}_1 \Vdash_e \overline{\pi}_3}{\Gamma; \overline{\pi}_1 \Vdash_e \pi_2}\ \text{INST}$$

Figure 3. Entailment

$$\begin{array}{lll} C & ::= Prove & \varpi \\ & \mid Assume & \varpi \\ & \mid Reduction\ \varpi\ info\ \{\varpi_1, ..., \varpi_n\} & \end{array}$$

A *Prove* ϖ constraint indicates a proof obligation for ϖ, an *Assume* ϖ corresponds to an assumption for ϖ introduced by a type signature, and a *Reduction* corresponds to a reduction step from the ordered sequence $\{\varpi_1, ..., \varpi_n\}$ to ϖ. Depending on the problem at hand, ϖ instantiates differently. For our initial solution without scopes we instantiate ϖ by: $\varpi =_{def} \pi.\ \varpi(C)$ denotes the ϖ of a constraint C. The empty set of constraints represents *True*.

Constraints and entailment Entailment on predicates, shown in Figure 3, is standard (Jones 1994). Γ holds the environment of class and instance definitions. We allow ourselves notational looseness by allowing π to denote a singleton whenever $\overline{\pi}$ is expected.

Definition 1 (Constraint solution Θ). *A* constraint solution Θ *is a pair* (CHR, C^A)*, where CHR is a set of Constraint Handling Rules (defined later) and* C^A *is a set of Assume constraints.* $CHR^{i+c} =_{def} \Gamma$ *denotes the instance and class declarations which induce CHR,*

Constraint satisfaction is defined in terms of entailment:

Definition 2 (Constraint Satisfaction $\vdash_s$). *Let* Θ *be a constraint solution* (CHR, C^A)*. A* Prove *constraint is satisfied if the corresponding* ϖ *is entailed by* Θ*.*

$$\begin{array}{lll} \Theta \vdash_s Prove\ \ \varpi & =_{def} & CHR^{i+c}; \varpi(C^A) \Vdash_e \varpi \\ \Theta \vdash_s Assume\ \ \varpi & =_{def} & Assume\ \varpi \in C^A \\ \Theta \vdash_s Reduction\ \varpi\ info\ \{\varpi_1, ..., \varpi_n\} & =_{def} & True \end{array}$$

An Assume *is satisfied if the assumed* ϖ *is an element of* C^A*. A* Reduction *is trivially satisfied.*

Constraint Handling Rules A Constraint Handling Rule (CHR) declaratively specifies how to rewrite a constraint for a particular problem domain (Frühwirth 1998; Frühwirth and Abdennadher 2003). Each such CHR must be of a specific form:

$$\begin{array}{l} H_1, ..., H_i \iff G_1, ..., G_j \mid B_1, ..., B_k\ (simplification) \\ H_1, ..., H_i \implies G_1, ..., G_j \mid B_1, ..., B_k\ (propagation) \\ (i > 0, j \geqslant 0, k \geqslant 0) \end{array}$$

The set of constraints $(H_1, ..., H_i)$ is the *head* of a CHR, the set of conditions $(G_1, ..., G_j)$ form the *guard* of a CHR, and the constraints $(B_1, ..., B_k)$ form the *body* of a CHR.

Operationally, CHRs modify a working set of constraints: a simplification rule replaces the set of constraints matching the head by the constraints in the body when the conditions in the guard are satisfiable. A propagation rule adds instead of replaces the constraints in the body if the constraints in the head are present and the conditions in the guards are satisfiable. A propagation rule can be applied infinitely many times, but we avoid non-termination as a result of this by applying a propagation rule only once to every constraint in the constraint set. Duplicate constraints are considered redundant, that is, we use set semantics for sets of constraints.

CHRs can be given a declarative semantics (Frühwirth 1998; Frühwirth and Abdennadher 2003). A simplification is a logical equivalence if the guards are satisfied:

$$\begin{array}{l} \forall \overline{x}\ \forall \overline{y}\ ((G_1 \wedge ... \wedge G_j) \\ \qquad \rightarrow (H_1 \wedge ... \wedge H_i \leftrightarrow \exists \overline{z}\ (B_1 \wedge ... \wedge B_k))) \end{array}$$

Similarly, a propagation is an implication if the guards are satisfied:

$$\begin{array}{l} \forall \overline{x}\ \forall \overline{y}\ ((G_1 \wedge ... \wedge G_j) \\ \qquad \rightarrow (H_1 \wedge ... \wedge H_i \rightarrow \exists \overline{z}\ (B_1 \wedge ... \wedge B_k))) \end{array}$$

where $\overline{x} = fv(\overline{G})$, $\overline{y} = fv(\overline{H}) \backslash \overline{x}$ and $\overline{z} = fv(\overline{B}) \backslash (\overline{x} \cup \overline{y})$ are the free variables of the guard, head and body respectively. Although we omit the quantifiers, rules still should be read as if quantified as above.

We introduce scopes in Section 6, where they are required for the first time.

5. Mapping classes and instances to CHRs

In this section we discuss:

- How class and instance declarations are translated to CHRs.
- How constraints that arise from program text are simplified, and how contexts are reduced.

We start without taking scopes into account, showing the basic use of our constraint language, and pointing out where and why we differ from the 'standard' (Stuckey and Sulzmann 2002).

Simplication and context reduction Suppose we have the following class and instance declarations:

$$\begin{array}{ll}
\textbf{class } Eq\ a & \text{-- (C1)} \\
\textbf{class } Eq\ a \Rightarrow Ord\ a & \text{-- (C2)} \\
\textbf{class } Ord\ a \Rightarrow Real\ a & \text{-- (C3)} \\
\textbf{instance } Eq\ a \Rightarrow Eq\ [a] & \text{-- (I1)}
\end{array}$$

These declarations translate to the following set of CHRs, where $\varpi =_{def} \pi$ and π is a class predicate:

$$\begin{array}{lll}
Prove\ (Eq\ a)\ , Prove\ (Ord\ a) & \Longleftrightarrow Prove\ (Ord\ a) & \text{-- (C2)} \\
Prove\ (Ord\ a), Prove\ (Real\ a) & \Longleftrightarrow Prove\ (Real\ a) & \text{-- (C3)} \\
Prove\ (Eq\ a)\ , Prove\ (Real\ a) & \Longleftrightarrow Prove\ (Real\ a) & \text{-- (C23)} \\
Prove\ (Eq\ [a]) & \Longleftrightarrow Prove\ (Eq\ a) & \text{-- (I1)}
\end{array}$$

A possible derivation using these rules is:

$$\begin{array}{ll}
 & \{Prove\ (Eq\ [v_1]), Prove\ (Ord\ v_1), Prove\ (Real\ v_1)\} \\
\longrightarrow_{I_1} & \{Prove\ (Eq\ v_1)\ \ , Prove\ (Ord\ v_1), Prove\ (Real\ v_1)\} \\
\longrightarrow_{C_2} & \{Prove\ (Ord\ v_1)\ , Prove\ (Real\ v_1)\} \\
\longrightarrow_{C_3} & \{Prove\ (Real\ v_1)\}
\end{array}$$

All rules but C23 directly correspond to a class or instance declaration. In order to guarantee confluence (Frühwirth 1998; Frühwirth and Abdennadher 2003) it is necessary to explicitly represent the transitive closure of the class hierarchy by a set of rules. Were rule C23 is not available, the following derivation for the same initial constraint set is possible:

$$\begin{array}{ll}
 & \{Prove\ (Eq\ [v_1]), Prove\ (Ord\ v_1), Prove\ (Real\ v_1)\} \\
\longrightarrow_{C_3} & \{Prove\ (Eq\ [v_1]), Prove\ (Real\ v_1)\} \\
\longrightarrow_{I_1} & \{Prove\ (Eq\ v_1)\ \ , Prove\ (Real\ v_1)\}
\end{array}$$

This derivation follows a different path of applied CHRs but does not lead to the same final constraint set: without rule C23 the CHRs are not confluent and we cannot conclude with $Prove\ (Real\ v_1)$.

Interaction with type inference The constraint set $\{Prove\ (Eq\ [v_1]), Prove\ (Ord\ v_1), Prove\ (Real\ v_1)\}$ may have arisen from:

$$\begin{array}{ll}
f\ (x : xs) = xs == [\,] & \text{-- Prove(Eq [v1])} \\
\quad \wedge\ x\ \ < \ 1 & \text{-- Prove(Ord v1)} \\
\quad \wedge\ const\ True\ (toRational\ x) & \text{-- Prove(Real v1)}
\end{array}$$

Constraint solving is done for f's binding group, after all information about v_1 is reconstructed, and then leaves us with an unsatisfied $Prove\ (Real\ v_1)$. We only can satisfy this constraint if we have a corresponding assumption $Assume\ (Real\ v_1)$ using the following rule:

$$Prove\ p, Assume\ p \Longleftrightarrow Assume\ p \qquad \text{-- (E)}$$

An $Assume\ (Real\ v_1)$ can be introduced in two ways:

- The type inferencer decides to generalize over class predicate $Real\ v_1$ and constructs the following type for f:

  $$f :: Real\ a \Rightarrow [a] \rightarrow Bool$$

  In this case unresolved *Prove* constraints lead to corresponding *Assume* constraints, and constraint solving can proceed with these *Assume* constraints.
- The programmer has provided the type signature:

  $$f :: Real\ a \Rightarrow [a] \rightarrow Bool$$

  In this case the type signature directly leads to a constraint $Assume\ (Real\ c_1)$ for some fixed type variable c_1, and the body yields $\{Prove\ (Eq\ [c_1]), Prove\ (Ord\ c_1), Prove\ (Real\ c_1)\}$.

Here we differ from the 'standard' CHR (Stuckey and Sulzmann 2002) way of dealing with type signatures by encoding their entailment check as rule E combined with a richer constraint language which includes an *Assume*. We also expect the type inferencer to be capable of using a type signature in such a way that fixed type variables introduced by the signature end up in *Prove* constraints via their binding from program variables; this is usually accomplished by a combination of type inference and type checking (Peyton Jones and Shields 2004; Vytiniotis et al. 2006; Dijkstra 2005). Such explicit assumptions will come to good use when dealing with scopes and alternate entailment solutions.

Type signatures Suppose we have a simpler definition of f:

$$f\ (x : xs) = xs == [\,] \quad \text{-- Prove(Eq [v1])}$$

for which we infer:

$$f :: Eq\ a \Rightarrow [a] \rightarrow Bool$$

However, providing a type signature $f :: Real\ a \Rightarrow [a] \rightarrow Bool$ may leave us with unresolved constraints, unless we add the class hierarchy induced rules for *Assume* constraints as well. Otherwise we get stuck at $\{Prove\ (Eq\ c_1), Assume\ (Real\ c_1)\}$ because there are no rules to simplify to $\{Prove\ (Eq\ c_1), Assume\ (Eq\ c_1)\}$. Once again, by adding propagation rules for *Assume* constraints corresponding to the class hierarchy:

$$\begin{array}{lll}
Assume\ (Ord\ a) & \Longrightarrow Assume\ (Eq\ a) & \text{-- (CA2)} \\
Assume\ (Real\ a) & \Longrightarrow Assume\ (Ord\ a) & \text{-- (CA3)}
\end{array}$$

the derivation for $\{Prove\ (Eq\ c_1), Assume\ (Real\ c_1)\}$ can proceed:

$$\begin{array}{ll}
 & \{Prove\ (Eq\ c_1), Assume\ (Real\ c_1)\} \\
\longrightarrow_{CA3} & \{Prove\ (Eq\ c_1), Assume\ (Real\ c_1), Assume\ (Ord\ c_1)\} \\
\longrightarrow_{CA2} & \{Prove\ (Eq\ c_1), Assume\ (Real\ c_1) \\
 & \ , Assume\ (Ord\ c_1), Assume\ (Eq\ c_1)\} \\
\longrightarrow_{E} & \{Assume\ (Real\ c_1), Assume\ (Ord\ c_1), Assume\ (Eq\ c_1)\}
\end{array}$$

Overlapping instances: simplification versus propagation rules Until now we have used simplification rules for encoding instance declarations:

$$Prove\ (Eq\ [a]) \Longleftrightarrow Prove\ (Eq\ a) \qquad \text{-- (I1)}$$

Operationally this means that a CHR solver may replace the prove requirement for $Prove\ (Eq\ [a])$ with a prove requirement for $Prove\ (Eq\ a)$. However, in the presence of overlapping instances we loose confluence:

$$\begin{array}{ll}
\textbf{instance } Eq\ a \Rightarrow Eq\ [a] & \text{-- (I1)} \\
\textbf{instance } Eq\ [Int] & \text{-- (I2)}
\end{array}$$

From (I2) we now also have the rule:

$$Prove\ (Eq\ [Int]) \Longleftrightarrow True \qquad \text{-- (I2)}$$

Solving $Prove\ (Eq\ [Int])$ either leads to $Prove\ (Eq\ Int)$ via (I1) or to *True* via (I2). A solution to this ambiguity may be to add guards (Stuckey and Sulzmann 2002):

$$\begin{array}{lll}
Prove\ (Eq\ [a]) & \Longleftrightarrow a \not\equiv Int \mid Prove\ (Eq\ a) & \text{-- (I1)} \\
Prove\ (Eq\ [Int]) & \Longleftrightarrow True & \text{-- (I2)}
\end{array}$$

However, although this indeed inhibits rule (I1) where appropriate, it requires all instances to be known, checked for their specificness relative to each other, and guards to be generated accordingly. This conflicts with separate compilation and a module structure – GHC therefore inhibits context reduction using instances unless enforced by a type signature – and it is unclear how it interacts with more complex type-class extensions. Furthermore, a simplification rule

as above enforces a choice between instances; this conflicts with our intention to make such choices visible, so we can deal with those choices separately.

For these reasons we will avoid simplification rules and use propagation rules instead. A direct consequence is that we have confluence because no constraint is ever removed from the working set of a solver, and every matching propagation rule is used once for matching constraints.

However, the final constraint set yielded by a solver now contains all possible solutions, because we never get rid of constraints anymore. We have pushed the decision making problem ahead, and this is where our framework for heuristically making choices comes into play (Section 7).

A more subtle consequence of this design choice is that the interaction with a constraint solver becomes slightly different: entailment does no longer follow from the absence of *Prove* constraints in the final set of solved constraints, but non-entailment follows from a *Prove* constraint *not* being propagated from by any rule. We come back to this later (Section 7).

Making choice explicit: derivation tracing The given rules describe how to solve constraints, but leave no trace of how this is done, do not indicate solution alternatives, and do not give enough information when proceeding with code generation. We therefore let each rule also generate a *Reduction* constraint, thus encoding a derivation trace. Note that such traces are also generated for derivations which lead to unsatisfiable constraint sets; choosing between alternate reductions later will prune away such traces, or pick the best of these, depending on the strategy for making such choices. Consider again the following instance declarations:

```
instance Eq a ⇒ Eq [a]     -- (I1, dEqList)
instance Eq [Int]          -- (I2, dEqListInt)
```

These instances lead to the following propagation rules:

```
Prove (Eq [a])
   ⟹ Prove (Eq a)                                  -- (I1)
    , Reduction (Eq [a]) "dEqList"{Eq a}
Prove (Eq [Int])
   ⟹ Reduction (Eq [Int]) "dEqListInt"{}           -- (I2)
```

Solving *Prove* (*Eq* [*Int*]) now results in the following constraint set:

```
{Prove (Eq [Int])
,Reduction (Eq [Int]) "dEqList"{Eq Int}
,Prove (Eq Int)
,Reduction (Eq [Int]) "dEqListInt"{}}
```

The *info* field of a *Reduction* constraint holds additional (generated) information we need in subsequent stages. Here it holds the dictionary name. Later we introduce a more elaborate *Annotation* datatype for the *info* field of a *Reduction*.

We also need different rules for a class hierarchy. For example, for the following class:

```
class (Eq a, Show a) ⇒ Num a
```

we generate the following CHRs, where the *info* field of a *Reduction* now refers to the field of the dictionary holding the superclass dictionary:

```
Assume (Num a)
   ⟹ Assume (Eq a)
    , Reduction (Eq a)   "eqOfNum"   {Num a}
    , Assume (Show a)
    , Reduction (Show a) "showOfNum" {Num a}
Prove (Eq a), Prove (Num a)
   ⟹ Reduction (Eq a)   "eqOfNum"   {Num a}
Prove (Show a), Prove (Num a)
   ⟹ Reduction (Show a) "showOfNum" {Num a}
```

The reduction steps now tell us whether a class predicate π is entailed; in the above example *Eq Int* is not entailed because there is no *Reduction* constraint for *Eq Int*. Before proceeding with the construction of a reduction graph to be used in choosing between alternate reduction paths in Section 7, we first turn our attention to scoped instances and their CHR representation.

6. Mapping scoped instances to CHRs

Scoped instances allow us to write:

```
class Eq a where
  (==) :: a → a → Bool
instance Eq Int where                          -- (I1)
  x == y = primEqInt x y
e1 = 3 == 5                                    -- result: False
e2 = let instance Eq Int where                 -- (I2)
         x == y = primEqInt (x `mod` 2)
                            (y `mod` 2)
     in 3 == 5                                 -- result: True
```

A scope ξ is defined as a sequence of *Int*s, where longer sequences represent more deeply nested scopes, and later elements uniquely identify nested scopes within the enclosing scope, represented by the prefix of such an element:

```
type ξ = [Int]
```

Identifiers s, t are used to denote scopes ξ. Scope constants are denoted between < ... > to avoid confusion with other uses of [...] The common scope of two scopes is defined as their longest common prefix, which always exists because the global scope always is common:

```
commonScope :: ξ → ξ → ξ
commonScope (s : ss) (t : ts) | s == t      = s : commonScope ss ts
                              | otherwise   = [ ]
commonScope _        _                      = [ ]
```

A scope s is visible from a scope t if their common scope equals s:

```
visibleIn :: ξ → ξ → Bool
s `visibleIn` t = s == commonScope s t
```

Both CHRs and constraints reduced by those CHRs must be annotated with scope: constraints arisen in scope ξ can only be reduced by instances defined in ξ', where ξ' is visible from ξ. We define the scope ξ of a program location as follows:

- The global scope $\xi =_{def} <>$.
- The body of each **let** introduced binding has scope $\xi =_{def} \xi' +\!\!+ [n]$, where n is a unique *Int* value among all bindings in the surrounding scope ξ'.

The scope of both an instance and an induced constraint is defined to be the scope of the program location where the instance is defined or the constraint arises, respectively. A class predicate π in a constraint is annotated with this scope ξ, so $\varpi =_{def} (\pi, \xi)$, alternatively denoted by π^{ξ}. When the scope ξ is omitted from π^{ξ}, π ranges over any scope. For instance declarations we need to know their scope, so Γ from now on holds pairs (**instance** $\overline{\pi} \Rightarrow \pi, \xi$).

$$\boxed{\Gamma;\overline{\pi}^{\xi} \Vdash_e \overline{\pi}^{\xi}}$$

$$\frac{\textbf{class}\ \overline{\pi}_2 \Rightarrow \pi_3 \in \Gamma \quad \Gamma;\overline{\pi}_1 \Vdash_e \pi_3^s}{\Gamma;\overline{\pi}_1 \Vdash_e \overline{\pi}_2^s}\ \textsc{class} \qquad \frac{(\textbf{instance}\ \overline{\pi}_3 \Rightarrow \pi_2, t) \in \Gamma \quad \Gamma;\overline{\pi}_1 \Vdash_e \overline{\pi}_3^s \quad t\ \text{`}visibleIn\text{`}\ s}{\Gamma;\overline{\pi}_1 \Vdash_e \pi_2^s}\ \textsc{inst}$$

$$\frac{\Gamma;\overline{\pi}_1 \Vdash_e \pi_2^t \quad t\ \text{`}visibleIn\text{`}\ s}{\Gamma;\overline{\pi}_1 \Vdash_e \pi_2^s}\ \textsc{scope}$$

Figure 4. Entailment (scoped, declarative)

For example, the global scope in which (I1) is defined is ξ = <>, the scope in which (I2) is defined is ξ = <3>, where the 3 is arbitrary. The constraint *Prove* (*Eq Int*) for expression e_1 arises in scope <>, for expression e_2 it arises in scope <3>. For the latter both (I1) and (I2) can be used for context reduction. This normally implies an error because instances overlap, but by using scope information we can disambiguate.

The entailment relation is extended (in Figure 4) with a rule for making a global scoped predicate locally available. The rule for instances is restricted to apply to predicates of the same or more local scope only. The class rule still applies to any predicate since classes are only defined globally.

The scope rule is non-deterministic: it can be applied anywhere and reduce to any outer or equal scope. In principle, this does not matter because we postpone making choices anyway. However, in practice we introduce many intermediate reduction steps which clutter the reduction graph. Therefore we make the rules *CHR suitable* by modifying the rules such that every rule contributes a step in the right direction, that is strictly more global scope, fewer predicates to proof, or both:

- Rules for class and instances are as usual, but preserve scope.
- Scope reducing rules are for combinations of predicates of which scopes strictly differ, and lead to a simplification with a strictly more global scope.

In rule INST of Figure 5 we already automatically get scope reduction for instances without context. For instances with context, the scope is propagated to the context. For scope reductions the rule SCOPE now only reduces two predicates simultaneously which only differ in their scope. The rule CLASS1 is the same as the previous rule CLASS but is incomplete on its own because two predicates with different scope in a superclass relationship are not reduced. The previous version of rule SCOPE did 'spontaneously' reduce predicates to the required scope for rule CLASS1, but now we require additional rules CLASS2 and CLASS3 to take care of this combination. The overbar notation in these rules should be read as a rule for each predicate related by the superclass relation.

We have not proven soundness and completeness of the rules in Figure 5 relative to Figure 4. The intuition is that both sets of rules in Figure 4 independently reduce via instances (and classes) and via scopes. The rules in Figure 5 do this in a syntax directed manner by letting the set of the to be proven predicates determine which rule applies.

CHRs for instances We change the translation of instances to CHRs accordingly, for the declaration **instance** $(\pi_1,\ldots,\pi_n) \Rightarrow \pi$, in scope ξ we get:

$$\boxed{\Gamma;\overline{\pi}^{\xi} \Vdash_e \overline{\pi}^{\xi}}$$

$$\frac{\textbf{class}\ \overline{\pi}_2 \Rightarrow \pi_3 \in \Gamma \quad \Gamma;\overline{\pi}_1 \Vdash_e \pi_3^s}{\Gamma;\overline{\pi}_1 \Vdash_e \overline{\pi}_2^s}\ \textsc{class1} \qquad \frac{\textbf{class}\ \overline{\pi}_2 \Rightarrow \pi_3 \in \Gamma \quad \Gamma;\overline{\pi}_1 \Vdash_e \pi_3^{s'} \quad not\ (s\ \text{`}visibleIn\text{`}\ t) \quad t' \equiv s\ \text{`}commonScope\text{`}\ t}{\Gamma;\overline{\pi}_1 \Vdash_e \pi_3^s, \overline{\pi}_2^t}\ \textsc{class2}$$

$$\frac{\textbf{class}\ \overline{\pi}_2 \Rightarrow \pi_3 \in \Gamma \quad \Gamma;\overline{\pi}_1 \Vdash_e \pi_2^{s'} \quad not\ (t\ \text{`}visibleIn\text{`}\ s) \quad t' \equiv s\ \text{`}commonScope\text{`}\ t}{\Gamma;\overline{\pi}_1 \Vdash_e \pi_3^s, \overline{\pi}_2^t}\ \textsc{class3}$$

$$\frac{(\textbf{instance}\ \overline{\pi}_3 \Rightarrow \pi_2, t) \in \Gamma \quad \Gamma;\overline{\pi}_1 \Vdash_e \overline{\pi}_3^s \quad t\ \text{`}visibleIn\text{`}\ s}{\Gamma;\overline{\pi}_1 \Vdash_e \pi_2^s}\ \textsc{inst} \qquad \frac{\Gamma;\overline{\pi}_1 \Vdash_e \pi_2^t \quad t\ \text{`}visibleIn\text{`}\ s \quad s \not\equiv t}{\Gamma;\overline{\pi}_1 \Vdash_e \pi_2^s, \pi_2^t}\ \textsc{scope}$$

Figure 5. Entailment (scoped, CHR suitable)

$$\begin{array}{l} Prove\ (\pi, s) \\ \quad \Longrightarrow \xi\ \text{`}visibleIn\text{`}\ s \\ \quad\ \ |\ \ Reduction\ (\pi, s)\ (ByInstance\ name\ \xi)\ \{(\pi_1, s), \ldots, (\pi_n, s)\} \\ \quad\ \ ,\ \ Prove\ (\pi_1, s), \ldots, Prove\ (\pi_n, s) \end{array}$$

The *Reduction* constraint is annotated with both the name and the scope of the instance; we come back to *ByInstance* later when this annotation is used to choose between *Reduction* constraints. For the current example we have:

$$\begin{array}{l} Prove\ (Eq\ Int, s) \\ \quad \Longrightarrow <>\ \text{`}visibleIn\text{`}\ s \\ \quad\ \ |\ \ Reduction\ (Eq\ Int, s)\ (ByInstance\ \texttt{"I1"}<>)\ \{\} \\ Prove\ (Eq\ Int, s) \\ \quad \Longrightarrow <3>\ \text{`}visibleIn\text{`}\ s \\ \quad\ \ |\ \ Reduction\ (Eq\ Int, s)\ (ByInstance\ \texttt{"I2"}<3>)\ \{\} \end{array}$$

and the following derivation for *Prove* (*Eq Int*, <3>) arising at (e2):

$$\begin{array}{ll} & \{Prove\ (Eq\ Int, <3>)\} \\ \longrightarrow_{I_1} & \{Prove\ (Eq\ Int, <3>) \\ & ,Reduction\ (Eq\ Int, <3>)\ (ByInstance\ \texttt{"I1"}<>)\{\}\} \\ \longrightarrow_{I_2} & \{Prove\ (Eq\ Int, <3>) \\ & ,Reduction\ (Eq\ Int, <3>)\ (ByInstance\ \texttt{"I1"}<>)\{\} \\ & ,Reduction\ (Eq\ Int, <3>)\ (ByInstance\ \texttt{"I2"}<3>)\{\}\} \end{array}$$

where the *Reduction* constraints in the final constraint set correspond to the graph in Figure 6. We still have overlapping instances, but we are prepared for making the choice.

CHRs for classes For classes we generate rules for each pair (π_2, π_3), where π_2 is a (transitive) superclass of π_3. These CHRs correspond to rule CLASS2 and rule CLASS3, respectively:

$$\begin{array}{l} Prove\ (\pi_3, s), Prove\ (\pi_2, t) \\ \quad \Longrightarrow not\ (s\ \text{`}visibleIn\text{`}\ t) \\ \quad\ \ |\ \ Prove\ (\pi_3, commonScope\ s\ t) \\ \quad\ \ ,\ \ Reduction\ (\pi_3, s) \\ \qquad\qquad\qquad (ByScope\ (commonScope\ s\ t)) \\ \qquad\qquad\qquad \{(\pi_3, commonScope\ s\ t)\} \\ Prove\ (\pi_3, s), Prove\ (\pi_2, t) \end{array}$$

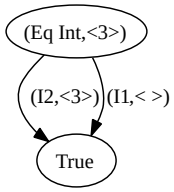

Figure 6. Context reduction for scoped Eq Int

$$
\begin{array}{l}
\Longrightarrow \mathit{not}\ (t\ \text{`}\mathit{visibleIn}\text{`}\ s) \\
\quad \mid \mathit{Prove}\ (\pi_2, \mathit{commonScope}\ s\ t) \\
\quad , \mathit{Reduction}\ (\pi_2, t) \\
\qquad\qquad (\mathit{ByScope}\ (\mathit{commonScope}\ s\ t)) \\
\qquad\qquad \{(\pi_2, \mathit{commonScope}\ s\ t)\}
\end{array}
$$

Rule CLASS1 differs from rule CLASS only in the scope, which is irrelevant for classes. Scope is simply propagated; we therefore omit further discussion.

CHRs for scoped predicate pairs The CHR counterpart of rule SCOPE also is straightforward:

$$
\begin{array}{l}
\mathit{Prove}\ (p, s), \mathit{Prove}\ (p, t) \\
\quad \Longrightarrow t\ \text{`}\mathit{visibleIn}\text{`}\ s, s \not\equiv t \\
\quad \mid \mathit{Prove}\ (p, t) \\
\quad , \mathit{Reduction}\ (p, s)\ (\mathit{ByScope}\ t)\ \{(p, t)\}
\end{array}
$$

Remaining CHRs We also require CHRs dealing with entailment relative to *Assume* constraints. These CHRs are variants of the CHRs for rules CLASS2, CLASS3, and SCOPE, where the more global *Prove* constraint has been replaced by an *Assume* constraint; here we have omitted these CHRs, see [redacted] instead.

7. Heuristics over the reduction graph

The CHR solver leaves us with a final set of constraints, for which:

- We extract all *Reduction* constraints.
- From these *Reduction* constraints we construct a reduction graph such as in Figure 6, and consider all possible alternative reductions.
- From alternative reductions we choose heuristically.
- From the chosen reduction we generate evidence, that is, a representation of the code required for constructing dictionaries.

We use the following running example, where Figure 7 shows the reduction graph for f:

$$
\begin{array}{ll}
\textbf{instance}\ \mathit{Eq}\ \mathit{Int} & \text{-- dEqInt} \\
\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a] & \text{-- dEqL} \\
\textbf{instance}\ (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow \mathit{Eq}\ (a, b) & \text{-- dEqTup} \\
\textbf{instance}\ \mathit{Ord}\ \mathit{Int} & \text{-- dOrdInt} \\
\textbf{instance}\ \mathit{Ord}\ a \Rightarrow \mathit{Ord}\ [a] & \text{-- dOrdL} \\
\textbf{instance}\ (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow \mathit{Ord}\ (a, b) & \text{-- dOrdTup}
\end{array}
$$

$$
\begin{array}{l}
f :: \mathit{Ord}\ a \Rightarrow (\mathit{Int}, a) \rightarrow [(\mathit{Int}, a)] \rightarrow \mathit{Bool} \\
f\ x\ (y : ys) = x == y \wedge x < y \wedge [x] == ys \wedge [y] < ys
\end{array}
$$

We present this sequence of steps in Haskell. We omit the glue for all code fragments, and only show the essentials; the implementation is fully described in [redacted].

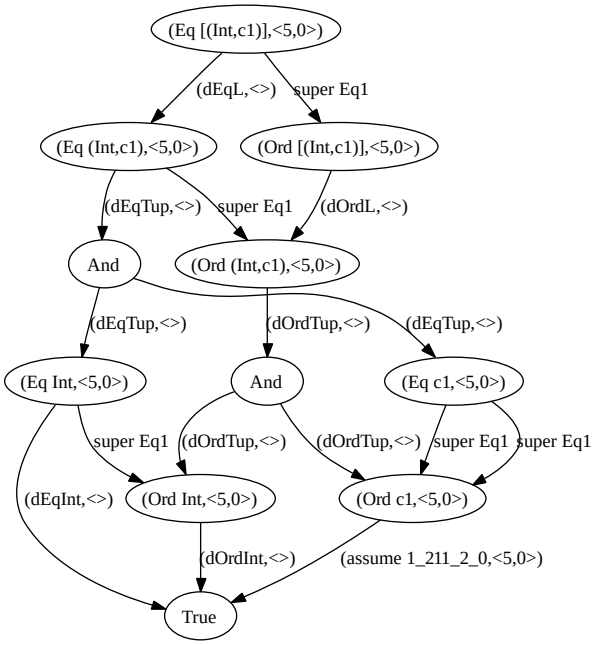

Figure 7. A reduction graph

Constraints From the CHR solver we get the final set of constraints, where constraints are represented as:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Constraint}\ \varpi\ \mathit{info} \\
\quad = \mathit{Prove}\ \ \varpi \\
\quad \mid \mathit{Assume}\ \ \varpi \\
\quad \mid \mathit{Reduction}\ \varpi\ \mathit{info}\ [\varpi]
\end{array}
$$

Again, with $\varpi =_{def} (\pi, \xi)$. The *info* parameter is defined to be:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Annotation} & & \\
\quad = \mathit{ByInstance}\ \mathit{String}\ \pi\ \xi & & \text{-- instance name, pred, scope} \\
\quad \mid \mathit{BySuper}\ \ \mathit{String} & & \text{-- superclass field name} \\
\quad \mid \mathit{ProveObl}\ \ \mathit{String} & & \text{-- evidence name} \\
\quad \mid \mathit{Assumption}\ \mathit{String} & & \text{-- assumed instance name} \\
\quad \mid \mathit{ByScope}\ \ \xi & & \text{-- scope reduction}
\end{array}
$$

In an *Annotation* we encode all the relevant information required to make choices between reduction alternatives and to generate evidence, hence these *Annotation*s are stored along with the edges in the reduction graph constructed below from *Reduction* constraints. Later we define various orderings on *Annotation*s to be used by heuristics to make a choice between the possible paths.

Reduction graph From these constraints we construct the reduction graph, for which we use the inductive graph library (Erwig 2001) to encode a graph (Annotated Graph, *AGraph*) for nodes a and edges b with the following interface:

$$
\begin{array}{l}
\mathit{emptyAGraph} :: \mathit{Ord}\ a \Rightarrow \mathit{AGraph}\ a\ b \\
\mathit{insertEdge}\ \ :: \mathit{Ord}\ a \Rightarrow\ (a, a, b)\ \ \rightarrow \mathit{AGraph}\ a\ b \rightarrow \mathit{AGraph}\ a\ b \\
\mathit{insertEdges} :: \mathit{Ord}\ a \Rightarrow [(a, a, b)] \rightarrow \mathit{AGraph}\ a\ b \rightarrow \mathit{AGraph}\ a\ b \\
\mathit{deleteEdge}\ \ :: \mathit{Ord}\ a \Rightarrow\ (a, a)\ \ \ \ \rightarrow \mathit{AGraph}\ a\ b \rightarrow \mathit{AGraph}\ a\ b \\
\mathit{successors}, \mathit{predecessors} :: \mathit{Ord}\ a \Rightarrow \mathit{AGraph}\ a\ b \rightarrow a \rightarrow [(b, a)]
\end{array}
$$

Nodes correspond to predicates $\varpi(C)$ from constraints C, and edges are reductions labeled with an *Annotation*. Predicates ϖ are wrapped in a *Node* so we can represent conjunction:

```
data Node ϖ = Pred ϖ
            | And [ϖ]

true :: Node ϖ
true = And [ ]
```

Disjunction is encoded by multiple outgoing edges in the graph structure. From the body of f we extract the following constraints: one *Assume* constraint from the type signature with scope <5, 0>, and four *Prove* constraints. All constraints are mapped to *Annotation*s holding the program variable by which we can refer to the assumed evidence or evidence of the proof obligations yet to be proven; it is the responsibility of the user of this framework to provide this information:

```
Prove   (Eq   [(Int, c1)], <5, 0>) ↦ {ProveObl   "p1"}
Prove   (Eq   (Int, c1)  , <5, 0>) ↦ {ProveObl   "p2"}
Prove   (Ord [(Int, c1)], <5, 0>) ↦ {ProveObl   "p3"}
Prove   (Ord (Int, c1)  , <5, 0>) ↦ {ProveObl   "p4"}
Assume (Ord c1          , <5, 0>) ↦ {Assumption "a1"}
```

Note that each constraint maps to multiple *Annotation*s since a constraint may arise many times. In our example however we only have one *Annotation* per constraint.

From these constraints, together with the class and instance CHRs, the CHR solver produces the following *Reduction* constraints which lead to the reduction graph in Figure 7. We show two typical elements of the full set of *Reduction* constraints in their full glory:

```
{Reduction (Ord (Int, c1), <5, 0>)
           (ByInstance "dOrdTup" (Ord (Int, c1))<5, 0>)
           {(Ord Int, <5, 0>), (Ord c1, <5, 0>)}
, Reduction (Eq   c1, <5, 0>)
           (BySuper    "eq0fOrd"                    )
           {(Ord c1, <5, 0>)}
, ...
}
```

Reduction alternatives From a reduction graph we compute reduction alternatives for each *Prove* π^{ξ} proof obligation. This is a straightforward tree representation of the part of the graph relevant for π^{ξ}:

```
data Alts ϖ info = Alts ϖ   [Red ϖ info]
data Red ϖ info = Red info [Alts ϖ info]

alternatives :: Ord ϖ ⇒ Graph ϖ info → ϖ → Alts ϖ info
alternatives gr = recOr
  where recOr   p      = Alts p (map recAnd
                                     (successors gr (Pred p)))
        recAnd (i, n)  = Red i  (map recOr (preds n))
        preds n = case n of
                    Pred q  → [q]
                    And  qs → qs
```

Choosing heuristically From these reduction alternatives we choose one based on a heuristic, and immediately generate evidence:

```
data Evidence ϖ info = Build ϖ info [Evidence ϖ info]
                     | Unresolved ϖ

unresolved :: Eq p ⇒ Evidence p info → [p]
unresolved (Unresolved p) = [p]
unresolved (Build _ _ ps)  = nub (concatMap unresolved ps)

type Heuristic ϖ info = [info] → Alts ϖ info
                      → [(info, Evidence ϖ info)]
type SimpleHeuristic ϖ info = Alts ϖ info → Evidence ϖ info

toHeuristic :: SimpleHeuristic p info → Heuristic p info
toHeuristic h infos alts = zip infos (repeat (h alts))
```

A *SimpleHeuristic* computes evidence for a single π^{ξ} from reduction alternatives, *toHeuristic* lifts this computation to a *Heuristic*, which binds the list of *Annotation*s bound to π^{ξ} to a single *Evidence*.

A *Heuristic* is our mechanism for steering the otherwise fixed strategy of choosing between reduction alternatives, thereby making explicit what usually is implicit. Although we present this mechanism in the form of hard-coded Haskell functions, one can easily envision the use of a domain specific language for the construction of heuristics instead.

A heuristic is steered by an ordering on *Annotation*s. For example, the following ordering prefers instance reductions over superclass reductions, superclass reductions over assumptions, and so on, like in Haskell98:

```
haskell98 :: Annotation → Annotation → Ordering
haskell98 (ByInstance _ _ _) _                = GT
haskell98 _                  (ByInstance _ _ _) = LT
haskell98 (BySuper     _) _                = GT
haskell98 _                  (BySuper     _) = LT
haskell98 (Assumption  _) _                = GT
haskell98 _                  (Assumption  _) = LT
...
h98Heuristic :: Heuristic ϖ Annotation
h98Heuristic = toHeuristic (binChoice haskell98)
```

For a Haskell98 heuristic it is sufficient to pairwise compare the *Annotation*s for reduction alternatives and make a local choice:

```
binChoice :: Eq info ⇒ (info → info → Ordering)
                     → SimpleHeuristic p info
binChoice order = localChoice (const local)
  where local [ ] = [ ]
        local is  = [maximumBy order is]

localChoice :: Eq info ⇒ (p → [info] → [info])
                       → SimpleHeuristic p info
localChoice choose (Alts p reds) =
  let redinfos = choose p (map info reds)
  in case filter ((∈ redinfos).info) reds of
       [ ]          → Unresolved p
       [(Red i rs)] → Build p i (map (localChoice choose) rs)
       _            → error "Alternatives left"
```

Function *binChoice* computes the preferred *Annotation* given an ordering on *Annotation*s; *localChoice* returns this choice unless there is no choice (unresolved constraint) or there is still ambiguity amongst reductions.

This is also the point where we get to know which constraints are unresolved; sooner is not possible because which constraints are unresolved depends on the choices we make between reductions. From this point onwards we also assume that neither unresolved nor ambiguous reductions occur in the reduction graph, either because they have been reported as an error, or because they have been replaced by assumptions. Ambiguous reductions occur when a heuristic can not make a unique choice, usually leading to "overlapping instance" errors. The latter is done by type inferencing as part of the generalization step. As a consequence, these functions

have to be used twice, first to detect unresolvedness and ambiguity, and subsequently to produce the actual evidence.

For the current example (Figure 7) the Haskell98 heuristic prefers reduction by *dEqL* for $(Eq\ [(Int, c_1)], <5, 0>)$, fully ignoring scope information. A *BySuper* reduction is chosen for $(Eq\ c_1, <5, 0>)$, for which no *ByInstance* alternative is available.

GHC heuristic GHC (Marlow 2004) uses a different context reduction strategy in order to support overlapping instances and arbitrary contexts in type signatures (Peyton Jones et al. 1997). Haskell98 eagerly reduces context as far as possible, whereas GHC delays context reduction using instances as long as possible. A predicate is only reduced when it can be resolved locally (tautological predicate) or when forced by a type signature. GHC first tries to reduce predicates using the following local heuristic:

```
ghcBinSolve :: Annotation → Annotation → Ordering
ghcBinSolve (Assumption _) _                    = GT
ghcBinSolve _              (Assumption _)       = LT
ghcBinSolve (BySuper _)    _                    = GT
ghcBinSolve _              (BySuper _)          = LT
ghcBinSolve (ByInstance _ _ _) _                = GT
ghcBinSolve _              (ByInstance _ _ _)   = LT
...

ghcSolve :: Eq p ⇒ SimpleHeuristic p Annotation
ghcSolve = binChoice ghcBinSolve
```

The most notable difference between the heuristics of Haskell98 and GHC is the way predicates are reduced using instance declarations: Haskell98 requires predicates to be reduced first with instance declarations, whereas GHC reduces predicates only with instance declarations if there is no other alternative left. This is possible because GHC allows arbitrary contexts in type signatures. Were GHC to first reduce using instances, possibly no assumptions introduced by type signatures would be found. GHC uses another heuristic when there are still *Unresolved* nodes in the solution found by the *ghcSolve* heuristic:

```
ghcLocalReduce :: a → [Annotation] → [Annotation]
ghcLocalReduce _ reds = let p (BySuper _) = True
                            p _           = False
                        in filter p reds

ghcReduce :: Eq p ⇒ SimpleHeuristic p Annotation
ghcReduce = localChoice ghcLocalReduce
```

This heuristic only reduces a predicate using the class hierarchy. Context reduction stops the class hierarchy does not enable reductions, even when there are other alternatives. We introduce the *try* combinator to combine both heuristics:

```
try :: Eq p ⇒ SimpleHeuristic p info → SimpleHeuristic p info
            → SimpleHeuristic p info
try f g a | null (unresolved e) = e
          | otherwise           = g a
          where e = f a

ghcHeuristic :: Eq p ⇒ Heuristic p Annotation
ghcHeuristic = toHeuristic (try ghcSolve ghcReduce)
```

Overlapping instance heuristic A heuristic that deals with overlapping instances by preferring the instance for a more specific class predicate. We assume function *specificness* which orders two class predicates accordingly. We adapt *ghcBinSolve* as an example:

```
specificness :: π → π → Ordering   -- p > q when more specific
specificness p q = ...             -- error when equal

ghcBinSolve :: Annotation → Annotation → Ordering
ghcBinSolve (Assumption _) _                    = GT
ghcBinSolve _              (Assumption _)       = LT
ghcBinSolve (BySuper _)    _                    = GT
ghcBinSolve _              (BySuper _)          = LT
ghcBinSolve (ByInstance _ p _) (ByInstance _ q _) = cmp
  where cmp = specificness p q
ghcBinSolve (ByInstance _ _ _) _                = GT
ghcBinSolve _              (ByInstance _ _ _)   = LT
...
```

Scoped instance heuristic Our last heuristic takes scope into account: instances with more deeply nested scopes are preferred.

```
ghcBinSolve (ByInstance _ _ s) (ByInstance _ _ t) = cmp
  where cmp = length s `compare` length t
ghcBinSolve (ByInstance _ _ _) _                  = GT
ghcBinSolve _                  (ByInstance _ _ _) = LT
...
```

With this heuristic we choose instance (I2) for $(Eq\ \ Int, <3>)$ in Figure 6.

8. Discussion, related and future work

Theoretical aspects The contribution of this paper is a proposal and an implementation, both of practical nature. Theoretical work has to be done for the following:

- Proof of soundness and completeness of the rules in Figure 5 relative to Figure 4.
- Embedding in a type system.
- Formalisation of the heuristics.

We expect the last of these to be the most difficult.

Explicitly dealing with otherwise implicit choices With our framework we now are able to be explicit in the choices made during context reduction, and thereby be free from the choices usually made deep inside the bowels of a compiler: context reduction is now programmable. Different varieties of context reduction (Peyton Jones et al. 1997) can be chosen from by means of compiler options, pragmas, or via a heuristics language and compiler plugins. We feel that separating proof machinery from solution choice provides a good starting point for making this part of the compilation process fully steerable by a programmer.

Backtracking or choosing between reduction alternatives? Our CHRs generate a reduction graph which represents reduction alternatives from which we heuristically choose. Alternatively, a backtracking CHR version (Frühwirth 1998; Frühwirth and Abdennadher 2003) could be used instead. Consequently, choices between alternatives then are expressed as part of CHRs. It is unclear whether implicitly enforcing choices offers enough or better expressability compared to a Haskell program dealing explicitly with reduction alternatives.

Improving substitution related extensions Extensions like functional dependencies (Jones 2000) require improving substitutions for type variables in class predicates. However, different reduction alternatives coexist, likely using different functional dependencies and resulting improving substitutions. Consequently, such substitutions, which we do not model, need to be carried along with reduction alternatives.

References

Manuel M.T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated Type Synonyms. In *International Conference on Functional Programming*, pages 241 – 253, 2005a.

Manuel M.T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated Types with Class. In *Principles of Programming Languages*, pages 1 – 13, 2005b.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

Derek Dreyer, Robert Harper, Manuel M.T. Chakravarty, and Gabriele Keller. Modular Type Classes. In *Principles of Programming Languages*, pages 63 – 70, 2007.

Martin Erwig. Inductive Graphs and Functional Graph Algorithms. *Journal of Functional Programming*, 11(5):467 – 492, 2001.

Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.

Bastiaan Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, Institute of Information and Computing Sciences, 2005.

Bastiaan Heeren and Jurriaan Hage. Type Class Directives. In *Seventh International Symposium on Practical Aspects of Declarative Languages*, pages 253 – 267. Springer-Verlag, 2005.

Bastiaan Heeren and Arjan van IJzendoorn. Helium, for learning Haskell. `http://www.cs.uu.nl/helium/`, 2005.

Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000,*, March 2000.

Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, 1993.

Mark P. Jones. *Qualified Types, Theory and Practice*. Cambridge Univ. Press, 1994.

Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

Stefan Kaes. Parametric overloading in polymorphic programming languages . In *Proc. 2nd European Symposium on Programming*, 1988.

Wolfram Kahl and Jan Scheffczyk. Named Instances for Haskell Type Classes. In *Haskell Workshop*, 2001.

Oleg Kiselyov and Chung-chieh Shan. Implict configuration - or, type classes reflect the value of types. In *Haskell Workshop*, 2004.

Jeffrey R. Lewis, Mark B. Shields, Erik Meijer, and John Launchbury. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*, pages 108–118, January 2000.

Simon Marlow. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2004.

Simon Peyton Jones. *Haskell 98, Language and Libraries, The Revised Report*. Cambridge Univ. Press, 2003.

Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types, 2004.

Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

Jan Scheffzcyk. Named Instances for Haskell Type Classes. Master's thesis, Universitat der Bundeswehr München, 2001.

Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. Technical Report TR2002/2, Dept. of Computer Science and Software Engineering, The University of Melbourne, Parkville 3052, Australia, June 2002.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Existentially Quantified Type Classes. `http://www.comp.nus.edu.sg/~sulzmann/`, 2004.

Martin Sulzmann. Polymorphism in Hindley/Milner Style Type Systems with Constraints. `http://www.comp.nus.edu.sg/~sulzmann/`, 1998.

Martin Sulzmann, Tom Schrijvers, and Peter J. Stuckey. Principal Type Inference for GHC-Style Multi-Parameter Type Classes. In *ASIAN Symposium on Programming Languages and Systems*, 2006.

Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. Understanding Functional Dependencies via Constraint Handling Rules. *Journal of Functional Programming*, 17(1):83 – 129, Jan 2007.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy Types: Inference for Higher-Rank Types and Impredicativity. In *ICFP*, pages 251–262, 2006.

Phil Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, 1988.